GuP: Fast Subgraph Matching by Guard-based Pruning

JUNYA ARAI, Nippon Telegraph and Telephone Corporation, Japan
YASUHIRO FUJIWARA, Nippon Telegraph and Telephone Corporation, Japan
MAKOTO ONIZUKA, Osaka University, Japan

Subgraph matching, which finds subgraphs isomorphic to a query, is crucial for information retrieval from data represented as a graph. To avoid redundant explorations in the data, existing methods restrict the search space by extracting candidate vertices and edges that may constitute isomorphic subgraphs. However, expensive computation is still required because candidate vertices induce many non-isomorphic subgraphs. In this paper, we propose GuP, a subgraph matching algorithm with pruning based on guards. A guard represents a pattern of intermediate search states that never lead to isomorphic subgraphs. By attaching a guard to each candidate vertex and edge, GuP adaptively filters out unnecessary candidates depending on the search state at each step. The experimental results show that GuP effectively reduces the search space and can answer difficult queries that state-of-the-art methods cannot answer within a practical time frame.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Information systems** → *Information retrieval query processing*; Graph-based database models.

Additional Key Words and Phrases: subgraph isomorphism, graph query, backtracking, nogood

ACM Reference Format:
Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proc. ACM Manag. Data* 1, 2, Article 167 (June 2023), 26 pages. https://doi.org/10.1145/3589312

1 INTRODUCTION

Similar to searching for a specific phrase within a document, searching for a specific structure within a graph is one of the most fundamental operations in graph databases. This operation is formally defined as *subgraph matching*. It enumerates all *full embeddings*, which map every vertex in a *query graph* to the corresponding vertex of an isomorphic subgraph in a *data graph*. We refer to a vertex in the query graph and the data graph as a *query vertex* and a *data vertex*, respectively. Fig. 1 shows an example of query graph Q and data graph G. Letting (u_i, v_j) denote an assignment of query vertex u_i to data vertex v_j, there exists full embedding $M = \{(u_0, v_1), (u_1, v_4), (u_2, v_7), (u_3, v_{10}), (u_4, v_0)\}$. Since subgraph matching is NP-hard [15] and computationally expensive for complex graphs, efficient methods have long been studied [3, 8, 14, 15, 20, 35–38].

Mainstream methods for subgraph matching perform a *backtracking* search. Given *partial embedding* M, a backtracking procedure extends M with a new vertex assignment and recurses with the extended M until M becomes a full embedding. M is extended so that it preserves isomorphism. If such an extension is impossible, the procedure returns to the caller, and the caller tries extending M with another assignment. A partial embedding is called a *deadend* if M is not extendable or no full

Authors' addresses: Junya Arai, junya.arai@ntt.com, Nippon Telegraph and Telephone Corporation, 3-9-11 Midoricho, Musashino-shi, Tokyo, Japan, 180-8585; Yasuhiro Fujiwara, yasuhiro.fujiwara@ntt.com, Nippon Telegraph and Telephone Corporation, 3-1 Morinosato Wakamiya, Atsugi-shi, Kanagawa, Japan, 243-0198; Makoto Onizuka, onizuka@ist.osaka-u.ac.jp, Osaka University, 1-5 Yamadaoka, Suita-shi, Osaka, Japan, 565-0871.

2836-6573/2023/6-ART167 $15.00
https://doi.org/10.1145/3589312

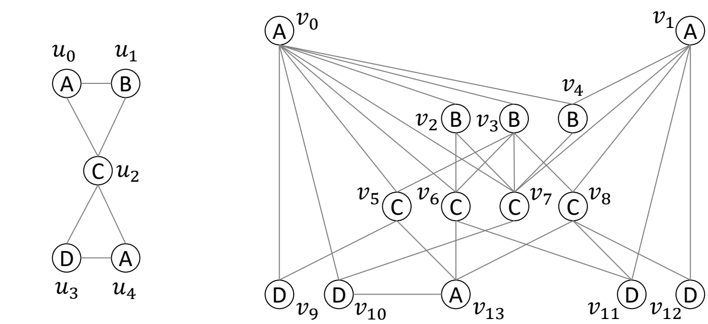


Fig. 1. Example of a query graph and a data graph

embeddings are yielded in the subsequent recursions [32]. Since it is futile to perform recursions with deadend partial embeddings, reducing such recursions is the key to improving the search performance.

To this end, most methods use *candidate filtering* [3]. For each query vertex u_i, candidate filtering collects data vertices that can be a destination of u_i into $C(u_i)$, a set of the *candidate vertices* of u_i. One of the most primitive filters is based on labels; it makes $C(u_i)$ of the data vertices with the same label as u_i. To further remove unnecessary vertices, modern methods perform matching with a tree or a directed acyclic graph (DAG) obtained from a query graph [2, 3, 14, 15, 20]. They manage the candidate vertices and the edges between them, which we call *candidate edges*, in an auxiliary data structure such as a candidate space [14]. These approaches significantly speed up the search.

However, even if candidate filtering is applied, backtracking still suffers from numerous futile recursions. This is because candidate filtering hardly captures conflicts between assignments, that is, a constraint violation caused by a combination of multiple assignments. Subgraph isomorphism requires that adjacent query vertices u_i and u_j are assigned to adjacent data vertices, constraining the combination of assignments of u_i and u_j. This implies that a cycle in a query graph must be mapped to a cycle in a data graph. Cycles are usually difficult to find because of the sparseness of real-world graphs [7], so partial embeddings tend to become deadends. Moreover, embeddings must be injective; specifically, each query vertex must be assigned to a different data vertex. This globally constrains the combination of assignments in a partial embedding. These constraints are not well captured in the extraction of $C(u_i)$ because it is based on the constraints on only u_i, without assumptions on the assignments of the other query vertices. Thus, candidate filtering fails to eliminate deadends due to conflicting assignments.

Our approach. In this paper, we propose *GuP*, an efficient algorithm for subgraph matching. In contrast to candidate filtering, which captures constraints on a single vertex, GuP utilizes a *guard* to capture constraints on a partial embedding. A guard is attached to each candidate vertex and candidate edge. If a partial embedding matches the attached guard, GuP adaptively filters out that vertex or edge. This enables early pruning of deadend partial embeddings before a violation of the constraints is detected.

For effective pruning, GuP combines two types of guards: a *reservation guard* and a *nogood guard*. A reservation guard propagates the injectivity constraint for checking it in earlier backtracking steps. Here we abuse the notation for an assignment, (u_i, v), to denote candidate vertex v of query vertex u_i. Intuitively, a reservation guard attached to candidate vertex (u_i, v) is a set of the data vertices to be used in future extensions of partial embeddings containing assignment (u_i, v). The data vertices

Table 1. Notations

Symbol	Definition	Symbol	Definition
u_i, v	Query vertex and data vertex	$M \oplus v$	Extension of M with v: $M \cup \{(u_{\lvert M \rvert}, v)\}$
$N(v)$	Neighbor set of a vertex	$\mathrm{dom}(M)$	Domain of a mapping: $\{u_i \mid \exists v, (u_i, v) \in M\}$
$N_{-}(u_i)$	Backward neighbor set: $\{u_j \in N(u_i) \mid j < i\}$	$\mathrm{Im}(M)$	Image of a mapping: $\{v \mid \exists u_i, (u_i, v) \in M\}$
$N_{+}(u_i)$	Forward neighbor set: $\{u_j \in N(u_i) \mid j > i\}$	$C(u_i; M)$	Local candidate-vertex set of u_i under M
M, D	Embedding and nogood: $M, D \subseteq V_Q \times V_G$	$B(u_i; M)$	Bounding set of u_i under M
$V_Q[:i]$	Set of query vertices before u_i: $\{u_j \mid j < i\}$	R	Reservation guard on a candidate vertex
$M[K]$	Restriction of M to K: $\{(u_i, v) \in M \mid u_i \in K\}$	NV	Nogood guard on a candidate vertex
$M[:i]$	Restriction of M to $V_Q[:i]$	NE	Nogood guard on a candidate edge

in the reservation guard must be kept unassigned in a partial embedding before extending it with (u_i, v); otherwise, it violates the injectivity constraint in the subsequent extensions. Hence, we can filter v out from $C(u_i)$ in such cases.

On the other hand, the nogood guards detect deadends by learning inconsistent assignments from the deadends encountered before. GuP attaches a nogood guard to both candidate vertex and candidate edge to exploit the adjacencies in a query graph for pruning. A nogood guard is a set of assignments that never co-occur in a full embedding along with the candidate vertex or edge where that nogood guard is attached. Thus, we can filter out that vertex or edge if a partial embedding includes the assignments in the nogood guard. GuP generates nogood guards on-the-fly during backtracking by discovering a set of inconsistent assignments called a *nogood* [34]. Although a nogood is a known concept in the field of constraint programming, few studies [26, 27] have adapted it to subgraph matching. In this study, we introduce novel nogood discovery rules and *search-node encoding* of a nogood guard. Our nogood discovery rules yield a general nogood that can be found in many partial embeddings, leading to high pruning power. GuP also has a special rule for a nogood guard on an edge, while conventional rules cannot yield a nogood that fits filtering of candidate edges. Furthermore, search-node encoding provides a compact representation for a nogood guard and enables pruning without increasing the time and space complexities.

GuP stores guards in a *guarded candidate space* (GCS), an auxiliary data structure that extends a candidate space with guards. The experimental results show that guards significantly reduce futile recursions, and as a result, GuP can process query graphs that cannot be processed by the state-of-the-art methods even after spending an hour. Our contributions introduced in this paper are summarized as follows:

(1) pruning approach based on guards,
(2) reservation, a concept for pruning based on propagated injectivity,
(3) nogood discovery rules to obtain general nogoods, and
(4) search-node encoding of a nogood guard.

Paper organization. We present the background and details of our approach in Sections 2 and 3, respectively. Section 4 discusses the experimental results, and we conclude this paper in Section 5.

2 BACKGROUND

This section introduces definitions used in this paper and then reviews related work.

2.1 Definitions

Table 1 lists the symbols used in this paper. We focus on vertex-labeled simple undirected graphs, similarly to previous studies [3, 8, 14, 15, 20, 35–38]. Note that our method can easily adapt to other kinds of graphs, such as directed graphs and edge-labeled graphs. Consider query graph

$Q = (V_Q, E_Q, \Sigma, \ell)$ and data graph $G = (V_G, E_G, \Sigma, \ell)$. V_Q and V_G are sets of vertices, E_Q and E_G are sets of edges, Σ is a set of labels, and ℓ is a mapping of a vertex to its label. We assume that the query vertices have consecutive ID numbers, i.e., $V_Q = \{u_0, u_1, u_2, \ldots\}$. If there is no ambiguity, we use u_i as a query vertex and v as a data vertex without explicitly mentioning it. $N(u_i)$ and $N(v)$ denote the sets of neighbors of u_i and v, respectively. Let the set of forward neighbors $N_+(u_i) = \{u_j \mid j > i\}$ and the set of backward neighbors $N_-(u_i) = \{u_j \mid j < i\}$. For arbitrary domain $X \subseteq V_Q$, mapping $M : X \longrightarrow V_G$ is denoted by binary relation $M \subseteq X \times V_G$. $\mathrm{dom}(M)$ and $\mathrm{Im}(M)$ denote the domain and the image of M, respectively. Notation $M(u_i)$ implicitly assumes $u_i \in \mathrm{dom}(M)$. Given query graph Q and data graph G, an embedding is defined as follows.

