Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together

Myoungji Han[†]
mjhan@theory.snu.ac.kr

Hyunjoon Kim[†]
hjkim@theory.snu.ac.kr

Geonmo Gu[†]
gmgu@theory.snu.ac.kr

Kunsoo Park[*†]
kpark@theory.snu.ac.kr

Wook-Shin Han[*‡]
wshan@dblab.postech.ac.kr

ABSTRACT

Subgraph matching (or subgraph isomorphism) is one of the fundamental problems in graph analysis. Extensive research has been done to develop practical solutions for subgraph matching. The state-of-the-art algorithms such as CFL-Match and Turbo$_{iso}$ convert a query graph into a spanning tree for obtaining candidates for each query vertex and obtaining a good matching order with the spanning tree. However, by using the spanning tree instead of the original query graph, it could lead to lower pruning power and a sub-optimal matching order. Another limitation is that they perform redundant computation in search without utilizing the knowledge learned from past computation. In this paper, we introduce three novel concepts to address these inherent limitations: 1) dynamic programming between a directed acyclic graph (DAG) and a graph, 2) adaptive matching order with DAG ordering, and 3) pruning by failing sets, which together lead to a much faster algorithm DAF for subgraph matching. Extensive experiments with real datasets show that DAF outperforms the fastest existing solution by up to orders of magnitude in terms of recursive calls as well as in terms of the elapsed time.

[*]contact author
[†]Seoul National University, Korea
[‡]Pohang University of Science and Technology (POSTECH), Korea

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319880

CCS CONCEPTS

• **Information systems** → *Information retrieval query processing*.

KEYWORDS

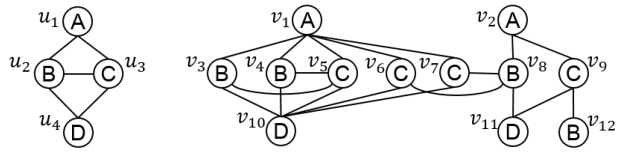
subgraph matching; subgraph isomorphism; dynamic programming; adaptive matching order; failing set

ACM Reference Format:
Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319880

1 INTRODUCTION

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 matching* (also known as *subgraph isomorphism*). Given a data graph G and a query graph q, subgraph matching 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, 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 matching has a wide range of applications including RDF query processing [21], protein interaction analysis [31], chemical compound search [45], and social network analysis [12, 37]. In such applications, subgraph matching is a bottleneck in the overall performance because it is one of the well-known NP-hard problems [15, 42].

Extensive research has been done to develop practical solutions for subgraph matching. Most practical solutions (including VF2 [9], QuickSI [35], GraphQL [17], GADDI [46], SPath [47], Turbo$_{iso}$ [16], and CFL-Match [5]) are based on the backtracking approach [26, 44], which recursively extends a partial embedding by mapping the next query vertex



(a) Query graph q **(b) Data graph G**

Figure 1: Query graph and data graph of subgraph matching.

to a data vertex. In Figure 1, $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_5)\}$ is a partial embedding, and u_4 is the next query vertex to map. 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 set of vertices in G which u can be mapped to. 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.

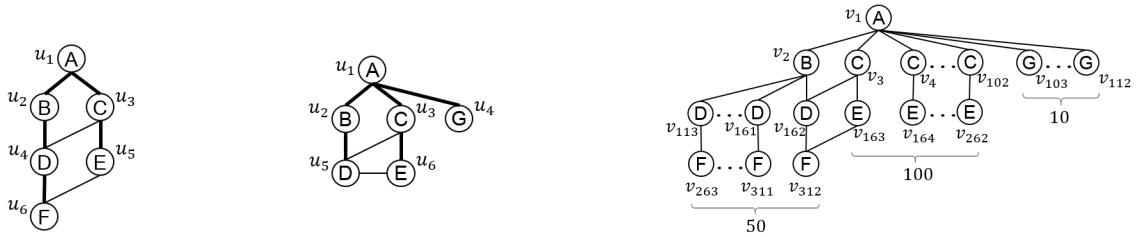
The state-of-the-art algorithms $\text{Turbo}_{\text{iso}}$ [16] and CFL-Match [5] use a spanning tree q_T of query graph q for a filtering process which finds (potential) embeddings of q_T in data graph G. CFL-Match additionally filters using non-tree edges (i.e., edges of q that are not in q_T). The candidate sets computed by the filtering process are stored into an auxiliary data structure, i.e., CR [16] 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 select an effective matching order.

Although the above framework seems to be standard, even the state-of-the-art algorithms show limited response time and scalability in handling real-world applications [28, 38]. There are several challenges we need to deal with to solve subgraph matching 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. Consider query graph q and data graph G in Figure 2. The spanning tree q_T (depicted by thick edges in Figure 2a) has two root-to-leaf paths $p_1 = (u_1, u_2, u_4, u_6)$ and $p_2 = (u_1, u_3, u_5)$, in which p_1 has 50 embeddings (i.e., matching paths) in G starting from v_1, and p_2 has 100 embeddings in G also starting from v_1. Hence there are 50×100 embeddings of q_T in G, most of which will not lead to an embedding of q. 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.

(2) **Sub-optimal Matching Order.** The state-of-the-art algorithms $\text{Turbo}_{\text{iso}}$ and CFL-Match use path-ordering techniques which determine the matching order by ordering the root-to-leaf paths of spanning tree q_T.
For query graph q and data graph G in Figure 2, suppose that there are 50 embeddings of p_1 and 100 embeddings of p_2 after the filtering process, and the path ordering selects p_1 and then p_2 as the matching order. Then backtracking will generate 50×100 partial embeddings. This phenomenon is called the *redundant Cartesian products* in [5]. But, if we check the non-tree edge (u_3, u_4) earlier, we can eliminate most of them. That is, non-tree edges have a strong pruning power and so they should be processed as early as possible during backtracking. 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 (the query graph in Figure 2a is a core in itself).
In addition, the path ordering selects a global matching order so that the precomputed matching order is used throughout the whole search process. Though $\text{Turbo}_{\text{iso}}$ [16] 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 [47] dynamically selects the next matching vertex considering the current partial embedding, but its matching order selection is found to be far from optimal [26].

(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. Consider query graph q' and data graph G in Figure 2. Assuming the path ordering (i.e., matching $p_3 = (u_1, u_4)$, $p_4 = (u_1, u_2, u_5)$, and $p_5 = (u_1, u_3, u_6)$ in this order), we first match p_3 to (v_1, v_{103}), which will not extend to any embeddings of q', and so we will try other embeddings of p_3 (i.e., (v_1, v_{103+i}) for $1 \le i \le 9$). All the following computations, however, are redundant because we have found that there exist no embeddings of $V(q') \setminus \{u_4\}$ in G when we matched p_3 to (v_1, v_{103}).



(a) Query graph q **(b) Query graph q'** **(c) Data graph G**

Figure 2: Spanning trees of query graphs and challenges in subgraph matching.

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 algorithm DAF for subgraph matching.

(1) **DAG-Graph DP.** 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 *DAG-graph DP*. Since q_D contains all the edges of q, the candidate sets computed by DAG-graph DP 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 to reduce redundant Cartesian products. We also propose two types of adaptive matching order, i.e., *candidate-size order* and *path-size order*.

(3) **Pruning by Failing Sets.** We introduce a new notion of *failing set* to prune out some parts of the search space. A failing set is a subset F of $V(q)$ associated with a partial embedding such that there exist no embeddings of F in G which are extensions of the partial embedding. For query graph q' and data graph G in Figure 2, $V(q') \setminus \{u_4\}$ is a failing set associated with the partial embedding $\{(u_1, v_1), (u_4, v_{103})\}$. 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 DAF, 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. Our experiments show that DAF outperforms the fastest existing algorithm CFL-Match by up to 4 orders of magnitude with respect to elapsed time and up to 6 orders of magnitude with respect to the number of recursive calls. We conduct a parameter sensitivity analysis to better understand the performance of subgraph matching. We also give an experiment with billion-scale graph Twitter.

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

2 PRELIMINARIES

In this paper we focus on undirected, connected, and vertex-labeled graphs. All techniques in this paper 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 Σ 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 $\deg_g(v)$, is the number of vertices that are adjacent to v in g. The *average degree* of a graph g, denoted by avg-deg(g), is $\frac{\sum_{v \in V(g)} \deg(v)}{|V(g)|}$. 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.

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 *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 matching problem* is to find all distinct embeddings of q in G.

The subgraph matching problem is NP-hard [15], 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 matching stop after finding k embeddings for some k in practical settings.

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. Frequently used notations are summarized in Table 1.

2.1 Related Work

1) Subgraph Matching. The study of practical subgraph matching algorithms was initiated by Ullmann's backtracking algorithm [44]. 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 [5, 9, 16, 17, 35, 46, 47]. Most algorithms are nicely surveyed and compared in depth [20, 26, 28]. In this paper we focus on two fastest algorithms $\text{Turbo}_{\text{iso}}$ [16] and CFL-Match [5], both of which use the general framework of backtracking in Section 1.

$\text{Turbo}_{\text{iso}}$ [16] uses the idea that an optimal matching order varies with each region of the data graph, and thus $\text{Turbo}_{\text{iso}}$ 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.

Table 1: Frequently used notations.