Definition 2.1 (Embedding). Mapping $M : V_Q \longrightarrow V_G$ is an *embedding* of Q into G if and only if M satisfies the following constraints:

(1) *Label constraint*: $\forall u_i \in V_Q,\ \ell(u_i) = \ell(M(u_i))$,
(2) *Adjacency constraint*: $\forall (u_i, u_j) \in E_Q,\ (M(u_i), M(u_j)) \in E_G$,
(3) *Injectivity constraint*: $\forall u_i, u_j \in V_Q,\ i \neq j \Rightarrow M(u_i) \neq M(u_j)$.

Then, the problem definition in this paper is given as follows.

Definition 2.2 (Subgraph matching). Given query graph Q and data graph G, *subgraph matching* is a problem of enumerating all the embeddings of Q in G.

An embedding is also called a full embedding in contrast with a partial embedding, which is an embedding of an induced subgraph of Q. The *length* of partial embedding M is the number of assignments in M, denoted by $|M|$. An *extension* is a mapping made by adding assignments to a partial embedding. Unlike a partial embedding, an extension may not satisfy the constraints of isomorphism in Definition 2.1.

In the following discussion, we assume that the matching order is ascending order of query vertex IDs. This preserves the generality because renumbering vertex IDs can change the matching order. Additionally, we assume that vertex IDs are numbered in a connected order [36], that is, every query vertex except u_0 has a neighbor with a smaller vertex ID. Note that matching orders used in subgraph matching usually hold this property [25, 35]. Under our assumptions, partial embeddings and extensions of length k always consist of assignments of $u_0, u_1, \ldots, u_{k-1}$. We denote an extension of partial embedding M with an assignment to v by $M \oplus v$ i.e., $M \oplus v = M \cup \{(u_k, v)\}$ where k is the length of M.

2.2 Related Work

There are various problem settings and algorithms related to the search of subgraphs, such as subgraph enumeration algorithms for unlabeled graphs [21, 23, 24] and RDF query engines [17, 22, 39] for edge-labeled graphs. On vertex-labeled graphs, subgraph containment algorithms [4, 6, 13] take a set of data graphs and find ones with at least one embedding of a query graph, and subgraph matching algorithms find all embeddings in a single data graph. Approaches based on join operations [1, 28, 37] are mainly used for subgraph homomorphism-based subgraph matching, which allows duplicate assignments of query vertices to the same data vertex. In contrast, subgraph isomorphism-based subgraph matching, the focus of this paper, prohibits it. Since most algorithms for this problem setting perform a backtracking search [38], we review three approaches widely used to improve the efficiency of backtracking.

Candidate filtering. Conventional filtering methods are based on local features. Ullmann [38] used label-and-degree filtering (LDF), which collects data vertex v as a candidate vertex of u_i if v has the same label as u_i and v has the degree greater than or equal to that of u_i. Neighborhood label frequency filtering (NLF) [3] checks for every label l if a candidate vertex of u_i has label-l neighbors not fewer than those of u_i. For example, v_{13} in Fig. 1b is removed from $C(u_0)$ because v_{13} has no label-B neighbor although u_0 has one label-B neighbor, u_1. Recent methods use LDF and NLF in common, but they also perform pseudo-matching between the vicinity of a candidate vertex and that of the corresponding query vertex [16, 36, 40] or matching with a spanning tree or a DAG built from a query graph [2, 3, 14, 15, 20]. All the previous approaches extract a candidate-vertex set before backtracking and do not change it after that. In contrast, GuP adaptively changes it depending on a partial embedding by using guards.

Optimization of matching order. The size of the search space varies depending on matching order, in which the destination of query vertices is determined. This is because the destination of query vertex u_i must be chosen from data vertices adjacent to the destinations of all the matched neighbors of u_i. Many efforts have been made to generate a good matching order that first decides the destinations of query vertices with fewer candidate vertices and keeps the search space of the remaining query vertices small [3, 14–16, 33, 36]. However, we still do not have a method that can generate a good order for an arbitrary query graph and data graph [19, 35]. Thus, reducing candidates is vital for consistently achieving high performance across various inputs.

Use of nogoods. A nogood was introduced by Stallman and Sussman in 1977 [34] and has been well studied in the AI community. Pruning with nogoods is performed in two ways: *backjumping* [11, 30, 34] and *nogood recording* [12, 18, 34]. Backjumping abandons deadends by escaping from ongoing recursions until the assignment shared with the last discovered nogood is changed. On the other hand, nogood recording stores discovered nogoods in a database and prunes partial solutions including a recorded nogood. Several studies recently exploit nogoods in subgraph matching [26, 27]. Backjumping was also independently proposed in the database community. DAF [14] performs failing set-based pruning [14], and VEQ [20] captures equivalences of vertices in backjumping, like symmetricity-based nogood discovery [10]. GuP also employs backjumping, but unlike DAF and VEQ, it additionally performs pruning with nogood guards, which is categorized as a nogood recording technique. This combination enables GuP to eliminate more search space.

3 METHOD

We propose GuP, an efficient algorithm for subgraph matching. GuP prunes deadend partial embeddings by filtering out unnecessary candidate vertices and edges adaptively to the assignments in a partial embedding. The key idea is *guards* attached to each candidate vertex and edge, which represent a filtering condition. In the following sections, we first present the overview of GuP in Section 3.1. Next, Sections 3.2 and 3.3 detail a reservation guard and a nogood guard, respectively, and Section 3.4 presents the backtracking algorithm using guards. We introduce search-node encoding in Section 3.5, and finally, Section 3.6 analyzes the computational complexities of GuP. All the examples in this section consider query graph Q and data graph G shown in Fig. 1.

3.1 Overview

GuP consists of the following steps.

(1) *Guarded candidate space (GCS) construction:* GuP builds a GCS, an auxiliary data structure that organizes candidate vertices, candidate edges, and guards. This step involves candidate filtering and matching order optimization.
(2) *Reservation guard generation:* GuP populates the reservation guards in the GCS by propagating the injectivity constraint between candidate vertices.

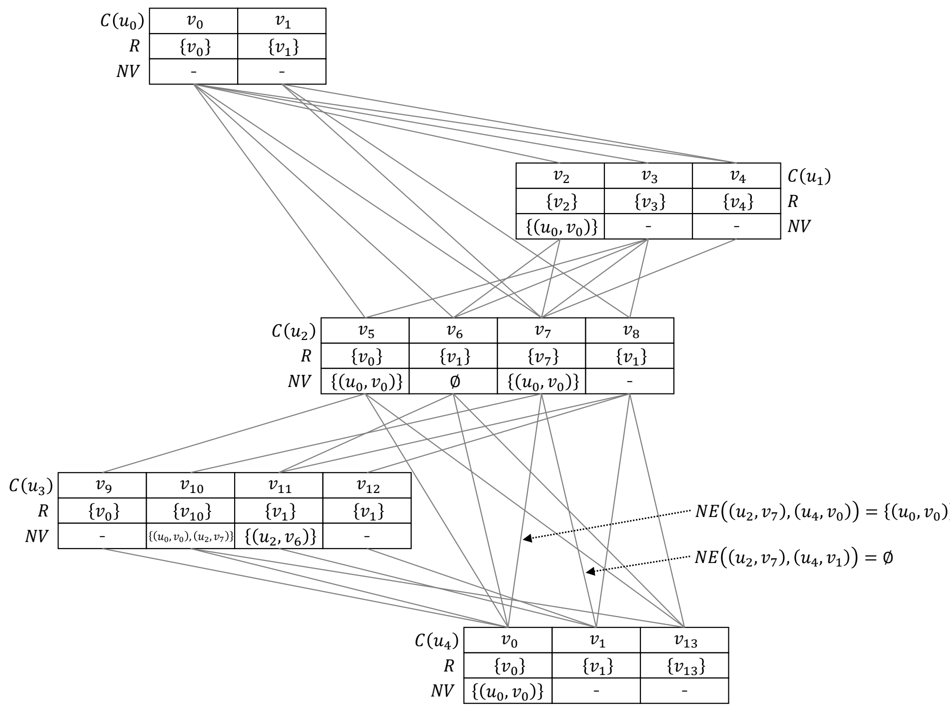


Fig. 2. Guarded candidate space for Q and G in Fig. 1.

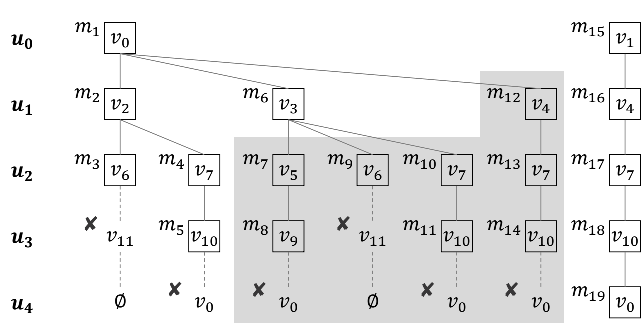


Fig. 3. Search tree for Q and G in Fig. 1.

(3) *Backtracking search:* GuP enumerates full embeddings using the information in the GCS. Nogood guards are generated on-the-fly and used for pruning along with reservation guards.

In the GCS construction step, GuP uses extended DAG-graph DP [20] for filtering candidates and a VC [36]-based approach for optimizing the matching order. In the following, we assume that the query vertices are numbered in the optimized order. Note that approaches for candidate filtering and matching order optimization are out of the scope of this work, and guard-based pruning can be combined with arbitrary existing approaches. Fig. 2 illustrates the GCS for Q and G shown in Fig. 1. All the candidate-vertex sets are simply a set of data vertices with the same label, except that v_{13} is removed from $C(u_0)$ by NLF as described in Section 2.2. R denotes a reservation guard, and *NV* and *NE* denote nogood guards on a vertex and an edge, respectively. A nogood guard is attached to every candidate edge, although not all of them are shown for conciseness. We will later describe how the guards shown in the figure are obtained. After the GCS construction, GuP generates reservation guards and then starts backtracking.

3.2 Reservation Guard

This section first introduces an abstract concept of a reservation and then presents an algorithm to generate reservation guards.

3.2.1 Reservation. We begin with the fundamental definitions. Let $Q[I]$ be a subgraph of Q induced by $I \subseteq V_Q$,

Definition 3.1 (Inclusive descendant). Let $u_i, u_j \in V_Q$. u_j is an *inclusive descendant* of u_i if and only if $u_j = u_i$ or $u_j \in N_+(u_k)$ for some u_k that is an inclusive descendant of u_i.

Definition 3.2 (Rooted subembedding). Let (u_i, v) be a candidate vertex, I be the set of all the inclusive descendants of u_i, and M be a set of assignments of $u_j \in I$. M is a *subembedding* rooted at (u_i, v) if and only if (i) M is an embedding of $Q[I]$, (ii) M contains assignment (u_i, v), and (iii) $M(u_j) \in C(u_j)$ holds for arbitrary u_j.