Notation	Description
q and G	Query graph and data graph
$V(g)$, $E(g)$, L_g	Vertex set, edge set, label function of g
$\deg_g(v)$	Degree of vertex v in g
avg-deg(g)	Average degree of g
$g[S]$	Subgraph of g induced by $S \subseteq V(g)$
$M(u)$	Mapping of u in (partial) embedding M
$C(u)$	Candidate set for query vertex u
q_D	Query DAG
$C_M(u)$	Set of extendable candidates of u regarding partial embedding M
$w_M(u)$	Weight of extendable vertex u regarding M
F_M	Failing set of partial embedding M

CFL-Match [5] is motivated by the inherent problems of $\text{Turbo}_{\text{iso}}$. The first problem is that the path-ordering technique may delay the checking of non-tree edges and generate many redundant Cartesian products 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 $\text{Turbo}_{\text{iso}}$, the exponential size of the CR structure, CFL-Match proposes a more compact auxiliary structure *CPI*.

Other Work. 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 [33] proposes a boosting technique that speeds up the existing subgraph matching algorithms by compressing similar vertices in the data graph. $\text{MQO}_{\text{subiso}}$ [34] proposes a multi-query optimization technique which revises existing algorithms to handle multiple query graphs efficiently. Efficient subgraph matching on streaming graphs has also been studied [8, 14, 22]. There are an increasing number of studies on the problem of subgraph matching in a distributed environment [3, 24, 36], where the *join paradigm* has been shown as the most popular strategy. Also, there are many interesting results on other related problems such as graph similarity [48], graph simulation [27], diversified top-k matching [13], etc [7, 30, 50].

2) 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 [2, 25, 43] such as approximate string matching, and it is also used to solve tree pattern matching and mining [32, 40, 41].

Dynamic programming has been shown to be an effective tool for subgraph matching when graphs are restricted to some special classes. For example, Bodlaender [6] showed that the decision problem of subgraph matching (i.e., "Does G contain a subgraph isomorphic to q?") is solvable in polynomial time for data graphs with bounded tree-width and degree. Alon et al. [1] proposed a polynomial-time subgraph matching algorithm when query graph q has a bounded tree-width and $|V(q)| = O(\log |V(G)|)$. Eppstein [11] proposed a linear-time algorithm for subgraph matching 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) [4, 6, 39]. A main technique in these studies 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.

3 OVERVIEW OF DAF

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

Algorithm 1: DAF

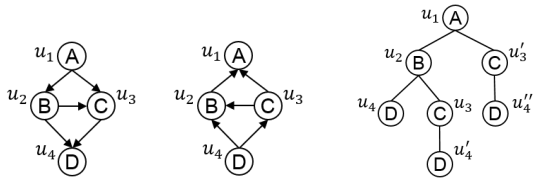
Input: query graph q, data graph G
Output: all embeddings of q in G

1 $q_D \leftarrow$ BuildDAG(q, G);
2 $CS \leftarrow$ BuildCS(q, q_D, G);
3 $M \leftarrow \emptyset$;
4 Backtrack(q, q_D, CS, M);

DAF consists of the following three procedures:

(1) Initially, BuildDAG is invoked to build a rooted DAG q_D from q. In BuildDAG, 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 $r \leftarrow \text{argmin}_{u \in V(q)} \frac{|C_{\text{ini}}(u)|}{\deg_q(u)}$, where the initial candidate set $C_{\text{ini}}(u)$ 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 upper levels to lower levels. In the same level, the directions between vertices are decided if we decide the order of the vertices. The vertices in the same level are grouped by labels (one group per label) and then the groups are sorted so that labels that are more infrequent in the data graph come earlier. The vertices within each group are sorted in descending order of vertex degrees. For example, Figure 3a shows a rooted DAG built from query graph q in Figure 1a when u_1 is the root.
(2) BuildCS is invoked to build the CS structure by using DAG-graph DP (Section 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 an adaptive matching order based on DAG ordering (Section 5). Backtrack also uses a pruning technique by failing sets (Section 6).

We adopt the leaf decomposition strategy of [5], i.e., 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 paper are $q[V']$.



(a) q_D **(b)** q_D^{-1} **(c) Path tree of** q_D

Figure 3: Query DAGs and path tree.

4 DAG-GRAPH DP

In this section we describe a technique called *DAG-graph DP*, by which we construct the CS structure that serves as a complete search space for all embeddings of q in G.

CS Structure. Given a query graph q and a data graph G, a *CS structure on q and G* consists of the candidate set $C(u)$ for each $u \in V(q)$ and edges between the candidates satisfying the following conditions.

(1) For each $u \in V(q)$, there is a candidate set $C(u)$, which is a subset of $C_{\text{ini}}(u)$.
(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 $(v, v') \in E(G)$.

Definition 4.1. An *embedding of q in the CS* is defined as an injective mapping $M : V(q) \rightarrow V(G)$ such that (i) $M(u) \in C(u)$ for every $u \in V(q)$, and (ii) there is an edge between $M(u)$ and $M(u')$ in the CS for every $(u, u') \in E(q)$.

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.

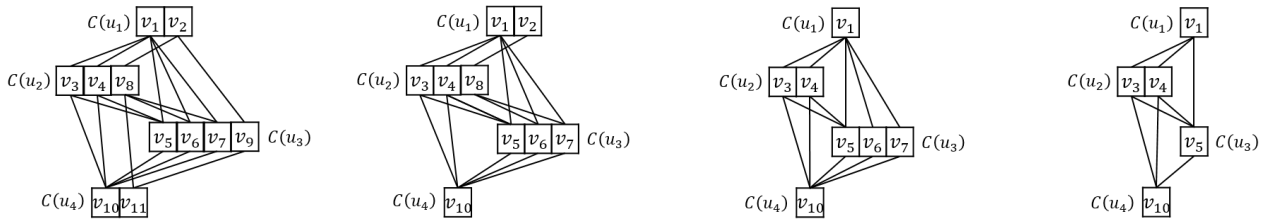
Definition 4.2. A CS structure on q and G is *sound* if it satisfies the following requirement: If there is an embedding M of q in G such that $M(u) = v$, then v must be in $C(u)$ of the CS.

Definition 4.3. A CS structure on q and G is *equivalent* to G with respect to q if the set of all embeddings of q in G is the same as the set of all embeddings of q in the CS.

Since every edge in q is accounted for in CS, we have the following *equivalence property*.

Theorem 4.1. *If a CS structure on q and G is sound, it is equivalent to G with respect to q.*

Proof. Suppose that M is an embedding of q in G. We show that M is also an embedding of q in the sound CS as follows. For every $u \in V(q)$, $M(u)$ is in $C(u)$ of the sound CS by Definition 4.2. For every edge $(u, u') \in E(q)$, there exists edge $(M(u), M(u')) \in E(G)$ because M is an embedding of q



(a) Initial CS **(b) CS after 1st refinement** **(c) CS after 2nd refinement** **(d) CS after 3rd refinement**

Figure 4: Refinements of CS using DAG-graph DP.

in G. Since $(u, u') \in E(q)$ and $(M(u), M(u')) \in E(G)$, there is an edge between $M(u) \in C(u)$ and $M(u') \in C(u')$ by Condition (2) in the definition of the CS structure. Hence M is an embedding of q in the CS.

Also, an embedding that is not in G cannot be created in the CS because the CS is sound and it does not have an edge (v, v') that is not in $E(G)$. □

This equivalence property does not hold in the auxiliary data structure CR of $\text{Turbo}_{\text{iso}}$ [16] or CPI of CFL-Match [5] due to non-tree edges of q. 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. Once we compute a compact sound CS, data graph G is no longer necessary for the remaining part of our algorithm by the equivalence property.

DAG-Graph DP. We first define the *weak embedding*, which is a key notion in our dynamic programming.

Definition 4.4. The *path tree* of a rooted DAG g is defined as the tree g' such that each root-to-leaf path in g' corresponds to a distinct root-to-leaf path in g, and g' shares common prefixes of its root-to-leaf paths (e.g., Figure 3c is the path tree of q_D in Figure 3a).

Definition 4.5. For a rooted DAG g with root u, a *weak embedding* M' of g at $v \in V(G)$ is defined as a homomorphism 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 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, $\{(u_1, v_1), (u_2, v_4), (u_4, v_{10}), (u_3, v_5), (u'_4, v_{10}), (u'_3, v_6), (u''_4, v_{10})\}$ is a weak embedding of q_D (Figure 3a) in G (Figure 1b), where u_3 in q_D is mapped to two different vertices v_5 and v_6 of data graph G via the path tree (Figure 3c). 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.

Definition 4.6. 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 its reverse q_D^{-1} in Figure 3 are query DAGs).

Given a sound CS and a query DAG q' (q' can be q_D or q_D^{-1}), we will refine the CS by checking weak embeddings of q' as follows. First, we have the following observation:

- If there is an embedding M of q in the CS such that $M(u) = v$ for a vertex $u \in V(q)$, 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 because a weak embedding is a necessary condition for an embedding.

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, if we apply the above refinement to the CS in Figure 4b by using the rooted DAG q_D in Figure 3a as query DAG q', we get Figure 4c as the result of the refinement. 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 of Figure 4b.

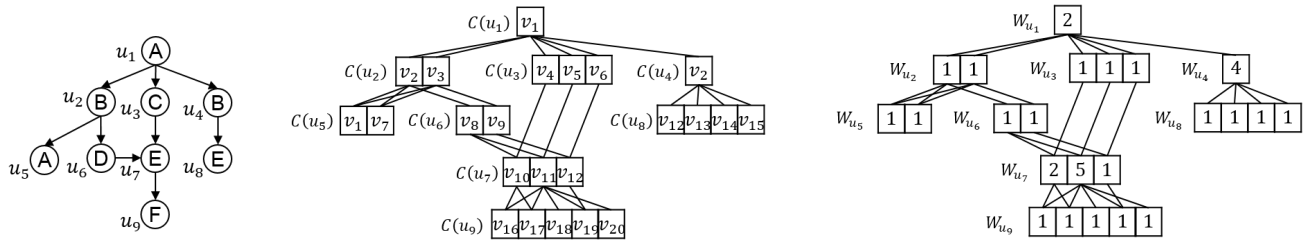
To compute $C'(u)$ by dynamic programming, we obtain the following recurrence:

$$v \in C'(u) \text{ iff } v \in C(u) \text{ and } \exists v_c \text{ adjacent to } v \text{ such that } v_c \in C'(u_c) \text{ for every child } u_c \text{ of } u \text{ in } q'. \quad (1)$$

One can prove by induction that Recurrence (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 *DAG-graph DP*.

Lemma 4.1. *Given a CS on q and G, the time complexity of DAG-graph DP on the CS is $O(|E(q)| \times |E(G)|)$.*

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



(a) Rooted DAG q_D **(b) CS structure** **(c) Weight array**

Figure 5: New running example, its CS, and weight array in path-size order.

of q'_u at v in the CS). Then Recurrence (1) would be:

$D'[u, v] = 1$ iff $D[u, v] = 1$ and $\exists v_c$ adjacent to v such that $D'[u_c, v_c] = 1$ for every child u_c of u in q'.

D is a typical dynamic programming table [10] between query DAG q' and data graph G, and the CS structure is in fact a compact representation of dynamic programming table D.

Optimizing CS. Now we describe our strategy to compute a compact CS. Initially, $C(u)$ is set to $C_{\text{ini}}(u)$ 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 4a shows the initial CS for the example in Figure 1.

Since DAG-graph DP refines the CS by using a query DAG, we will use the rooted DAG q_D and its reverse q_D^{-1} to refine the CS. (If we convert an undirected graph to a directed graph, an edge of the undirected graph becomes two antiparallel edges in the directed graph [10].) By using both q_D and q_D^{-1}, we use *all edges* in the directed version of q. In the first step of the refinements we perform DAG-graph DP using q_D^{-1} to the initial CS. In this first step, candidates may be further filtered by considering the local features of the query vertices (e.g., maximum neighbor degree [5], neighborhood label frequency [5, 16]). In the second step, we further refine the CS using q_D. Note that the changes of the candidate sets by the second step give a chance of further refinement using q_D^{-1} again, and so on. Consequently, we can optimize the CS by repeating DAG-graph DP with alternating q_D and q_D^{-1} until no changes occur in the candidate sets.

Figure 4 shows an example of the refinements of the CS regarding the running example and the DAGs (i.e., q_D and q_D^{-1}) in Figure 3. Note that the candidate sets in Figure 4d are optimized since no more refinement is 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 refinements, therefore, we maintain candidate sets $C(u)$ only. The edges depicted in Figure 4 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 4d). The edges are stored in the CS as an adjacency list $N^u_{u_c}(v)$ for each $v \in C(u)$ and each edge $(u, u_c) \in E(q_D)$, where $N^u_{u_c}(v)$ is the list of vertices v_c adjacent to v in G such that $v_c \in C(u_c)$. For example, in Figure 4d, $N^{u_1}_{u_2}(v_1) = \{v_3, v_4\}$.

5 DAG ORDERING AND ADAPTIVE MATCHING ORDER

In this section 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 5.1. As a new running example, we use the rooted DAG q_D of q in Figure 5a (q is easily inferred from q_D) and the CS in Figure 5b, which is constructed by our CS optimization technique in Section 4. There are 3 embeddings of q in the CS, i.e., $\{(u_1, v_1), (u_2, v_3), (u_3, v_6), (u_4, v_2), (u_5, v_7), (u_6, v_9), (u_7, v_{12}), (u_8, v_{13+i}), (u_9, v_{19})\}$, where $0 \leq i \leq 2$. 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 DAG Ordering

We now propose a new backtracking framework for subgraph matching based on the *DAG ordering*. Initially, we map the root vertex r of q_D to one of its candidates in $C(r)$ of the CS.

Definition 5.1. An unvisited (i.e., unmapped) vertex u of query DAG q_D in a partial embedding M is called *extendable* regarding M if all parents of u are matched in M. A *DAG ordering* always selects an extendable vertex as the next vertex to map.

Since the edges of query graph q are the source of the pruning power, we want the edges to be checked as early as possible. For that purpose, a query vertex u should be processed after all its parents in q_D have been processed, which is ensured by a DAG ordering.

In the CS, we can obtain the candidates of an extendable vertex in a DAG ordering as follows.

Definition 5.2. 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 $C_M(u) = \bigcap_{i=1}^{k} N_u^{p_i}(M(p_i))$.

Example 5.2. 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, 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 $C_M(u_7) = N_{u_7}^{u_3}(v_5) \cap N_{u_7}^{u_6}(v_8) = \{v_{11}\} \cap \{v_{10}, v_{11}\} = \{v_{11}\}$.

Lemma 5.1. *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 partial embedding.*

In Example 5.2, $M \cup \{(u_7, v_{11})\}$ is a partial embedding, but we cannot extend M to u_4 because the only extendable candidate $v_2 \in C_M(u_4)$ is already visited.

Backtracking Framework. Based on Lemma 5.1, 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? We answer this question in the following subsection.

5.2 Adaptive Matching Order

We propose two adaptive methods to select the next extendable vertex. Suppose that we are trying to extend a partial embedding M.

- Candidate-size order: we select an extendable vertex u such that $|C_M(u)|$ is the minimum. Since $C_M(u)$ is the set of extendable candidates of u, this is called the *candidate-size order*.
- Path-size order: we select an extendable vertex u such that $w_M(u)$ is the minimum. Since $w_M(u)$ (which is defined below) is an estimate of path embeddings, this is called the *path-size order*.

Since $|C_M(u)|$ or $w_M(u)$ is computed regarding the partial embedding M, the next vertex selected in each order may be different for different partial embeddings, i.e., both orders above are adaptive matching orders.

Path-Size Order. The common strategy of the path-ordering techniques [5, 16] 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 [5, 16, 47]. 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 [5, 16]). To address this issue, we propose the notion of *tree-like paths*.

Definition 5.3. Given a rooted DAG q_D, a path p in q_D starting from $u \in V(q_D)$ 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 5a are $\{(u_1, u_2, u_5), (u_1, u_2, u_6), (u_1, u_3), (u_1, u_4, u_8)\}$. The maximal tree-like path starting from u_7 is (u_7, u_9).

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.

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_u be the set of all maximal tree-like paths starting from u in q_D. For each path $p \in P_u$, we define $n(p, v)$ as the number of paths starting from v in the CS corresponding to p. In Figure 5, $n(p, v_1) = 4$ 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+i}, v_7), where $0 \le i \le 1$). Now we define $W_u(v) = \min_{p \in P_u} n(p, v)$. 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 5.3. Figure 5c shows the weight array for the CS in Figure 5b. 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 $W_u(v)$ 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 which have 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 $W_{u,c_i}(v) =$

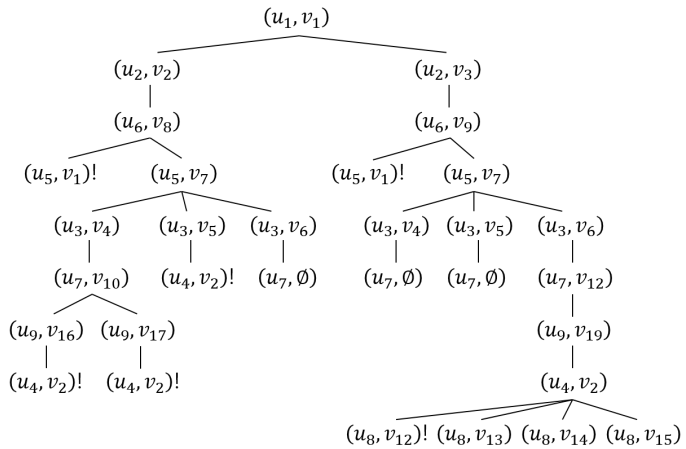


Figure 6: Search tree of backtracking for running example.

$\sum_{v' \in N^u_{c_i}(v)} W_{c_i}(v')$. Then, we set $W_u(v) = \min_{1 \le i \le k} W_{u,c_i}(v)$.