Condition (iii) makes sense because $Q[I]$ is a subgraph of Q, and hence its vertices may be mapped to data vertices that are not candidate vertices for Q.

Definition 3.3 (Reservation). Let (u_i, v) be a candidate vertex and $S \subseteq V_G$. S is a *reservation* of (u_i, v) if and only if an arbitrary subembedding rooted at (u_i, v) contains at least one assignment to a data vertex in S.

Example 3.4. The inclusive descendants of u_1 are u_1, u_2, u_3, and u_4. As shown in the GCS (Fig. 2), the subembeddings rooted at (u_1, v_3) are $\{(u_1, v_3), (u_2, v_5), (u_3, v_9), (u_4, v_0)\}$, $\{(u_1, v_3), (u_2, v_7), (u_3, v_{10}), (u_4, v_0)\}$, $\{(u_1, v_3), (u_2, v_8), (u_3, v_{11}), (u_4, v_1)\}$, and $\{(u_1, v_3), (u_2, v_8), (u_3, v_{12}), (u_4, v_1)\}$. All of them contain an assignment to v_0 or v_1, and thus $\{v_0, v_1\}$ is one of the reservations of (u_1, v_3).

GuP selects one of the possible reservations of (u_i, v) as reservation guard $R(u_i, v)$. The definition of a reservation implies that, if a partial embedding contains assignments to every data vertex in $R(u_i, v)$, we cannot assign inclusive descendants of u_i satisfying the injectivity constraint. We formally state it as follows. Let $M[:i]$ be $\{(u_j, v) \in M \mid j < i\}$.

Definition 3.5 (Matching with a reservation guard). We say that partial embedding M *matches* $R(u_i, v)$ if and only if $R(u_i, v) \subseteq \mathrm{Im}(M[:i])$ holds.

Lemma 3.6. *Let M be a partial embedding. Suppose that $R(u_i, v)$ is a reservation of (u_i, v) and M matches $R(u_i, v)$. Then, $M[:i] \cup \{(u_i, v)\}$ is a deadend.*

Proof. We make a proof by contradiction. Suppose that there exists full embedding $\hat{M}$ such that $R(u_i, v) \subseteq \mathrm{Im}(\hat{M}[:i])$ and $(u_i, v) \in \hat{M}$. Letting I be the set of all the inclusive descendants of u_i, $\hat{M}[I]$ is a (u_i, v)-rooted subembedding. Hence, by Definition 3.3, there exists $u_j \in I$ such that $\hat{M}(u_j) \in R(u_i, v)$. In addition, $j \geq i$ holds because u_j is an inclusive descendant of u_i. At the same time, from $R(u_i, v) \subseteq \mathrm{Im}(\hat{M}[:i])$, there exists u_k such that $k < i$ and $\hat{M}(u_k) = \hat{M}(u_j) \in R(u_i, v)$. Since $k \neq j$ holds by $k < i \leq j$, $\hat{M}$ violates the injectivity constraint, which is a contradiction. □

Letting $i = |M|$ and $v \in C(u_i)$, $M \oplus v$ is a deadend if M matches $R(u_i, v)$. We can filter out v from $C(u_i)$ by using this property.

3.2.2 Reservation Guard Generation. For effective pruning, we prefer that $R(u_i, v)$ is a reservation expected to be matched by many partial embeddings. To generate such a reservation guard, GuP employs two policies. First, GuP avoids generating a reservation guard that cannot be matched by any possible partial embedding. We utilize the following lemma to check it. Let $V_Q[:i] = \{u_j \mid j < i\}$, $C^{-1}(v) = \{u_i \mid v \in C(u_i)\}$, $C^{-1}(S) = \bigcup_{v \in S} C^{-1}(v)$, and $C^{-1}(v)[:i] = \{u_j \in C^{-1}(v) \mid j < i\}$.

LEMMA 3.7. *Let (u_i, v) be a candidate vertex and $R(u_i, v)$ be its reservation. Suppose that $R(u_i, v)$ holds (i) $\exists v' \in R(u_i, v), C^{-1}(v')[:i] = \emptyset$ or (ii) $\exists S \subseteq R(u_i, v), |S| > |C^{-1}(S)[:i]|$. Then, no partial embedding matches $R(u_i, v)$.*

PROOF. We make a proof by cases. Regarding the case that $R(u_i, v)$ holds condition (i), suppose that $v' \in R(u_i, v)$ holds $C^{-1}(v')[:i] = \emptyset$. Then, we have $v' \notin C(V_Q[:i])$. Since arbitrary partial embedding M holds $\mathrm{Im}(M[:i]) \subseteq C(V_Q[:i])$, we have $v' \notin \mathrm{Im}(M[:i])$. It follows that $R(u_i, v) \nsubseteq \mathrm{Im}(M[:i])$. Regarding the case that $R(u_i, v)$ holds condition (ii), we argue by contradiction. Suppose there exists partial embedding M such that $R(u_i, v) \subseteq \mathrm{Im}(M[:i])$. Since M is an injective mapping, we obtain have its inverse mapping M^{-1}. We have $M^{-1}(v') \in V_Q[:i]$ and $M^{-1}(v') \in C^{-1}(v')$ for all $v' \in \mathrm{Im}(M[:i])$, and thus $M^{-1}(v') \in C^{-1}(v')[:i]$ holds. Since $R(u_i, v) \subseteq \mathrm{Im}(M[:i])$ holds by hypothesis, for all $S \subseteq R(u_i, v)$, we have $M^{-1}(S) \subseteq C^{-1}(S)[:i]$, and thus $|S| \leq |C^{-1}(S)[:i]|$ holds because $|M^{-1}(S)| = |S|$. This contradicts supposition $\exists S \subseteq R(u_i, v), |S| > |C^{-1}(S)[:i]|$. Therefore, we have proved the statement. □

GuP considers a reservation to be *matchable* as a reservation guard of (u_i, v) if it satisfies neither condition (i) nor (ii) in Lemma 3.7.

Example 3.8. As shown in Example 3.4, $\{v_0, v_1\}$ is a reservation of (u_1, v_3). Let us check if this is matchable by using conditions (i) and (ii) in Lemma 3.7. Both $C^{-1}(v_0)$ and $C^{-1}(v_1)$ are $\{u_0, u_4\}$, and thus $C^{-1}(v_0)[:1] = C^{-1}(v_1)[:1] = \{u_0\} \neq \emptyset$. Hence, condition (i) is not held. However, we have $|C^{-1}(\{v_0, v_1\})[:i]| = |\{u_0\}| = 1$, which is less than $|\{v_0, v_1\}|$ (= 2). Therefore, condition (ii) is held, and $\{v_0, v_1\}$ is not matchable as a reservation guard of (u_1, v_3).

The second policy is to minimize the size of a reservation guard (i.e., $|R(u_i, v)|$) because a smaller reservation guard tends to be matched by more partial embeddings. By definition, an arbitrary candidate vertex (u_i, v) has a *trivial reservation* $\{v\}$ because any subembeddings rooted at (u_i, v) contain v. Although this is clearly the smallest choice for $R(u_i, v)$, this does not reduce futile recursions because it just performs an ordinary injectivity check that ensures v is not used in a partial embedding. To obtain a non-trivial small reservation, GuP generates candidates of reservation guards by propagating the injectivity constraint in a bottom-up manner and selects the smallest one as a reservation guard.

Definition 3.9 (Reservation guard candidate). Let (u_i, v) be a candidate vertex, $u_j \in N_+(u_i)$, and $R(u_i, v)$ be a reservation of (u_i, v). The *reservation guard candidate* of (u_i, v) regarding u_j is the smallest set of data vertices that is (i) matchable as a reservation guard of (u_i, v) and (ii) a superset of $\{v'\}$ or $R(u_j, v') \setminus \{v\}$ for all $v' \in N(v) \cap C(u_i)$. If there is no such set, the reservation guard candidate is undefined.

LEMMA 3.10. *Let (u_i, v) be a candidate vertex and $u_j \in N_+(u_i)$. Suppose that S is a reservation guard candidate of (u_i, v) regarding u_j. Then, S is a reservation of (u_i, v).*

PROOF. We argue by contradiction. Suppose that S is not a reservation of (u_i, v). It follows that there exists (u_i, v)-rooted subembedding M that does not have an assignment to a vertex in S. Since M satisfies the adjacency constraint, $M(u_j) \subseteq N(v) \cap C(u_j)$ holds. By hypothesis, we have $M(u_j) \in S$ or $R(u_j, M(u_j)) \setminus \{v\} \subseteq S$. Thus, we proceed by cases. Regarding the case that $M(u_j) \in S$

holds, M has an assignment to a vertex in S, which is a contradiction. Regarding the case that $R(u_j, M(u_j)) \setminus \{v\} \subseteq S$ holds, let I' be the inclusive descendant set of u_j. $M[I']$ has an assignment to a vertex in $R(u_j, M(u_j))$ because $M[I']$ is a subembedding rooted at $(u_j, M(u_j))$. In addition, since M maps u_i to v and satisfies the injectivity constraint, $M[I']$ does not have an assignment to v. Hence, $M[I']$ has an assignment to a vertex in $R(u_j, M(u_j)) \setminus \{v\} \subseteq S$, which is a contradiction. Therefore, S is a reservation of (u_i, v). □

We can obtain reservation guard candidates via solving the vertex cover problem.

Lemma 3.11. *Let (u_i, v) be a candidate vertex, $u_j \in N_+(u_i)$, and*

$$E_R = \{(v', w) \mid v' \in N(v) \cap C(u_j), w \in R(u_j, v') \setminus \{v\}\}. \tag{1}$$

Consider graph $G_R = (V_R, E_R)$ where V_R is a set of the data vertices that appear in E_R. In addition, suppose that $S \subseteq V_G$ holds that (i) S is matchable as a reservation guard of (u_i, v) and (ii) S is the minimum vertex cover of G_R. Then, S is the reservation guard candidate of (u_i, v) regarding u_j.

Proof. Let $v' \in N(v) \cap C(u_j)$ and $w \in R(u_j, v') \setminus \{v\}$. Since S is a vertex cover, S contains either or both of v' and w. If $v' \notin S$, by the definition of E_R, we have $w \in S$ for all $w \in R(u_j, v') \setminus \{v\}$. Hence, S is a superset of $\{v'\}$ or $R(u_j, v') \setminus \{v\}$ for all $v' \in N(v) \cap C(u_j)$. Since S is the smallest set that is matchable and covers G_R, it is the reservation guard candidate of (u_i, v) regarding u_j. □

Furthermore, we introduce user-defined parameter r to limit the size of a reservation guard because even the smallest reservation guard candidate can be too large for effective pruning. If a candidate vertex has reservation guard candidates whose sizes do not exceed r, the smallest one is chosen for its reservation guard. Otherwise, GuP resorts to a trivial reservation. Specifically, a reservation guard is defined as follows.

Definition 3.12 (Reservation guard). Let (u_i, v) be a candidate vertex and r be a user-defined parameter. The reservation guard on (u_i, v), denoted by $R(u_i, v)$, is defined as follows.