The weight (denoted by $w_M(u)$) of an extendable vertex u of q_D regarding a partial embedding M is computed as

$$w_M(u) = \sum_{v \in C_M(u)} W_u(v).$$

Example 5.4. Consider the running example in Figure 5 and the search tree in Figure 6. 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 in the path-size order. 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 (i.e., Backtrack) shows the backtracking process of DAF with the path-size order. (For the candidate-size order, $w_M(u)$ in Algorithm 2 should be replaced by $|C_M(u)|$.) It finds all embeddings of q in the CS by extending a partial embedding M. If $|M| = |V(q)|$, M is an embedding of q 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 an 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 an extendable 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 6 illustrates the search process of Backtrack for the running example in Figure 5. A node

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 ∈ C(r) do
5          M ← {(r, v)}; Mark v as visited;
6          Backtrack(q, q_D, CS, M); Mark v as unvisited;
7  else
8      u ← extendable vertex with min weight w_M(u);
9      foreach v ∈ C_M(u) do
10         if v is unvisited then
11             M' ← M ∪ {(u, v)}; Mark v as visited;
12             Backtrack(q, q_D, CS, M'); Mark v as
               unvisited;
13 return;
```

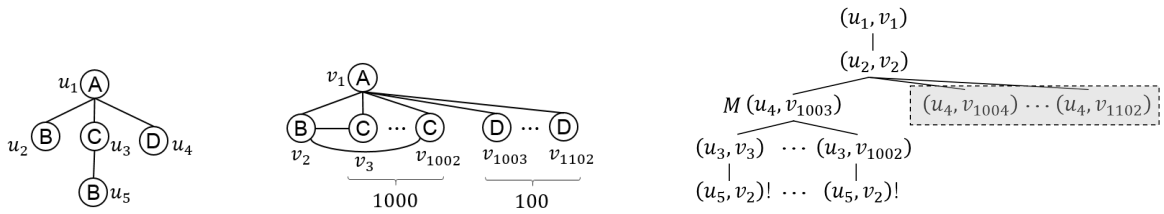
in the search tree in general corresponds to a partial embedding. For example, the root of the search tree in Figure 6 corresponds to partial embedding $M_1 = \{(u_1, v_1)\}$, its left child (labeled with (u_2, v_2)) corresponds to $M_2 = \{(u_1, v_1), (u_2, v_2)\}$, and so on. Thus, we use the mapping 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 6 may not correspond to partial embeddings, we represent them by mapping functions with '!' and $\emptyset$, where '!' 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.

The time complexity of backtracking is bounded by $\prod_{u \in V(q)} |C(u)|$, which is $O(|V(G)|^{|V(q)|})$ in the worst case [11, 47]. However, actual running time depends on the size of the search tree (i.e., number of recursive calls) that an algorithm generates. Hence we will count the number of recursive calls in performance evaluation.

6 PRUNING BY FAILING SETS

In this section we develop a new technique to prune out some parts of the search space by using the notion of failing sets. 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. Example 6.1 gives a motivating example.

Example 6.1. Figure 7c is a search tree for query graph q in Figure 7a and data graph G in Figure 7b. Suppose that we just came back to node $M = \{(u_1, v_1), (u_2, v_2), (u_4, v_{1003})\}$ after



(a) Query graph q **(b) Data graph G** **(c) Search tree**

Figure 7: Example of pruning by failing sets. Nodes in the dashed box are redundant.

the exploration of the subtree rooted at M. Recall that we use the mapping function M to represent a node as well as a partial embedding. During the exploration, we have tried all possible extensions to map u_3 and u_5, and all attempts to map u_5 have failed by the conflicts in the mappings of u_2 and u_5. However, u_4 is not relevant to any of the failures because u_4 is not involved in the conflicts. It means that no matter how we change the mapping of u_4 in the current mapping M, it will never lead to an embedding of q because all possible extensions will end up with failures in the same way. Thus, we need not extend all other siblings of node M (depicted by a dashed box in Figure 7c)

6.1 Failing Set

In Example 6.1 there are two main ingredients:

(1) A set of query vertices relevant to any of the failures during the exploration of the subtree rooted at M, which is $F = \{u_1, u_2, u_3, u_5\}$ (i.e., u_2 and u_5 that are involved in the conflicts, and their ancestors).
(2) A subset $M' = \{(u_1, v_1), (u_2, v_2)\}$ of M as a partial embedding of induced subgraph $q[F]$.

Suppose that we search for an embedding of $q[F]$ starting from M' by mapping u_3 and u_5. Then we will go through the same failures as those in the exploration of the subtree rooted at M. That is, there are no embeddings of $q[F]$ by extending M'. Since all the siblings of M also contain M', all the siblings of M cannot lead to an embedding of $q[F]$. We call the set F a *failing set* and it is the key to our pruning technique. We will first give a formal definition of the failing set and then describe how to compute it.

Definition 6.1. We say that a set of query vertices $S \subseteq V(q)$ is *ancestor-closed* in q_D if for any $u \in S$, all the ancestors of u in q_D are also in S.

For a set of query vertices $S \subseteq V(q)$ and a node M in the search tree, let $M[S]$ denote the largest subset of M that is a partial embedding of $q[S]$, i.e., $(u, v) \in M[S]$ if and only if $(u, v) \in M$ and $u \in S$.

Definition 6.2. We define a *failing set* $F_M \subseteq V(q)$ of node M as an ancestor-closed set satisfying the following *failure property*: Given a partial embedding $M[F_M]$, there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M]$.

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 6.1, $F_M = \{u_1, u_2, u_3, u_5\}$ is a failing set of node M. Note that F_M is ancestor-closed and 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.

Lemma 6.1. *Let M be a search tree node labeled with (u, v) (i.e., M is a partial embedding in which the last mapping is (u, v)), and F_M be a non-empty failing set of node M. If $u \notin F_M$, then all siblings of node M (including itself) are redundant.*

Proof. A sibling M' of M maps u to another vertex v' ($\neq v$) of G, and so all the mappings in M' except (u, v') are the same as the mappings in M except (u, v). Since u is not in F_M, $M[F_M]$ is a subset of M'. By definition of the failing set, there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M]$. Hence, M' (which includes $M[F_M]$) cannot lead to an embedding of $q[F_M]$. Since $q[F_M]$ is a subgraph of q, therefore, M' cannot lead to an embedding of q. □

In Example 6.1, since $u_4 \notin F_M$, we can immediately conclude that all siblings of node M are redundant by Lemma 6.1.

Computing Failing Sets. 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 6. For a conflict-class node $M = \{\ldots, (u', v), \ldots, (u, v)!\}$, we set $F_M = \text{anc}(u) \cup \text{anc}(u')$, where $\text{anc}(u)$ denotes the set of all ancestors of u in q_D including u itself. Clearly, F_M is ancestor-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 6. For an emptyset-class node $M = \{\ldots, (u, \emptyset)\}$, we set $F_M = \text{anc}(u)$. Clearly, F_M is ancestor-closed and satisfies the failure property.

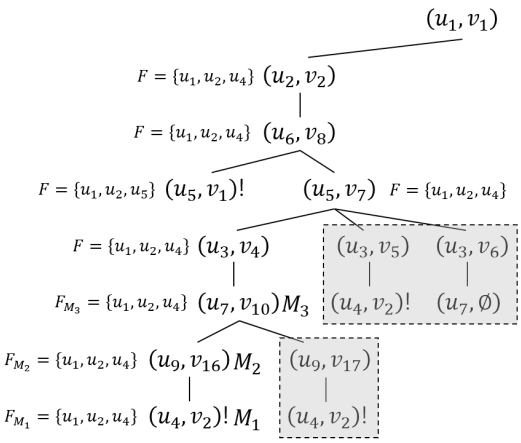


Figure 8: Pruned search tree. Redundant nodes enclosed by dashed boxes are pruned by failing sets.

(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 $F_M = \bigcup_{i=1}^{k} F_{M_i}$.

Example 6.2. Figure 8 shows the failing set of each node for the left half of the search tree in Figure 6. The leaf $M_1 = \{(u_1, v_1), (u_2, v_2), (u_6, v_8), (u_5, v_7), (u_3, v_4), (u_7, v_{10}), (u_9, v_{16}), (u_4, v_2)!\}$ belongs to the conflict-class, and so $F_{M_1} = \text{anc}(u_2) \cup \text{anc}(u_4) = \{u_1, u_2, u_4\}$. Since M_1's parent M_2 corresponds to Case 2.2, we set $F_{M_2} = \{u_1, u_2, u_4\}$ as the union of the failing sets of M_2's children (M_1 is the only child). Since $u_9 \notin F_{M_2}$, the siblings of M_2 are redundant by Lemma 6.1 and we can prune them. M_2's parent M_3 corresponds to Case 2.1 due to $u_9 \notin F_{M_2}$, so we set $F_{M_3} = F_{M_2}$.

The failing set is implemented as a bit-array ($|V(q)|$ bits per search tree node) so that union and membership operations can be done efficiently (i.e., union in $O(|V(q)|/w)$ time, where w is the word size, and membership in $O(1)$ time). To prevent the computational overhead during backtracking, $\text{anc}(u)$ for every $u \in V(q)$ is computed and stored in advance.

7 PERFORMANCE EVALUATION

In this section we present experimental results to show the effectiveness of our techniques. For performance evaluation, we compare VF2 [9], QuickSI [35], GraphQL [17], GADDI [46], SPath [47], Turbo$_{\text{iso}}$ [16], CFL-Match [5], and four versions of our algorithm:

- DA-cand and DA-path: using DAG-graph DP and adaptive matching order (candidate-size order and path-size order, respectively) without pruning by failing sets.
- DAF-cand and DAF-path: using DAG-graph DP, adaptive matching order (candidate-size order and path-size order, respectively), and pruning by failing sets.

By an experiment comparing the four versions of our algorithm in Appendix A.6, we choose DAF-path as the final version of our algorithm, called DAF (and DA-path as another version DA, which will be used in Section 7.1).

Since CFL-Match significantly outperformed the other existing algorithms [5], our main experiments on real datasets compare the performances of our algorithm and CFL-Match (Section 7.1). A parameter sensitivity analysis is described in Section 7.2. We also test our algorithm on billion-scale graph Twitter in Appendix A.1. The comparison between all other important existing algorithms and our algorithm will be presented in Appendix A.2. A number of additional experiments are also performed: evaluating on negative queries (Appendix A.3), parallelizing DAF (Appendix A.4), and boosting DAF with the technique in [33] (Appendix A.5).

All the source codes (i.e., CFL-Match, Turbo$_{\text{iso}}$, etc.) were obtained from the authors of previous papers. All algorithms including ours are implemented in C++. Experiments are conducted on a machine with two Intel Xeon E5-2680 v3 2.50GHz CPUs and 256GB memory running CentOS Linux.

Table 2: Characteristics of datasets.

Dataset (G)	$\|V(G)\|$	$\|E(G)\|$	$\|\Sigma\|$	avg-deg(G)
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
DBLP	317,080	1,049,866	20	6.62
YAGO	4,295,825	11,413,472	49,676	5.31

Datasets. We use six real datasets in our experiments: Yeast, Human, HPRD, Email, DBLP, and YAGO, which are commonly used in previous works [5, 16, 26, 33]. As no label information is available for Email and DBLP, we randomly assigned a label out of 20 distinct labels to each vertex. Since YAGO is an RDF dataset, we transformed it into a graph dataset by applying the *type-aware transformation* proposed in [21] (a similar transformation has also been employed in [49]). The characteristics of the datasets are given in Table 2.

Query Graphs. Since CFL-Match is the state-of-the-art algorithm to compare, we basically follow the experimental setting of the CFL-Match paper [5]. For each data graph, we generate 8 query sets Q_{iS}, Q_{iN}, where $i \in \{50, 100, 150, 200\}$ for Yeast and HPRD, and $i \in \{10, 20, 30, 40\}$ for the rest. Each query set contains 100 query graphs having i vertices. Every query graph q in Q_{iS} has avg-deg$(q) \leq 3$ (i.e., Sparse), while every query graph q in Q_{iN} has avg-deg$(q) > 3$ (i.e., Non-sparse). 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 perform a random walk on the data graph until we visit i distinct vertices, from which we obtain a subgraph of G consisting of all the visited vertices and some edges between these vertices.

Performance Measurement. To evaluate an algorithm, we measure the time in milliseconds to process each query graph in a query set. The total elapsed time for a query graph consists 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 subgraph matching is an NP-hard problem, some query graphs may be bad instances (i.e., processing a query graph cannot finish in a reasonable time), and any algorithm for subgraph matching will have bad instances. To address this issue, we set a time limit of 10 minutes for each query graph in a query set. We say that a query graph is *solved* if it finishes within the time limit.

To evaluate an algorithm regarding a query set, we report the average elapsed time and the average number of recursive calls (i.e., examined nodes in the search tree) for the n least time-consuming query graphs, where n is the minimum among the numbers of solved query graphs in the compared algorithms. We also report the percentage of solved query graphs in a query set as in [18].

7.1 Comparing with CFL-Match

In this section we compare DAF to CFL-Match with six real data graphs. To see the effectiveness of each of the three techniques (i.e., DAG-graph DP, adaptive matching order, and pruning by failing sets), we also consider DA in the comparison.

Effectiveness of DAG-Graph DP. To evaluate the effectiveness of DAG-Graph DP in filtering the candidates, we compare the sizes of auxiliary data structures produced by CFL-Match (i.e., CPI) and DAF (i.e., CS). We measure the size of an auxiliary data structure as the sum of sizes of the candidate sets, i.e., $\Sigma_{u \in V(q)}|C(u)|$. Figure 9 shows the results, where the y-axis represents the average size of auxiliary data structures for each query set. We can see that the average size of CS is always smaller than that of CPI. Especially for DBLP (see Figure 9e), the average size of CS is about 3 times smaller. Furthermore, CS has the equivalence property in

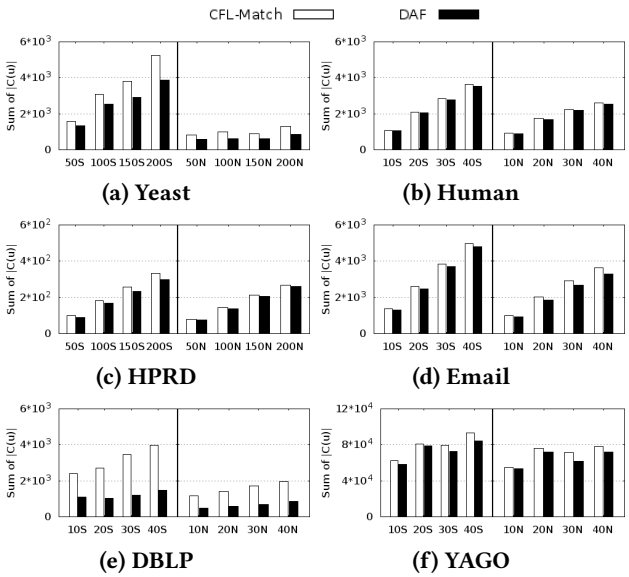


Figure 9: Sizes of auxiliary data structures of CFL-Match (CPI) and DAF (CS).

Theorem 4.1 (which CPI does not have) and thus there is no need to probe the data graph during backtracking.

Performance Comparison. Now we compare the performances of CFL-Match, DA, and DAF. Figure 10 shows the experimental results.

A general trend in Figures 9 and 10 is as follows.

(1) As query sizes increase, the sizes of auxiliary data structures increase (Figure 9), and thus the sizes of search spaces increase as well.
(2) Since a bigger query graph means a larger search space and a fewer embeddings in general, it will become more difficult to find 10^5 embeddings within the time limit. This can be confirmed by the decreasing percentage of solved queries as query sizes increase in Figure 10.