(1) $R(u_i, v) = \{v\}$ if, for all $u_j \in N_+(u_i)$, the reservation guard candidate of (u_i, v) regarding u_j is undefined or larger than r in size.
(2) Otherwise, $R(u_i, v)$ is the smallest reservation guard candidate of (u_i, v) regarding u_j where $u_j \in N_+(u_i)$.

Lemma 3.13. *For all candidate vertex (u_i, v), $R(u_i, v)$ given by Definition 3.12 is a reservation of (u_i, v).*

Proof. By Definition 3.12, $R(u_i, v)$ is either trivial reservation $\{v\}$ or a reservation guard candidate of (u_i, v) regarding some $u_j \in N_+(u_i)$, both of which are reservations of (u_i, v). □

Example 3.14. Fig. 2 shows reservation guard R of each candidate vertex. For example, $R(u_2, v_5)$ is obtained from the reservation guards of forward-adjacent candidate vertices, $R(v_3, v_9)$, $R(u_4, v_0)$, and $R(u_4, v_{13})$, as follows. $R(u_4, v_0)$ and $R(u_4, v_{13})$ is given by the trivial reservations, $\{v_0\}$ and $\{v_{13}\}$, respectively. Since u_3 has only forward neighbor u_4, $R(u_3, v_9)$ is given by the reservation guard candidate regarding u_4. It is the smallest matchable superset of $\{v_0\}$ or $R(u_4, v_0) \setminus \{v_9\}$ $(= \{v_0\})$, and thus $R(u_3, v_9) = \{v_0\}$. Next, we consider the reservation guard candidates of (u_2, v_5) regarding u_3 and u_4. That regarding u_3 is $R(u_3, v_9) \setminus \{v_5\}$ $(= \{v_0\})$ because $\{v_9\}$ is not matchable. Regarding u_4, a matchable superset of both $\{v_0\}$ and $\{v_{13}\}$ does not exist because of condition (i) in Lemma 3.7 $(C^{-1}(v_{13})[:2] = \emptyset)$. Hence, the reservation guard of (u_2, v_5) is given by the candidate regarding u_3, which is $\{v_0\}$.

Algorithm 1 Reservation guard generation

Input: Candidate-vertex set C and reservation size limit r
Output: Reservation guard $R(u_i, v)$ for all $u_i \in V_Q$ and $v \in C(u_i)$
1: **for** $u_i \in V_Q$ in reverse order and $v \in C(u_i)$ **do**
2: $\quad R(u_i, v) \leftarrow \{v\}$
3: $\quad$ **for** $u_j \in N_+(u_i)$ **do**
4: $\quad\quad$ Construct graph G_R from edge set E_R (Eq. (1))
5: $\quad\quad$ Find vertex cover S over G_R s.t. $|S| \leq r$ and S is matchable
6: $\quad\quad$ **if** such S is found, and $R(u_i, v) = \{v\}$ or $|R(u_i, v)| > |S|$ **then**
7: $\quad\quad\quad R(u_i, v) \leftarrow S$

Algorithm 1 shows the pseudocode of the reservation guard generation. This algorithm computes $R(u_i, v)$ in the descending order of u_i so that the reservation guards of all the forward-adjacent candidate vertices are computed before $R(u_i, v)$. The loop at line 3 computes S, the reservation guard candidate of (u_i, v) regarding $u_j \in N_+(u_i)$. Since the vertex cover problem is NP-hard, GuP employs the 2-approximate algorithm [9], which iteratively chooses an edge in E_R and adds its endpoints to S until all the vertices are covered. To find matchable S whose size does not exceed r, our algorithm ignores an edge that makes S unmatchable by checking the conditions in Lemma 3.7 and stops the iteration if $|S|$ exceeds r. If such S is found for at least one of $u_j \in N_+(u_i)$, the smallest one is chosen for $R(u_j, v)$. Otherwise, $R(u_j, v)$ is set to trivial reservation $\{v\}$. We analyze the complexity of this algorithm in Section 3.6.

3.3 Nogood Guard

This section first defines a nogood guard and then details the nogood discovery rules for nogood guards on vertices and edges, respectively.

3.3.1 Definitions. A nogood guard is a set of assignments that composes a nogood with a candidate vertex or the endpoints of candidate edges where it is attached.

Definition 3.15 (Nogood). Set of assignments $D \subseteq V_Q \times V_G$ is a *nogood* if and only if there is no full embedding $\hat{M}$ such that $D \subseteq \hat{M}$.

For example, $\{(u_0, v_0), (u_4, v_0)\}$ is a nogood because any partial embedding including these assignments violates the injectivity constraint.

Definition 3.16 (Nogood guard on a vertex). Let (u_i, v) be a candidate vertex. A nogood guard on (u_i, v), denoted by $NV(u_i, v)$, is a subset of $V_Q[:i] \times V_G$ such that $NV(u_i, v) \cup \{(u_i, v)\}$ is a nogood.

Definition 3.17 (Nogood guard on an edge). Let $((u_i, v), (u_j, v'))$ be a candidate edge. A nogood guard on $((u_i, v), (u_j, v'))$, denoted by $NE((u_i, v), (u_j, v'))$, is a subset of $V_Q[:i] \times V_G$ such that $NE((u_i, v), (u_j, v')) \cup \{(u_i, v), (u_j, v')\}$ is a nogood.

Definition 3.18 (Matching with a nogood guard). We say that partial embedding M *matches* $NV(u_i, v)$ or $NE((u_i, v), (u_j, v'))$ if and only if M is a superset of them.

A nogood is useful for pruning because a partial embedding including a nogood never yields any full embedding. Suppose that M is a partial embedding of length i. We can filter out candidate vertex (u_i, v) if M matches $NV(u_i, v)$ because $M \oplus v$ is a superset of $NV(u_i, v) \cup \{(u_i, v)\}$. Similarly, we can filter out candidate edge $((u_i, v), (u_j, v'))$ if M matches $NE((u_i, v), (u_j, v'))$. Filtering out the edge can be rephrased as prohibiting mapping query edge (u_i, u_j) to data edge (v, v') with two assignments (u_i, v) and (u_j, v'). Such mapping makes a deadend because $M \cup \{(u_i, v), (u_j, v')\}$ is a superset of $NE((u_i, v), (u_j, v')) \cup \{(u_i, v), (u_j, v')\}$, which is a nogood.

The actual value of a nogood guard is determined procedurally during backtracking. To generate a nogood guard, we need to discover a nogood from deadend partial embeddings encountered during backtracking. The following sections describe how to do it.

3.3.2 Nogood Guard on a Vertex. If a partial embedding is a deadend, the set of all its assignments is a nogood by the definition. However, such nogoods are ineffective for pruning because the same partial embedding does not appear again during the search, so no partial embedding matches it. For higher effectiveness, a smaller nogood is preferred because it is expected to be a subset of more partial embeddings. Thus, we need to carefully drop assignments irrelevant to the inconsistencies in a deadend. To capture the adjacency constraint, we introduce a set of local candidate vertices, which satisfy the adjacency constraints with the assignments in a partial embedding.

Definition 3.19 (Local candidate-vertex set). Let M be a partial embedding. The *local candidate-vertex set* of u_i under M, denoted by $C(u_i; M)$, is a set of $v \in C(u_i)$ that holds, for all $u_j \in N_-(u_i)$, (i) $v \in N(M(u_j))$ and (ii) M does not match $NE((u_j, M(u_j)), (u_i, v))$.

Condition (ii) reflects edge filtering; v cannot satisfy the adjacency constraint because edge $((u_j, M(u_j)), (u_i, v))$ is ignored if its nogood guard is matched by M. The search space is confined by local candidate-vertex sets because M is extended only with a local candidate vertex to satisfy the adjacency constraint. Hence, the assignments involved in the computation of a local candidate-vertex set are relevant to generating a deadend. Conversely, the assignments that do not influence a local candidate-vertex set are irrelevant to a deadend, so we can drop them to reduce the size of nogood. On the basis of this idea, we define a bounding set, which is a set of query vertices whose assignments determine the local candidate-vertex sets.

Definition 3.20 (Bounding set). Let M be a partial embedding and $u_i \in V_Q$. The *bounding set of u_i under M*, denoted by $B(u_i; M)$, is the union of B_{adj} and B_{guard}. B_{adj} is the set of $u_j \in N_-(u_i)$ such that $C(u_i; M[:j]) \neq C(u_i; M[:j+1])$. B_{guard} is the union of $\text{dom}(NE((u_j, M(u_j)), (u_i, v)))$ for all combinations of $u_j \in N_-(u_i)$ and $v \in C(u_i; M[:j])$ such that M matches $NE((u_j, M(u_j)), (u_i, v))$.

B_{adj} is a set of query vertices whose assignments reduce the size of the local candidate-vertex set by its adjacency relation regardless of guards. It is examined by condition $C(u_i; M[:j]) \neq C(u_i; M[:j+1])$, which says that u_j is included in the bounding set if adding assignment $(u_j, M(u_j))$ changes the local candidate-vertex set of u_i. On the other hand, B_{guard} is a set of query vertices whose assignments are committed in the matches between M and nogood guards. They are also involved in the decision of the local candidate-vertex set because they are necessary to match M with the nogood guards and filter out some edges.

Example 3.21. Let $M = \{(u_0, v_0), (u_1, v_3)\}$ and assume that M does not match any nogood guard on edges. Consider the bounding set of u_2 under M. We have $N_-(u_2) = \{u_0, u_1\}$. B_{adj} contains u_0 because $C(u_2; M[:0]) = \{v_5, v_6, v_7, v_8\}$, which differs from $C(u_2; M[:1]) = C(u_2; \{(u_0, v_0)\}) = \{v_5, v_6, v_7\}$. On the other hand, B_{adj} does not contain u_1 because $N(M(u_1))$ is a superset of $C(u_2; M[:1])$ and thus $C(u_2; M[:2]) = C(u_2; M[:1])$. In addition, $B_{\text{guard}} = \emptyset$ holds by assumption. Therefore, we have $B(u_2; M) = \{u_0\}$.

Lemma 3.22. *Let M and M' be partial embeddings and $u_i \in V_Q$. If $M[B(u_i; M)] \subseteq M'$, we have $C(u_i; M) \supseteq C(u_i; M')$.*

Proof. We make a proof by contradiction. Suppose that there exists $v \in C(u_i)$ such that $v \in C(u_i; M')$ and $v \notin C(u_i; M)$. From $v \notin C(u_i; M)$, there exists $u_j \in N_-(u_i) \cap \text{dom}(M)$ that holds either or both of (i) $v \notin N(M(u_j))$ and that (ii) $NE((u_j, M(u_j)), (u_i, v))$ is matched by M. Regarding case (i), we have $u_j \in B(u_i; M)$ by Definition 3.20 because $C(u_i; M[:j]) \neq C(u_i; M[:j+1])$. Since

$M[B(u_i;M)] \subseteq M'$ by hypothesis, $M(u_j) = M'(u_j)$ holds. However, from $v \in C(u_i;M')$, we have $v \in N(M'(u_j)) = N(M(u_j))$, which is a contradiction. Regarding case (ii), by Definition 3.20, we have $\mathrm{dom}(NE((u_j, M(u_j)), (u_i, v))) \subseteq B(u_i;M)$. By hypothesis, $M[B(u_i;M)] \subseteq M'$ holds, and thus we have $NE((u_j, M(u_j)), (u_i, v)) \subseteq M'[:j]$. However, from $v \in C(u_i;M')$, $NE((u_j, M(u_j)), (u_i, v))$ is not matched by M', which is a contradiction. Therefore, we have $C(u_i;M) \supseteq C(u_i;M')$. □

In backtracking, each local candidate vertex is further checked for conflicts in a partial embedding.