Overall in Figure 10, the best performer is DAF, and then come DA and CFL-Match in this order. DAF consistently outperforms CFL-Match with respect to the percentage of solved queries and the average number of recursive calls. Specifically, CFL-Match solves only 11% of queries while DAF solves 62% of queries for Q_{150S} of Yeast (Figure 10c). Also, CFL-Match solves 76% of queries while DAF solves 100% of queries for Q_{40N} of Email (Figure 10l). In the average number of recursive calls, DAF outperforms CFL-Match by up to 6 orders of magnitude for Yeast (Q_{150S} in Figure 10b), and up to 5 orders of magnitude for Human (Q_{30S} in Figure 10e). In the average elapsed time, DAF outperforms CFL-Match in most cases. Specifically, DAF outperforms CFL-Match in the average elapsed time by up to 4 orders of magnitude for Yeast (Q_{200S} in Figure 10a) and up to 3 orders of magnitude for Human (Q_{30S} in Figure 10d). Note that DAF runs consistently faster than CFL-Match except for HPRD. Regarding HPRD,

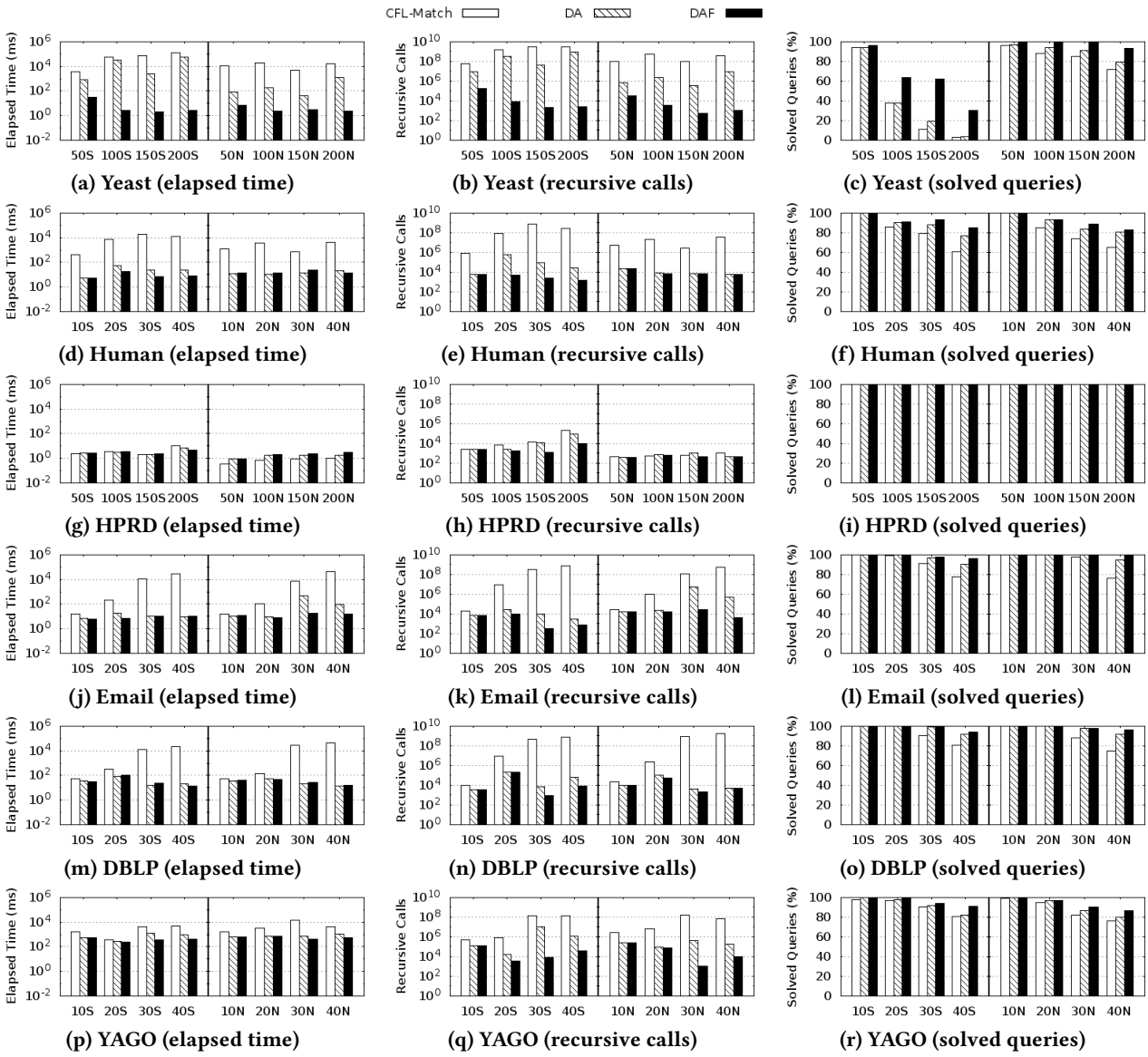
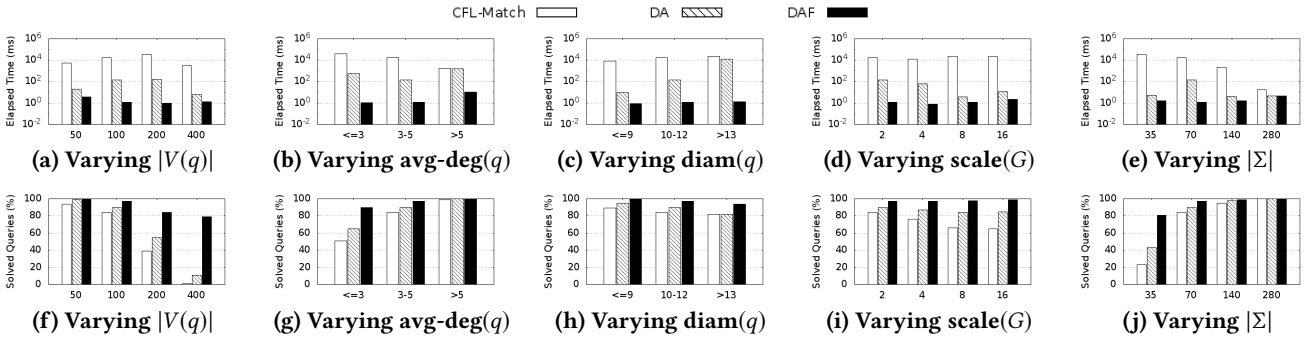


Figure 10: Elapsed time, recursive calls, and solved queries of CFL-Match, DA, and DAF.

we observe that all the query sets are answered within average 10 ms, which means that these queries are easy instances. In this dataset, DAF runs marginally slower than CFL-Match, especially on non-sparse queries. Nevertheless, DAF consistently outperforms CFL-Match in the average number of recursive calls. This discrepancy between the elapsed time and the number of recursive calls is due to the overhead of additional computations during the search of DAF (e.g., computing weights for the adaptive matching order and failing sets), resulting in a higher average cost per recursion.

Effectiveness of Adaptive Matching Order. The effectiveness of DAG-Graph DP and the adaptive matching order can be verified by the performance gap between CFL-Match and DA in Figure 10. We can see that DA consistently outperforms CFL-Match with respect to the percentage of solved queries. The performance gap is significant especially for Email and DBLP. Specifically, DA outperforms CFL-Match by up to 3 orders of magnitude in the average elapsed time and up to 5 orders of magnitude in the average number of recursive calls (see Q_{40S} in Figure 10j, 10k, and Q_{40N} in Figure 10m, 10n). For Human, most of the improvements of DA over CFL-Match are due to the adaptive matching order, since the differences in sizes of auxiliary data structures are very small for Human (Figure 9b).

Effectiveness of Pruning by Failing Sets. The effectiveness of our pruning technique can be verified by the performance gap between DA and DAF in Figure 10. We can see



(a) Varying $|V(q)|$ (b) Varying avg-deg(q) (c) Varying diam(q) (d) Varying scale(G) (e) Varying $|\Sigma|$

(f) Varying $|V(q)|$ (g) Varying avg-deg(q) (h) Varying diam(q) (i) Varying scale(G) (j) Varying $|\Sigma|$

Figure 11: Sensitivity analysis. (a)-(e): elapsed time, (f)-(j): solved queries.

that DAF consistently outperforms DA with respect to the percentage of solved queries and the number of recursive calls. The improvement by the pruning technique is the most significant on Yeast due to large query sizes (i.e., more redundancies in the search space). Specifically, DAF improves DA by up to 4 orders of magnitude in the average elapsed time and up to 5 orders of magnitude in the average number of recursive calls (see Q_{200S} in Figure 10a and 10b). The elapsed time of DAF is worse than that of DA in some cases (see Human and DBLP in Figure 10), even though DAF always solves more queries and makes less recursive calls. This trade-off is due to DAF's use of failing sets, i.e., failing sets are effective in dealing with harder queries, but for easier queries the cost of computing failing sets is more evident.

7.2 Sensitivity Analysis

We conduct a parameter sensitivity analysis to see how various parameters affect the performance of subgraph matching. We generate synthetic data graphs and query graphs. For data graphs we upscale Yeast using EvoGraph [29] and assign labels to vertices according to power-laws. Query graphs are generated similarly as before and those with desired properties (e.g., degree, diameter) are sampled. We vary five types of parameters:

- number of vertices in the query graph: $|V(q)| = 50, 100, 200, 400$,
- average degree of the query graph: avg-deg$(q) \leq 3$, $3 <$ avg-deg$(q) \leq 5$, avg-deg$(q) > 5$,
- diameter of the query graph: diam$(q) \leq 9$, $10 \leq$ diam$(q) \leq 12$, diam$(q) \geq 13$,
- scaling factor of the data graph: scale$(G) = 2, 4, 8, 16$,
- number of labels in the data graph: $|\Sigma| = 35, 70, 140, 280$,

where scale$(G) = x$ means that data graph G is obtained by upscaling Yeast x times in the number of edges (the number of vertices increases as well to keep statistical properties), and diam(q) refers to the diameter of q. The default parameter settings are $|V(q)| = 100$, $3 <$ avg-deg$(q) \leq 5$, $10 \leq$ diam$(q) \leq 12$, scale$(G) = 2$, and $|\Sigma| = 70$.

Figure 11 shows the results for CFL-Match, DA, and DAF. In general, subgraph matching tends to become easier (i.e., more solved queries and less elapsed time) as $|\Sigma|$ or avg-deg(q) increases. This is because more distinct labels and a higher average degree (of a query graph) result in smaller CS. On the other hand, subgraph matching tends to become harder as $|V(q)|$ or diam(q) increases. Especially when $|V(q)| = 400$, CFL-Match solves few queries while DAF solves about 80% of them. Different scales have little effect on the performance (especially of DAF) because statistical properties of graphs do not change by scaling and we find the first 10^5 embeddings in such graphs.

8 CONCLUSION

In this paper we have proposed novel techniques for subgraph matching: dynamic programming between a DAG and a graph, adaptive matching order with DAG ordering, and pruning by failing sets. Extensive experiments on real datasets show that DAF outperforms the state-of-the-art algorithm by up to several orders of magnitude. Applying our techniques to some other problems related to subgraph matching would be an interesting future work. Extending our work to parallel and distributed platforms such as LDBC [19] is also a future work.

ACKNOWLEDGMENTS

H. Kim, G. Gu, and K. Park were supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). W.-S. Han was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No.NRF-2017R1A2B3007116).

REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

[4] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time Computation of Optimal Subgraphs of Decomposable Graphs. *Journal of Algorithms*, 8(2):216–235, 1987.

[5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of SIGMOD*, pages 1199–1214, 2016.

[6] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of ICALP*, pages 105–118, 1988.

[7] Y. Cao, W. Fan, J. Huai, and R. Huang. Making pattern queries bounded in big graphs. In *Proceedings of IEEE ICDE*, pages 161–172, 2015.

[8] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of EDBT*, pages 157–168, 2015.

[9] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[11] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Graph Algorithms And Applications I*, pages 283–309. Society for Industrial and Applied Mathematics, 2002.

[12] W. Fan. Graph pattern matching revised for social network analysis. In *Proceedings ICDT*, pages 8–21, 2012.

[13] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, 6(13):1510–1521, 2013.

[14] W. Fan, X. Wang, and Y. Wu. Incremental Graph Pattern Matching. *ACM Transactions on Database Systems*, 38(3):18:1–18:47, 2013.

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

[16] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of SIGMOD*, pages 337–348, 2013.

[17] H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proceedings of SIGMOD*, pages 405–418, 2008.

[18] V. Ingalalli, D. Ienco, P. Poncelet, and S. Villata. Querying RDF Data Using A Multigraph-based Approach. In *Proceedings of EDBT*, pages 12–23, 2016.

[19] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[20] F. Katsarou, N. Ntarmos, and P. Triantafillou. Hybrid algorithms for subgraph pattern queries in graph databases. In *IEEE International Conference on Big Data*, pages 656–665, 2017.

[21] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming Subgraph Isomorphism for RDF Query Processing. *Proceedings of the VLDB Endowment*, 8(11):1238–1249, 2015.

[22] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of SIGMOD*, pages 411–426, 2018.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of International Conference on World Wide Web*, pages 591–600. ACM, 2010.

[24] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable Subgraph Enumeration in MapReduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[25] G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[26] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012.

[27] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems*, 39(1):4, 2014.

[28] T. Ma, S. Yu, J. Cao, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.

[29] H. Park and M.-S. Kim. Evograph: An effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of SIGKDD*, pages 2051–2059. ACM, 2018.

[30] P. Peng, L. Zou, L. Chen, X. Lin, and D. Zhao. Answering Subgraph Queries over Massive Disk Resident Graphs. *World Wide Web*, 19(3):417–448, 2016.

[31] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient Estimation of Graphlet Frequency Distributions in Protein–protein Interaction Networks. *Bioinformatics*, 22(8):974–980, 2006.

[32] P. Rao and B. Moon. Sequencing xml data and query twigs for fast pattern matching. *ACM Transactions on Database Systems*, 31(1):299–345, Mar. 2006.

[33] X. Ren and J. Wang. Exploiting Vertex Relationships in Speeding Up Subgraph Isomorphism over Large Graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[34] X. Ren and J. Wang. Multi-query Optimization for Subgraph Isomorphism Search. *Proceedings of the VLDB Endowment*, 10(3):121–132, 2016.

[35] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[36] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel Subgraph Listing in a Large-scale Graph. In *Proceedings of SIGMOD*, pages 625–636, 2014.

[37] T. A. B. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological Methodology*, 36(1):99–153, 2006.

[38] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient Subgraph Matching on Billion Node Graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[39] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time Computability of Combinatorial Problems on Series-parallel Graphs. *Journal of the ACM*, 29(3):623–641, 1982.

[40] S. Tatikonda and S. Parthasarathy. Mining tree-structured data on multicore systems. *Proceedings of the VLDB Endowment*, 2(1):694–705, 2009.

[41] S. Tatikonda, S. Parthasarathy, and M. Goyder. LCS-TRIM: Dynamic programming meets xml indexing and querying. In *Proceedings of the VLDB Endowment*, pages 63–74, 2007.

[42] H.-N. Tran, J. Kim, and B. He. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Proceedings of DASFAA*, pages 299–315, 2015.

[43] E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, 64(1-3):100–118, 1985.

[44] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

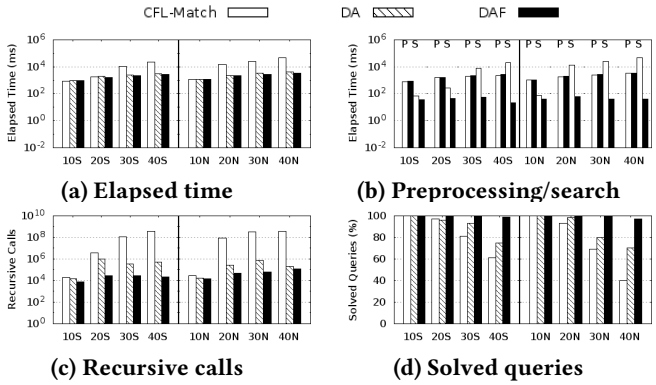
[45] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of SIGMOD*, pages 335–346, 2004.

[46] S. Zhang, S. Li, and J. Yang. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of EDBT*, pages 192–203, 2009.

[47] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

[48] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal*, 22(6):727–752, 2013.

[49] W. Zheng, L. Zou, X. Lian, H. Zhang, W. Wang, and D. Zhao. SQBC: An efficient subgraph matching method over large and dense graphs. *Information Sciences*, 261:116 – 131, 2014.



(a) Elapsed time (b) Preprocessing/search
(c) Recursive calls (d) Solved queries

Figure 12: Elapsed time, recursive calls, and solved queries of CFL-Match, DA, and DAF on Twitter. For CFL-Match and DAF, elapsed time is divided into preprocessing time and search time.

[50] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.

A APPENDIX

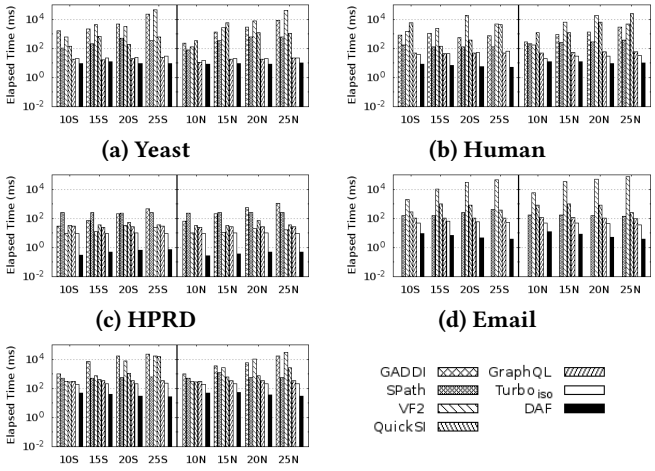
A.1 Experiment with Billion-Scale Graph

Since graphs in real life nowadays are much larger than those used in Section 7, often containing billions of edges, we also tested our algorithm on Twitter graph [23], which has 41.7 million vertices and 1.47 billion edges. We randomly assigned labels out of 1000 distinct labels. Figure 12 shows the results. Since the preprocessing time of CFL-Match and DAF takes quite a portion of the total elapsed time for big graphs, we show the break-down of the elapsed time into preprocessing time and search time in Figure 12b.

DAF outperforms CFL-Match in all metrics. As query sizes increase, the gap between CFL-Match and DAF in solved queries increases. In total elapsed time, DAF is up to 14 times faster than CFL-Match; preprocessing times of CFL-Match and DAF are similar, but in search time DAF is faster than CFL-Match by up to 3 orders of magnitude (Q_{40N} in Figure 12a and 12b). The performance of DA lies between DAF and CFL-Match, reflecting the effects of our three proposed techniques as in Section 7.1.

A.2 Comparing with Other Algorithms

We compare DAF against existing algorithms other than CFL-Match, i.e., VF2 [9], QuickSI [35], GraphQL [17], GADDI [46], SPath [47], Turbo$_{iso}$ [16]. Since the implementations of the existing algorithms (except CFL-Match) are based on Windows, we performed experiments using a machine with an Intel i7-4790K 4.00GHz CPU and 8GB memory running Windows 8.1 Pro. Figure 13 shows the results. (We were not able to test YAGO due to small memory on our Windows machine. Also GADDI did not run on Email.) We can see that DAF is always the best and Turbo$_{iso}$ is the runner-up.



(a) Yeast (b) Human
(c) HPRD (d) Email
(e) DBLP

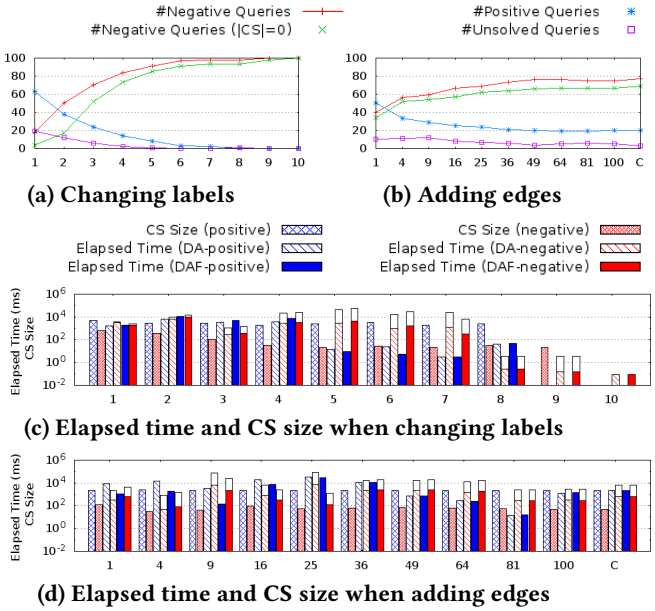
Figure 13: Elapsed time of DAF and existing algorithms (except CFL-Match).

A.3 Negative Queries

To study the behavior on negative queries, i.e., query graphs having no embeddings, we generate query sets that may have negative queries by two methods: 1) randomly change vertex labels of each query graph in Q_{20N} of Human and 2) add random edges to each query graph in Q_{20N} of Human.

Figure 14a (resp. 14b) shows the numbers of positive/negative/unsolved queries in Q_{20N} by DAF when we change 1 to 10 labels (resp. add 1 to 100 edges; C means that queries are complete graphs). For the positive queries, all 10^5 embeddings are found in 10 min. Among the negative queries, those whose CS size is 0 are also depicted in Figure 14a and 14b, and in this case DAF's preprocessing immediately ensures the *negativity* and so there is no backtracking, i.e., search time is 0. For a negative query whose CS size is not 0, DAF explores the entire search space. A common phenomenon as we change labels and add edges is that the number of negative queries whose CS size is 0 is high among negative queries, which shows that our DAG-graph DP is powerful in dealing with negative queries.