Definition 3.23 (Conflict). Let M be a partial embedding, k be the length of M, and $v \in C(u_k;M)$. Extension $M \oplus v$ has a *conflict* if any of the following conditions hold.

(1) *Injectivity conflict*: M has an assignment to v (i.e., $v \in \mathrm{Im}(M)$).
(2) *Reservation-guard conflict*: M matches $R(u_k, v)$.
(3) *Nogood-guard conflict*: M matches $NV(u_k, v)$.
(4) *No-candidate conflict*: There exists u_i ($i > k$) that has no local candidate vertices under $M \oplus v$ (i.e., $C(u_i;M \oplus v) = \emptyset$).

We can discover a nogood from an extension if it has conflicts. To indicate assignments that constitute a nogood, we introduce *mask* $K \subseteq V_Q$ that gives a nogood of partial embedding or extension M as $M[K]$, where $M[K] = \{(u_i, v) \in M \mid u_i \in K\}$. The following definition gives the mask for extensions that have conflicts.

Definition 3.24 (Conflict mask). Let M be a partial embedding, k be the length of M, and $v \in C(u_k, M)$. The *conflict mask* of extension $M \oplus v$ is $\emptyset$ if the extension has no conflict; otherwise, it is defined for each conflict case as follows.

(1) *Injectivity conflict*: $\{u_i, u_k\}$ where u_i holds $v = M(u_i)$.
(2) *Reservation-guard conflict*: $\{u_j \mid \exists v' \in R(u_i, v), (u_j, v') \in M\} \cup \{u_k\}$ where M matches $R(u_k, v)$.
(3) *Nogood-guard conflict*: $\mathrm{dom}(NV(u_k, v)) \cup \{u_k\}$ where M matches $NV(u_k, v)$.
(4) *No-candidate conflict*: $B(u_i;M \oplus v)$ where u_i holds $C(u_i;M \oplus v) = \emptyset$.

Example 3.25. Let $M = \{(u_0, v_0), (u_1, v_2), (u_2, v_6)\}$ and assume that M does not match any nogood guards on edges. $M \oplus v_{11}$ has the no-candidate conflict because v_6 and v_{11} lack a common neighbor in $C(u_4)$ and so $C(u_4;M \oplus v_{11}) = \emptyset$. Therefore, the conflict mask of $M \oplus v_{11}$ is $B(u_4;M \oplus v_{11}) = \{u_2, u_3\}$.

Lemma 3.26. *Let M be a partial embedding, k be the length of M, $v \in C(u_k, M)$, and K be the conflict mask of $M \oplus v$. If $M \oplus v$ has conflicts, $(M \oplus v)[K]$ is a nogood.*

Proof. We make a proof for each conflict case. Regarding the injectivity conflict, suppose that $K = \{u_i, u_k\}$. By Definition 3.24, we have $M[u_i] = v$, and thus $(M \oplus v)[K] = \{(u_i, v), (u_k, v)\}$. This is a nogood because of the violation of the injectivity constraint. Regarding the reservation-guard conflict, suppose that there exists full embedding $\hat{M}$ such that $(M \oplus v)[K] \subseteq \hat{M}$. We have $\mathrm{Im}(\hat{M}[:k]) \supseteq \mathrm{Im}(\hat{M}[K]) = \mathrm{Im}(M[K]) \supseteq R(u_k, v)$. In addition, $(u_k, v) \in \hat{M}$ because $u_k \in K$. From Lemma 3.6, $\hat{M}[:k] \cup \{(u_k, v)\}$ is a nogood. Thus, $\hat{M}$ includes a nogood, which is a contradiction. Therefore, $(M \oplus v)[K]$ is a nogood. Regarding the nogood-guard conflict, we have $(M \oplus v)[K] = NV(u_k, v) \cup \{(u_k, v)\}$, which is a nogood by Definition 3.16. Regarding the no-candidate conflict, suppose that there exists full embedding $\hat{M}$ such that $(M \oplus v)[K] \subseteq \hat{M}$. From $B(u_i;M \oplus v) \subseteq K$, we have $(M \oplus v)[B(u_i;M \oplus v)] \subseteq \hat{M}$. Here Lemma 3.22 gives $C(u_i;\hat{M}) \subseteq C(u_i;M \oplus v) = \emptyset$, which is a contradiction. Hence, $(M \oplus v)[K]$ is a nogood. We have shown that $(M \oplus v)[K]$ is a nogood for all the cases. □

A conflict mask indicates the conflicting assignments in an extension. However, even if an extension has no conflict, it may be found to be a deadend after the subsequent recursions where it failed to yield a full embedding. Hence, we generalize the definition for an extension with and without conflicts.

Definition 3.27 (Deadend mask). Let M be an extension and k be the length of M. In addition, for any v', suppose that $K_{v'}$ is the deadend mask of $M \oplus v'$. The *deadend mask* of M is given by K defined as follows.

(1) If M is not a deadend, $K = \emptyset$.
(2) If M has a conflict, K is the conflict mask of M.
(3) If some $v' \in C(u_k; M)$ holds $u_k \notin K_{v'}$, $K = K_{v'}$.
(4) Otherwise, $K = \bigcup_{v' \in C(u_k;M)} K_{v'} \cup B(u_k; M) \setminus \{u_k\}$.

Example 3.28. Let $M = \{(u_0, v_0), (u_1, v_2), (u_2, v_6)\}$. M does not have any conflict but is a deadend because $C(u_3; M) = \{v_{11}\}$ and $M \oplus v_{11}$ has the no-candidate conflict. Let $K_{v_{11}}$ be the deadend mask of $M \oplus v_{11}$. $K_{v_{11}}$ is given by the conflict mask of $M \oplus v_{11}$, which is $\{u_2, u_3\}$ (Example 3.25). Hence, the deadend mask of M is $K_{v_{11}} \cup B(u_3; M) \setminus \{u_3\} = \{u_2, u_3\} \cup \{u_2, u_3\} \setminus \{u_3\} = \{u_2\}$.

Lemma 3.29. *Let M be an extension and K be the deadend mask of M. If M is a deadend, $M[K]$ is a nogood.*

Proof. We make a proof by cases for each of cases (1)–(4) in Definition 3.27. Regarding case (1), we ignore this case because M is a deadend by hypothesis. Regarding case (2), by Lemma 3.26, $M[K]$ is a nogood. Regarding case (3), we prove this case by induction. The base case is case (2). Suppose that $v' \in C(u_k; M)$ gives $K_{v'}$ such that $u_k \notin K_{v'}$, and $(M \oplus v')[K_{v'}]$ is a nogood. From $K = K_{v'}$ and $u_k \notin K_{v'}$, we have $(M \oplus v')[K_{v'}] = M[K]$. Thus, $M[K]$ is a nogood. Regarding case (4), similar to case (3), we use induction using case (2) as the base case; suppose that, for all $v' \in C(u_k; M)$, $(M \oplus v')[K_{v'}]$ is a nogood. Moreover, we use a proof by contradiction; suppose that there exists full embedding $\hat{M}$ such that $M[K] \subseteq \hat{M}$. Then, we have $\hat{M}(u_k) \in C(u_k; \hat{M}) \subseteq C(u_k; M)$ because $B(u_k; M) \subseteq K$ and Lemma 3.22. Since $K \cup \{u_k\}$ includes the deadend mask of $M \oplus v'$ for all $v' \in C(u_k; M)$, it also includes the deadend mask of $M \oplus \hat{M}(u_k)$. Thus, $(M \oplus \hat{M}(u_k))[K \cup \{u_k\}]$ is a nogood. Since $(M \oplus \hat{M}(u_k))[K \cup \{u_k\}] = \hat{M}[K \cup \{u_k\}]$, $\hat{M}$ includes a nogood, which is a contradiction. Therefore, $M[K]$ is a nogood. We have proved that $M[K]$ is a nogood for all the cases. □

GuP discovers a nogood from a deadend by using the deadend mask. Specifically, when extension M has a conflict or is determined to be a deadend after the exploration, GuP obtains nogood $M[K]$ where K is the deadend mask of M. Then, letting (u_i, v) be the last assignment in $M[K]$ (i.e., $M[K] = \{\ldots, (u_i, v)\}$), GuP records $M[K] \setminus \{(u_i, v)\}$ in $NV(u_i, v)$. Such $NV(u_i, v)$ holds Definition 3.16. Note that $NV(u_i, v)$ is overwritten if it has an old value. In this way, GuP generates nogood guards on vertices during backtracking.

Example 3.30. Let $M = \{(u_0, v_0), (u_1, v_2), (u_2, v_6)\}$. Since deadend mask K of M is $\{u_2\}$ (Example 3.28), GuP records $M[\{u_2\}] \setminus \{(u_2, v_6)\} = \emptyset$ in $NV(u_2, v_6)$. Note that $\emptyset$ is a subset of an arbitrary set and hence (u_2, v_6) is never used in the subsequent search.

3.3.3 Nogood Guard on an Edge. Nogood guards on edges are used for filtering out candidate edges, reducing the number of local candidate vertices as shown in Definition 3.19. This enables us to detect the no-candidate conflict in earlier backtracking steps. In particular, as mentioned in Section 1, the cycles in a query graph must be mapped to cycles in a data graph, although cycles are difficult to find because of the sparseness of graphs. Such a search tends to involve many no-candidate conflicts, and thus by detecting them earlier, we can improve the search performance.

As defined in Definition 3.17, $NE((u_i, v), (u_j, v'))$ is a set of assignments such that $NE((u_i, v), (u_j, v')) \subseteq V_Q[: i] \times V_G$ holds and D is a nogood, where $D = NE((u_i, v), (u_j, v')) \cup \{(u_i, v), (u_j, v')\}$.

This definition implies that there are no assignments of u_k in D where $i < k < j$. A nogood discovered using a deadend mask may not satisfy this condition, and hence we need another rule to discover a nogood for a nogood guard on an edge.

For conciseness of the discussion, we relax the format of the nogood as follows. Assume that M is a partial embedding whose length is $i + 1$ (i.e., M contains an assignment of u_i). Then, our goal is to find mask $K \subseteq V_Q$ such that $M[K] \cup \{(u_j, v')\}$ is a nogood. This allows us to discuss this problem for an arbitrary combination of partial embedding M and candidate vertex (u_j, v') regardless of the existence of candidate edge $((u_i, M(u_i)), (u_j, v'))$. We formally define such mask K.

Definition 3.31 (Fixed deadend mask). Let M be an extension, k be the length of M, and (u_i, v) be a candidate vertex. In addition, for any v', suppose that $K_{v'}$ is the (u_i, v)-fixed deadend mask of $M \oplus v'$. The (u_i, v)-*fixed deadend mask* of M is given by K defined as follows.

(1) If $i < k$ holds, $K = K' \setminus \{u_i\}$ where K' is the deadend mask of $M[: i] \oplus v$.
(2) If there is a full embedding that includes $M \cup \{(u_i, v)\}$, $K = \emptyset$.
(3) If M has a conflict, K is the conflict mask of M.
(4) If M holds $v \notin N(M(u_j))$ for some $u_j \in N(u_i)$, $K = \{u_j\}$.
(5) If M matches $NE((u_j, v'), (u_i, v))$ for some $(u_j, v') \in M$, $K = \mathrm{dom}(NE((u_j, v'), (u_i, v))) \cup \{u_j\}$.
(6) If there is $v' \in C(u_k; M)$ that holds $u_k \notin K_{v'}$, $K = K_{v'}$.
(7) Otherwise, $K = \bigcup_{v' \in C(u_k;M)} K_{v'} \cup B(u_k; M) \setminus \{u_k\}$.

The definition of the (u_i, v)-fixed deadend mask resembles that of the deadend mask. The main differences are that (i) $K_{v'}$ is recursively given by (u_i, v)-fixed deadend mask, and (ii) it has conditions on u_i and v (cases (1), (4), and (5)). Case (1) is the base case defined using the deadend mask. Cases (4) and (5) handle the case that v is not a local candidate vertex of u_i.

Example 3.32. Let $M = \{(u_0, v_0), (u_1, v_2), (u_2, v_7)\}$ and consider the (u_4, v_0)- and (u_4, v_1)-fixed deadend masks of M. Since $M \oplus v_{10} \oplus v_0$ has the injectivity conflict, its (u_4, v_0)-fixed deadend mask is $\{u_0, u_4\}$ by case (3) of Definition 3.31. Then, (u_4, v_0)-fixed deadend mask of $M \oplus v_{10}$ is $\{u_0, u_4\} \cup B(u_4; M \oplus v_{10}) \setminus \{u_4\} = \{u_0, u_2, u_3\}$ by case (7). It follows that (u_4, v_0)-fixed deadend mask of M is $\{u_0, u_2, u_3\} \cup B(u_3; M) \setminus \{u_3\} = \{u_0, u_2\}$. On the other hand, the (u_4, v_1)-fixed deadend mask of $M \oplus v_{10}$ is $\{u_3\}$ by case (4) because $v_{10} \notin v_{10}$. Hence, by case (7), (u_4, v_1)-fixed deadend mask of M is $\{u_3\} \cup B(u_3; M) \setminus \{u_3\} = \{u_2\}$.

LEMMA 3.33. *Let M be an extension, (u_i, v) be a candidate vertex, and K be the (u_i, v)-fixed deadend mask of M. Suppose that $|M| \leq i$ and $M \cup \{(u_i, v)\}$ is a nogood. Then, $M[K] \cup \{(u_i, v)\}$ is a nogood.*

PROOF. We make a proof by cases for cases (1), (4), and (5) in Definition 3.31 and omit the others because they can be proved similarly to the proof of Lemma 3.29. Regarding case (1), let K' be the deadend mask of $M[: i] \oplus v$. We have $(M[: i] \oplus v)[K'] = M[: i][K' \setminus \{u_i\}] \cup \{(u_i, v)\} = M[K] \cup \{(u_i, v)\}$ because $M[: i] = M$ holds by hypothesis $|M| \leq i$. Hence, $M[K] \cup \{(u_i, v)\}$ is a nogood. Regarding case (4), we have $M[K] \cup \{(u_i, v)\} = \{(u_j, M(u_j), (u_i, v)\}$. By hypothesis, $(u_j, u_i) \in E_Q$ and $(M(u_j), v) \notin E_G$ hold, and thus $M[K] \cup \{(u_i, v)\}$ is a nogood because of the violation of the adjacency constraint. Regarding case (5), let M' be an arbitrary partial embedding such that $M[K] \cup \{(u_i, v)\} \subseteq M'$. Since $\mathrm{dom}(NE((u_j, M(u_j)), (u_i, v))) \subseteq K$ holds, we have $NE((u_j, M(u_j)), (u_i, v)) \subseteq M'$, and thus $M'[: j] \cup \{(u_j, M(u_j)), (u_i, v)\}$ is a nogood by Definition 3.17 From $u_j \in K$, we have $M'(u_j) = M(u_j)$. Hence, $M'[: j] \cup \{(u_j, M(u_j)), (u_i, v)\} \subseteq M'$ holds, which means M' is a deadend. It follows that $M[K] \cup \{(u_i, v)\}$ is a nogood. We have proved the statement. □

During backtracking, GuP updates nogood guards on edges as follows. Suppose that M is a partial embedding of length i, $C(u_i; M)$ contains v, and there exists candidate edge $((u_i, v), (u_j, v'))$. If extension $M \oplus v$ did not yield any full embedding containing (u_j, v') in the subsequent recursions, GuP

Algorithm 2 Backtrack(M): Backtracking procedure of GuP

1: **if** $|M| = |V_Q|$ **then** output M and **return**
2: $k \leftarrow |M|$
3: **for** $v \in C(u_k; M)$ **do**
4:   **if** $v \in \mathrm{Im}(M)$ **then continue** ▷ Injectivity conflict
5:   **if** M matches $R(u_k, v)$ or $NV(u_k, v)$ **then continue** ▷ Reservation- and nogood-guard conflicts
6:   **if** $C(u_i; M \oplus v) = \emptyset$ for some u_i **then continue** ▷ No-candidate conflict
7:   Backtrack($M \oplus v$) ▷ Find all the full embeddings yielded by $M \oplus v$
8:   **for** $u_i \in N_+(u_k)$ and $v' \in C(u_i; M \oplus v)$ **do**
9:     **if** $M \oplus v$ yielded no full embedding containing (u_i, v') **then**
10:       $NE((u_k, v), (u_i, v')) \leftarrow M[K]$, where K is the (u_i, v')-fixed deadend mask of $M \oplus v$
11:   **if** $M \oplus v$ yielded no full embedding **then**
12:     $D \leftarrow (M \oplus v)[K]$, where K is the deadend mask of $M \oplus v$ ▷ D is a nogood
13:     $NV(u_k, v) \leftarrow D \setminus \{(u_k, v)\}$
14:     **if** $D \subseteq M$ **then return** ▷ Backjumping

computes the (u_j, v')-fixed deadend mask K of $M \oplus v$. Then, GuP records $M[K]$ in $NE((u_i, v), (u_j, v'))$. This holds Definition 3.17 because $M[K] \cup \{(u_i, v), (u_j, v')\} = (M \oplus v)[K] \cup \{(u_j, v')\}$, which is a nogood by Lemma 3.33.

Example 3.34. Let $M = \{(u_0, v_0), (u_1, v_2)\}$. Since the (u_4, v_0)-fixed deadend mask of $M \oplus v_7$ is $\{u_0, u_2\}$ (Example 3.32), GuP records $M[\{u_0, u_2\}]$ $(= \{(u_0, v_0)\})$ in $NE((u_2, v_7), (u_4, v_0))$. In addition, since the (u_4, v_1)-fixed deadend mask of $M \oplus v_7$ is $\{u_2\}$, GuP records $M[\{u_2\}]$ $(= \emptyset)$ in $NE((u_2, v_7), (u_4, v_1))$, which filters out $((u_2, v_7), (u_4, v_1))$ for all partial embeddings.

Since generating a nogood guard incurs slight overhead, our implementation omits nogood guards on candidate edges outside of the 2-core of a query graph. This is because the outside of the 2-core consists of trees although, as mentioned above, nogood guards on edges are effective for pruning in the search of cyclic structures. This optimization allows GuP to benefit from pruning opportunities offered by nogood guards on edges without sacrificing efficiency.

3.4 Backtracking with Guards

GuP efficiently performs backtracking by utilizing reservation guards and nogood guards as shown in Algorithm 2. Calling Backtrack($\emptyset$) starts the search. Backtrack(M) recursively extends partial embedding M with $v \in C(u_k; M)$ until it obtains a full embedding. The algorithm first checks if $M \oplus v$ has a conflict defined in Definition 3.23. If it has no conflict, Backtrack is recursively called with $M \oplus v$. After the recursion, nogood guards on vertices and edges are updated. The algorithm also performs backjumping if a nogood is discovered in M. Backjumping abandons all the remaining candidate vertices in $C(u_k; M)$ by immediately returning to the caller. Note that we omit the details of optimizations from the pseudocode to keep it concise. Our implementation incrementally computes local candidate-vertex sets, bounding sets, and (fixed) deadend masks by fusing the computation into the backtracking. For example, letting $u_i \in N_+(u_k)$, $C(u_i; M \oplus v)$ is obtained from $C(u_i; M)$ as follows:

$$C(u_i; M \oplus v) = \{v' \in C(u_i; M) \cap N(v) \mid M \text{ does not match } NE((u_k, v), (u_i, v'))\} \tag{2}$$

Guard-based pruning improves the search efficiency by filtering out unnecessary candidate vertices before the recursion (lines 4-6) and the computation of local candidate-vertex sets (Eq. (2)).

Example 3.35. The process of backtracking can be considered as a depth-first search on a *search tree* whose node corresponds to a recursive call with an extension. Fig. 3 shows the search tree of conventional backtracking. X-marks indicate conflicting assignments. Fig. 2 shows all the reservation guards and the nogood guards on vertices when the backtracking search reaches search node m_6, which corresponds to $M_6 = \{(u_0, v_0), (u_1, v_3)\}$. Our backtracking algorithm now tries extending M_6 with each $v \in C(u_2; M_6)$ $(= \{v_5, v_6, v_7\})$, but all of them are filtered out by $R(u_2, v_5)$, $NV(u_2, v_6)$, and $NV(u_2, v_7)$, respectively. Hence, the procedure returns to node m_1, which corresponds to $M_1 = \{(u_0, v_0)\}$. Since $M_1 \oplus v_3$ $(= M_6)$ was found to be a deadend, GuP computes deadend mask K of M_6. From $K_{v_5} \cup K_{v_6} \cup K_{v_7} = \{u_0, u_2\}$ and $B(u_2; M_6) = \{u_0\}$ (Example 3.21), $K = \{u_0, u_2\} \cup \{u_0\} \setminus \{u_2\} = \{u_0\}$ by case (4) of Definition 3.27. Thus, GuP discovers nogood $D = M_6[K] = \{(u_0, v_0)\}$. Since $D \subseteq M_1$ holds, the procedure performs backjumping to the caller (line 14), pruning search node m_{12}. As a whole, our approach prunes the shadowed nodes in Fig. 3.