Figure 14c (resp. 14d) shows the average elapsed time and CS size of DAF for positive queries and negative queries in Figure 14a (resp. 14b). The elapsed time of DA is also shown, which is not much different from that of DAF. In general, the elapsed time of positive queries is smaller than the elapsed time of negative queries whose CS size is not 0 (because DAF explores the entire search space), but it is larger than the elapsed time of all negative queries (because there are many negative queries whose CS size is 0). When we change labels, the number of negative queries whose CS size is 0 increases



(a) Changing labels (b) Adding edges

(c) Elapsed time and CS size when changing labels

(d) Elapsed time and CS size when adding edges

Figure 14: Behavior of DAF on negative queries. In (c) and (d), the higher bar (resp. lower bar) in elapsed time for negative queries represents elapsed time of negative queries whose CS size is not 0 (resp. elapsed time of all negative queries).

rapidly, and thus the elapsed time of negative queries falls down to almost zero. When we add edges, however, the number of negative queries whose CS size is 0 saturates (even until queries become complete graphs), and the elapsed time remains more or less the same.

A.4 Parallelizing DAF

Alhough DAF is basically memory-based and optimized on a single computer, it can still benefit from parallelism by means of multithreading on multiple processing units with a shared memory. A simple method is to load the CS structure on a shared memory and fork threads at the beginning of backtracking. Each thread is allocated a set of candidates which the root of the query DAG is mapped to, and it simultaneously performs backtracking from those candidates. Our implementation uses the loop parallelism of OpenMP; the iterations at line 4 of Algorithm 2 are run by multiple threads concurrently. Whenever a thread discovers a new embedding, it atomically updates the count value (i.e., number of embeddings) which is shared globally, and continues, or it terminates if the global count value exceeds the threshold (i.e., k).

We first test on Human in the same setting as our main experiment except for varying the number of threads $p \in \{1, 2, 4, 8, 16\}$, i.e., up to 16 cores. Figure 15 shows the results. In general, the elapsed time decreases as the number of threads increases. In particular, when the number of threads

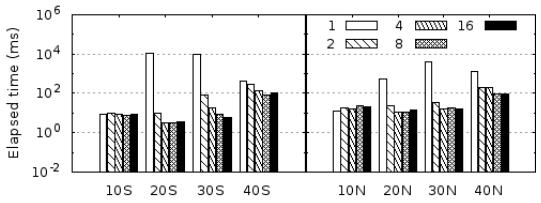


Figure 15: Elapsed time for multiple threads (1, 2, 4, 8, 16) in finding 10^5 embeddings on Human.

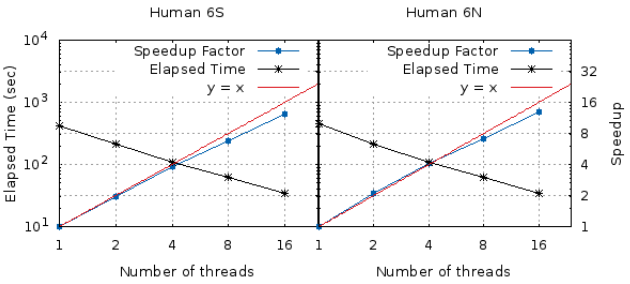


Figure 16: Speedup of multiple threads in finding all embeddings on Human.

changes from 1 to 2, the improvement in the elapsed time is dramatic (i.e., hundreds times faster). Since we find a moderate number (e.g., $k = 10^5$) of embeddings, an excessively bad behavior of an iteration run by a thread can be offset by a random iteration run by another thread.

To understand the speedup of the parallelized DAF, we measure the elapsed time to find *all embeddings* in the data graph (i.e., $k = \infty$) for queries of size 6 in Human (i.e., the total amount of work is the same for any number of threads). Figure 16 shows the speedup for $p \in \{1, 2, 4, 8, 16\}$, where the speedup is the ratio of the average elapsed time by one thread to that by p threads. In general it shows a good scalability, e.g., the speedup is 12.7 for non-sparse queries when $p = 16$.

A.5 Evaluating Boosting Technique in [33]

We applied the boosting technique in [33] to DAF. 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 [33]) may miss some embeddings. Thus, our boosted version of DAF, called DAF-Boost, considers only the equivalence relationships. The results are shown in Figure 17. The effect of the boosting technique varies with data graphs. For Human, DAF-Boost shows a considerable improvement over DAF. For Email, DAF and DAF-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, YAGO, Email, Yeast, DBLP, and HPRD are

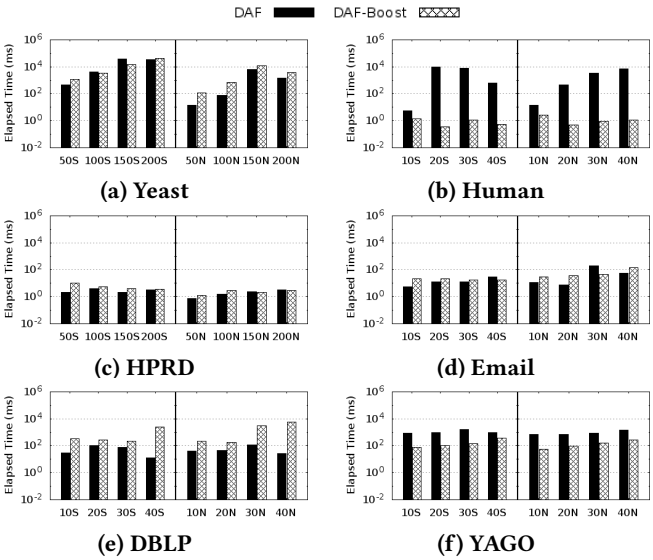


Figure 17: Elapsed time of DAF and DAF-Boost.

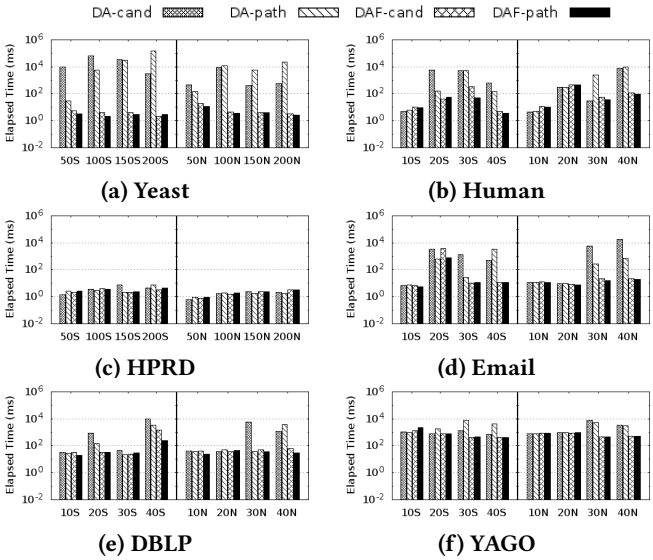


Figure 18: Elapsed time of DA-cand, DA-path, DAF-cand, and DAF-path.

about 53.1%, 41.4%, 16.4%, 5.1%, 2.1%, and 1.4%, respectively. As expected, the boosting technique can be highly effective when the data graph has many similarities in itself.

A.6 Choosing Techniques for DAF

Since data graphs have very different properties, we analyze the effectiveness of adaptive matching orders (i.e., candidate-size order and path-size order) and pruning by failing sets, in order to determine which techniques to use. Figure 18 shows the results. In most cases, the technique of pruning by failing sets improves performance, i.e., the benefit of pruning by failing sets outweighs the overhead of computing failing sets. Hence, we always choose to use pruning by failing sets. The gap between DAF-cand and DAF-path is marginal, but DAF-path is slightly better. Therefore, we choose DAF-path as the final version of our algorithm, called DAF. (To be consistent with DAF, we choose DA-path as DA.)

A.7 Proof of Lemma 4.1

Proof. For each candidate $v \in C(u)$, we look at all edges (v, v_c) such that $v_c \in C'(u_c)$ (i.e., all downward edges from v in Figure 4) by Recurrence (1). Hence the overall amount of work we perform in DAG-graph DP is proportional to the number of edges in the CS. Note that for two adjacent vertices $u, u' \in V(q)$, there are $O(|E(G)|)$ edges between the candidates in $C(u)$ and $C(u')$. Therefore, the number of edges in the CS (and the time complexity of DAG-graph DP) is bounded by $O(|E(q)| \times |E(G)|)$. □

A.8 Correctness of Failing Set Computation

Lemma A.1. *The set F_M computed in Section 6.1 is a failing set of search tree node M.*

Proof. It is easy to see that the ancestor-closure property is always satisfied for F_M. Hence it suffices to prove that the set F_M computed in Section 6.1 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, there exist no embeddings of $q[F_{M_i}]$ which are extensions of $M_i[F_{M_i}]$. 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, there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M]$.

Case 2.2: We will prove that there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M]$ by testing all extensions of $M[F_M]$ to u_n. Since $u_n \in F_M$ and F_M is ancestor-closed, $M[F_M]$ maps all the parents of u_n, and the mappings of the parents of u_n in $M[F_M]$ are the same as those 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 because of $F_{M_i} \subseteq F_M$. Since there exist no embeddings of $q[F_{M_i}]$ which are extensions of $M_i[F_{M_i}]$ by induction hypothesis, there exist no embeddings of $q[F_{M_i}]$ which are extensions of $M[F_M] \cup \{(u_n, v_i)\}$, and so there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M] \cup \{(u_n, v_i)\}$. Since none of the extensions of $M[F_M]$ to u_n leads to an embedding of $q[F_M]$, there exist no embeddings of $q[F_M]$ which are extensions of $M[F_M]$. □