THEOREM 3.36. *GuP finds all the full embeddings.*

PROOF. Let M be a partial embedding, k be the length of M, and $v \in C(u_k; M)$. Since GuP without guard-based pruning and backjumping performs a conventional backtracking search [38], we prove that v is filtered out only if $M \oplus v$ is a deadend. Regarding guard-based pruning, Algorithm 2 filters out v from $C(u_k; M)$ if M matches $R(u_k, v)$, $NV(u_k, v)$, or there exists u_j $(j < k)$ such that M matches $NE((u_j, M(u_j)), (u_k, v))$. If M matches $R(u_k, v)$, $M \oplus v$ is a deadend (Lemma 3.6). Similarly, if M matches $NV(u_k, v)$ or $NE((u_j, M(u_j)), (u_k, v))$, $M \oplus v$ includes a nogood (Definitions 3.16 and 3.17), and thus it is a deadend (Definition 3.15). Regarding backjumping, Algorithm 2 returns to the caller if M includes a nogood, which means that $M \oplus v$ is a deadend for any v. Therefore, GuP finds all the full embeddings. □

Comparison with failing set-based pruning. Failing set-based pruning proposed in DAF [14] is a backjumping [32] method that is widely used in the database community [20, 35, 37]. Although both DAF and GuP exploit nogoods for pruning, GuP is more effective for two reasons. First, GuP reuses a discovered nogood for pruning multiple times, whereas DAF discards a nogood after using it for backjumping. Note that GuP also performs backjumping (line 14 in Algorithm 2). Second, GuP discovers smaller nogoods, which offer higher pruning power. Like a deadend mask, DAF discovers a nogood using a failing set, which is defined as a set of query vertices and all their ancestors in terms of the matching order. Owing to the ancestors, a failing set tends to be large and so offers a large nogood. For example, a failing set of $M_6 = \{(u_0, v_0), (u_1, v_3)\}$ is $\{u_0, u_1\}$, and this fails to trigger a backjumping at search node m_1. On the other hand, GuP produces a small deadend mask, $\{u_0\}$, and can prune search node m_{12} by backjumping (Example 3.35). Thus, our nogood discovery rule enables more effective pruning.

Parallelization. We can easily parallelize backtracking of GuP, which tends to dominate query processing time, by searching different subtrees of the search tree in different threads. Since the size of the search space is unknown in advance and usually very skewed, a work-stealing approach needs to be used that dynamically splits the search tree and assigns it to an idle thread for load balancing. Threads share the candidate vertices and edges and the reservation guards in a GCS but maintain thread-local nogood guards because those are modified during parallel backtracking. Since the pruning efficiency may degrade because the information of nogoods is not shared between threads, we empirically evaluate it in Section 4.3.4.

3.5 Search-node Encoding

Nogood guards reduce futile recursions, but their matching tests incur nonnegligible computational costs. Consider the matching test between partial embedding M and $NV(u_i, v)$. It takes $O(|NV(u_i, v)|)$ time to check if $M(u_j) = v'$ or not for each $(u_j, v') \in NV(u_i, v)$. This is the same for nogood guards on edges. The size of a nogood guard can be up to $|V_Q| - 1$, and GuP performs the matching test many times for filtering out candidate vertices and edges. Thus, this overhead may spoil the performance benefit resulting from pruning.

To mitigate the overhead, we introduce a search-node encoding, which represents a nogood with a node in the search tree. Since each node corresponds uniquely to the partial embedding at that backtracking step, we refer to the node corresponding to partial embedding M as the *search node* of M. Suppose that m_i and m_j are the search nodes of partial embeddings M and M'. Then, if M' is an extension of M (i.e., $M \subseteq M'$), m_j is a descendant of m_i in the search tree. It can be checked in $O(1)$ time by maintaining the *ancestor array* of m_j, denoted by *anc*. Assuming that there exists an imaginary root node m_0, which corresponds to the empty partial embedding, *anc* contains m_0 at $anc(0)$, a length-1 partial embedding at $anc(1)$, its child at $anc(2)$, and so on. $anc(|M'|)$ is set to m_j. Here, if $anc(|M|) = m_i$ holds, m_j is a descendant of m_i. This also means we can check if M is a subset of M' in $O(1)$ time.

Example 3.37. On the search tree shown in Fig. 3, let us check if M_3 is a subset of M_5 where $M_3 = \{(u_0, v_0), (u_1, v_2), (u_2, v_6)\}$ and $M_5 = \{(u_0, v_0), (u_1, v_2), (u_2, v_7), (u_3, v_{10})\}$. In the search tree, node m_3 and m_5 correspond to M_3 and M_5, respectively. Ancestor array *anc* of m_5 contains the IDs of m_0 (imaginary root node), m_1, m_2, m_4, m_5 in $anc(0)$ to $anc(4)$. From $anc(|M_3|) = anc(3) = m_4$ and thus $anc(|M_3|) \neq m_3$, we can find that M_3 is not a subset of M_5.

For applying this idea to matching tests with nogood guards, GuP encodes nogood guards into the ID of a search node. Since a nogood guard is a subset of a partial embedding, it may not have a corresponding search node. Thus, GuP "rounds up" a nogood guard to the smallest partial embedding that is a superset of a nogood guard.

Definition 3.38 (Minimum superset embedding). Let M be a partial embedding and D be a subset of M. The *minimum superset embedding* of D in M is $M[:i+1]$ where i is the ID of the query vertex lastly assigned in D (i.e., $D = \{\dots, (u_i, v)\}$).

In the definition above, letting *anc* be the ancestor array of M, we can obtain the search node of minimum superset embedding $M[:i+1]$ as $anc(i)$. By using this, GuP stores nogood guards in a GCS as follows. Regarding $NV(u_i, v)$ obtained from deadend partial embedding M, let *anc* be the ancestor array of M and L be the minimum superset embedding of $NV(u_i, v)$ in M. Then, $NV(u_i, v)$ is stored as a triplet (id, len, K) where $id = anc(|L|)$, $len = |L|$, and $K = \mathrm{dom}(NV(u_i, v))$. Given partial embedding M' and its ancestor array anc', we can check if M' matches $NV(u_i, v)$ by $anc'(len) = id$. K is used to obtain $\mathrm{dom}(NV(u_i, v))$ in nogood discovery for the case of the nogood-guard conflict (Definition 3.24). Nogood guards on edges are stored in the same way. Thanks to the lightweight matching test with search-node encoding, GuP can efficiently filter out candidate vertices and edges.

3.6 Complexity Analysis

We first analyze the time complexity of each of three steps listed in Section 3.1 and then discuss the space complexity of the whole of GuP. The following analyses assume that a bit vector of length $|V_Q|$ takes $O(1)$ space and $O(1)$ time for set operations, such as union and intersection, since a query graph is supposed to be small.

Time complexity of the GCS construction. GuP employs extended DAG-graph DP, which provides a candidate space through candidate filtering in $O(|E_Q||E_G|)$ time [20]. Candidate filtering and GCS construction of GuP have the same complexity because we can obtain a GCS by attaching a null-valued guard to each candidate vertex and edge during extended DAG-graph DP. GuP also adopts VC for optimizing the matching order, whose complexity is $O(|E_Q||E_G|)$ [36]. Therefore, the complexity of this step is $O(|E_Q||E_G|)$.

Time complexity of the reservation guard generation. In the following, we show that Algorithm 1 takes $O(|E_Q||E_G|)$ time. Let $\bar{d}_Q$ and $\bar{d}_G$ be the average degrees of Q and G, respectively. The loop over the candidate vertices (line 1) iterates up to $|V_Q||V_G|$ times, and the loop over forward neighbors of a query vertex (line 3) iterates $\bar{d}_Q$ times. The complexity for computing E_R by Eq. (1) (line 4) is bounded by the size of E_R, which is $O(\sum_{v' \in N(v) \cap C(u_j)} |R(u_j, v')|) = O(|N(v)| \times r) = O(\bar{d}_G)$ since r is a constant. After that, Algorithm 1 solves the vertex-cover problem for graph $G_R = (V_R, E_R)$. V_R consists of both endpoints of the edges, and thus $|V_R| \leq 2|E_R|$ holds. We employ the 2-approximation algorithm [9], whose complexity is $O(|V_R| + |E_R|) = O(|E_R|) = O(|\bar{d}_G|)$. Therefore, the whole complexity of Algorithm 1 is $O(|V_Q||V_G|\bar{d}_Q\bar{d}_G) = O(|E_Q||E_G|)$.

Time complexity of backtracking search. Matching with a reservation guard takes $O(1)$ time because its size is bounded by r. It also takes $O(1)$ time to perform matching with and generation of a nogood guard with search-node encoding as shown in Section 3.5. Hence, the complexity of backtracking is determined by the number of recursions. Although it is $O(\prod_{u_i} |C(u_i)|) = O(|V_G|^{|V_Q|})$ in the worst case [40], the number significantly decreases in practice thanks to candidate filtering and guards. However, theoretically analyzing their contribution is difficult because of their sensitivity to input graphs. Thus, following previous studies [14, 20, 35, 37], we experimentally evaluate it in Section 4.

Space complexity. Since every part of GuP focuses only on candidate vertices and edges, a GCS dominates the space complexity of GuP. A reservation guard consists of up to r data vertices, and hence its size is regarded as $O(1)$. A nogood guard in search-node encoding also takes $O(1)$ space because it is a triplet of integers and a bit vector of query vertices, whose size is $O(1)$. Therefore, the space complexity of a GCS is the same as that of a candidate space, which is $O(|E_Q||E_G|)$ [14, 35].

Comparison with existing methods. If we leave out the exponential time complexity of backtracking, $O(|E_Q||E_G|)$ is a common time and space complexity among recent methods [2, 14, 20, 37]. GQL [16] has an even higher time complexity due to semi-perfect matching [35]. However, the practical performance of subgraph matching largely depends on that of backtracking, and hence we experimentally show it in Section 4.

4 EVALUATION

In this section, we compare the performance of GuP with existing methods and analyze GuP from various aspects.

4.1 Experimental Setup

Methods. We compared the performance of GuP with the following methods[1]: DAF [14], GQL-G [35], GQL-R [35], and RapidMatch (RM) [37]. All of them have been proposed in the last several years and use failing set-based pruning. GQL-G and GQL-R are combinations of candidate filtering of GraphQL [16] and the matching orders of GraphQL and RI [5], respectively. They performed the best in the evaluation by Sun et al. [35]. Every implementation was obtained from the authors' GitHub repository[2]. We set r, the size limit of reservation guard, to 3 unless otherwise specified.

[1]We also tried to measure the performance of VEQ [20], but the binary obtained from https://github.com/SNUCSE-CTA/VEQ crashes during the process of over thousands of query graphs used in our experiment. Thus, we omitted its results for a fair comparison.

[2]DAF: https://github.com/SNUCSE-CTA/DAF

GQL-G and GQL-R: https://github.com/RapidsAtHKUST/SubgraphMatching

RapidMatch: https://github.com/RapidsAtHKUST/RapidMatch

Table 2. Finished (i.e., non-DNF) query sets

	Human								WordNet								Patents								Count
	8S	16S	24S	32S	8D	16D	24D	32D	8S	16S	24S	32S	8D	16D	24D	32D	8S	16S	24S	32S	8D	16D	24D	32D	
GuP	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	20
DAF	✓				✓				✓								✓	✓			✓	✓	✓		8
GQL-G	✓	✓		✓	✓	✓			✓	✓	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓		17
GQL-R	✓	✓			✓				✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓		16
RM	✓				✓				✓	✓	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓		14

Graphs. We used the following four data graphs: Yeast (3,112 vertices, 12,519 edges, 71 labels), Human (4,674 vertices, 86,282 edges, 44 labels), WordNet (76,853 vertices, 120,399 edges, 5 labels), and Patents (3,774,768 vertices, 16,518,947 edges, 20 labels). The first three graphs are labeled real-world graphs and are common in studies of subgraph matching [14, 20, 35–37]. The last one, Patents, is the largest graph used in recent studies [35, 36]. This is an unlabeled graph, and thus we gave it the randomly-assigned labels used in the evaluation by Sun et al. [35], which is publicly available[3]. We generated query graphs also in the same manner as Sun et al. Specifically, we performed a random walk on a data graph and extracted a subgraph induced by the visited vertices as a query graph. A query graph is classified as a sparse query graph if its average degree is less than three; otherwise, it is classified as a dense query graph. We generated query sets of sparse and dense query graphs by changing the number of vertices. Query sets of sparse query graphs are 8S, 16S, 24S, and 32S, and those of dense query graphs are 8D, 16D, 24D, and 32D. Thus, there are 32 query sets in total for four data graphs, four sizes, and two densities. Each query set contains 50,000 query graphs. While it is common to make a query set of 100 or 200 query graphs [3, 14, 20, 35–37], such a set is too small considering that an n-vertex query graph has $(n-1)!$ possible topologies and $|\Sigma|^n$ possible label assignments. Although certain applications such as crime detection [29, 31] focus on subgraphs that rarely occur in a data graph, they tend not to be extracted as a query graph. Thus, large query sets are necessary to extensively evaluate the efficiency of each method.

Machine and terminate conditions. We conducted the experiments on a machine with four Intel Xeon E7-8890 v3 processors (18 cores per socket, and thus 72 cores in total) and 2 TB of memory. Except for the evaluation of parallel processing (Section 4.3.4), all the methods were executed in a single thread using one physical core exclusively. To reduce the experimental time, we used up to 70 cores to run 70 experiments simultaneously. Similarly to the existing studies [3, 14, 20, 35, 36], we terminated the search for a query graph when 10^5 embeddings were discovered. We set a time limit for a query graph and a query set, respectively. A search for a single query graph was terminated after one hour. On the other hand, the query set was divided into subgroups of 100 query graphs, and when the total processing time of any subgroup exceeded three hours, the whole query set was judged as "did not finish" (DNF).

4.2 Comparison with Existing Methods

We first focus on the distribution of the processing time of each query and then show the average time. We consider the distribution more informative because the average is largely affected by the setting of the time limit; specifically, a short time limit hides the impact of expensive query graphs, and in contrast, a long time limit lets expensive query graphs dominate the results.

[3]https://github.com/RapidsAtHKUST/SubgraphMatching#experiment-datasets

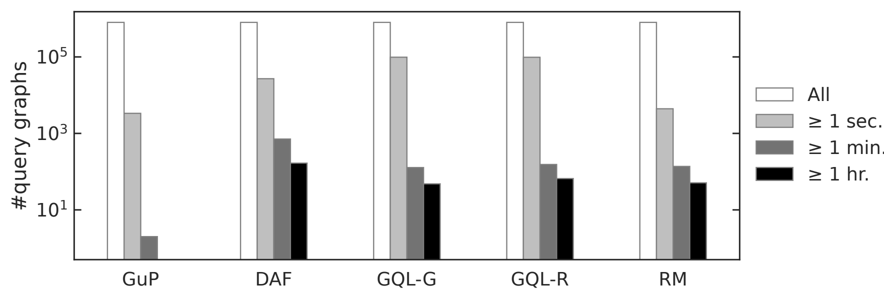


Fig. 4. Total number of query graphs for each processing time range.

4.2.1 Distribution of Processing Time. Table 2 shows query sets that each method finished, specifically, processed all the query graphs avoiding a DNF. GuP finished the most query sets. In addition, GuP is the only method that could finish 24S of Human and 32D of Patents. Fig. 4 shows the processing time distribution of query graphs. To equalize the number of query graphs for all the methods, we focused on 16 query sets for which no method yielded a DNF. The "All" bar indicates 800,000 ($50,000 \times 16$) query graphs in those query sets. We counted the number of query graphs that took a processing time more than or equal to the following thresholds: one second ("$\geq$ 1 sec."), one minute ("$\geq$ 1 min."), and one hour ("$\geq$ 1 hr."). Note that, since the time limit per query graph is set to one hour, all the query graphs that took more than an hour were terminated before their completion. As shown in the figure, GuP yielded the fewest query graphs for all the thresholds. Most notably, GuP has no query graphs that took more than an hour. The overall results in Table 2 and Fig. 4 demonstrate the high robustness of GuP, which enables the method to process query graphs that the state-of-the-art methods cannot handle within a practical time frame.

Next, we present the processing time distribution for query sets 16S, 32S, 16D, and 24D of each data graph. Like Fig. 4, Fig. 5 shows bars of the number of query graphs that took more than a second, a minute, and an hour. Instead of the "All" bars, the top of the Y axis is set to 50,000, the number of query graphs in each query set. GQL-G and GQL-R are shown as "G.-G" and "G.-R" because of space limitation. GuP showed shorter query processing time than the existing methods as a whole, and we find that GuP performs stably for various query graphs and data graphs. In addition, GuP always yielded the fewest query graphs that took more than an hour, except for 16D of WordNet. This proves that GuP can effectively reduce the search space of difficult queries.

4.2.2 Average Processing Time. Fig. 6 presents the average processing time of each query graph in the query sets of Yeast. The results for the other data graphs are omitted because of a lack of comparability caused by DNF query sets. Timed-out query graphs are counted as if they were completed in one hour, which is the time limit per query graph. This figure reveals another aspect of the performance because Figs. 4 and 5 classify query graphs into the ranges of the processing time and do not care about the actual value of the processing time in each range. Since generation of and matching with guards involve additional overheads, GuP yielded only moderate performance for 8- and 16-vertex query graphs. However, GuP became one of the best methods for 24- and 32-vertex query graphs because larger query graphs have larger search space, where the performance gain offered by guards more easily surpasses the overheads. As we can confirm from the processing time distribution and the number of DNFs depicted in Fig. 5, the query graphs of the other data graphs are even more difficult to solve, and hence GuP tends to perform better than the other methods.

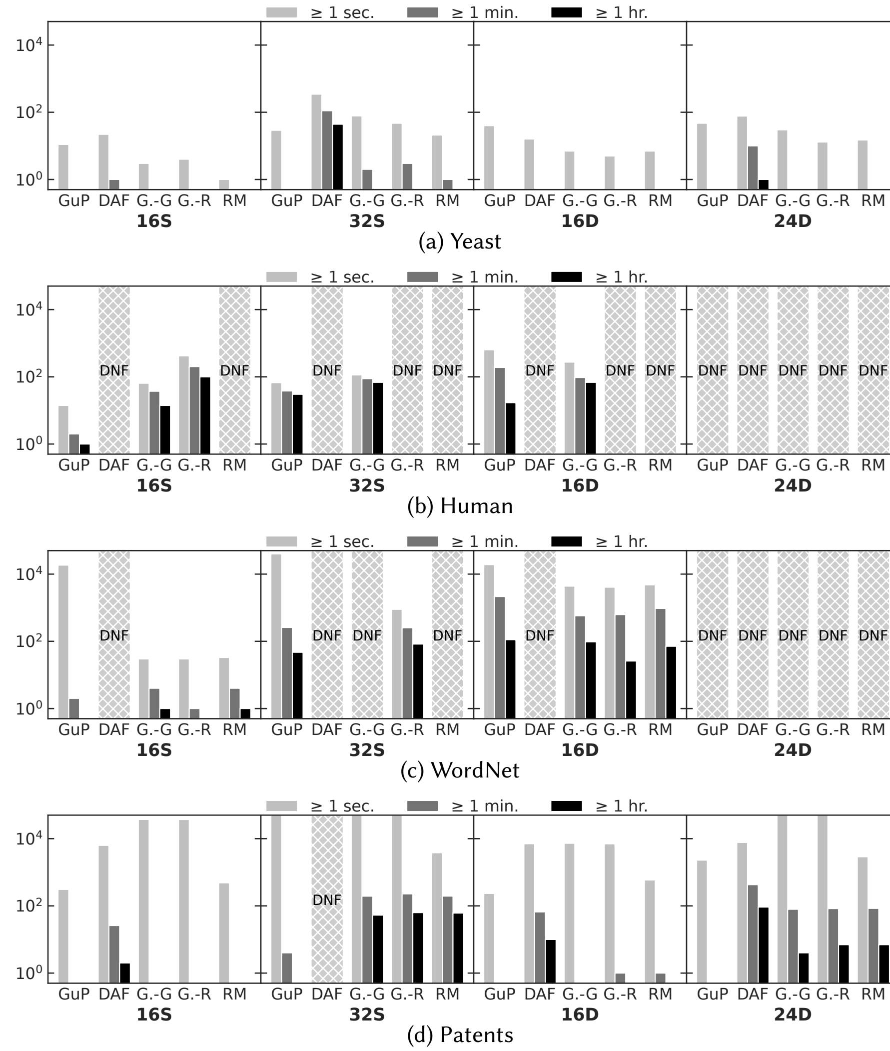


Fig. 5. Breakdown of the number of query graphs for each processing time range.

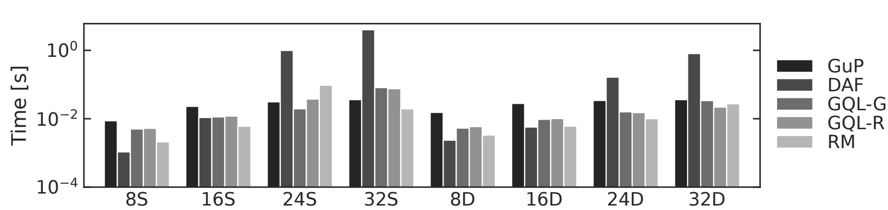


Fig. 6. Average processing time for each query set of Yeast.

4.2.3 Number of Recursions. To evaluate the size of the search space, we compared the number of recursive calls of the backtracking procedure. Fig. 7 shows the total number of recursions needed

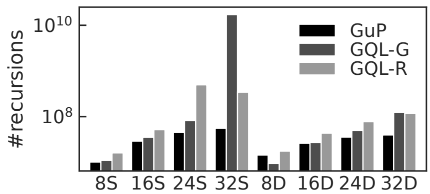


Fig. 7. Number of recursions on Yeast.

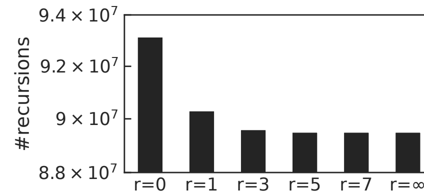


Fig. 8. Parameter search for reservation size r.

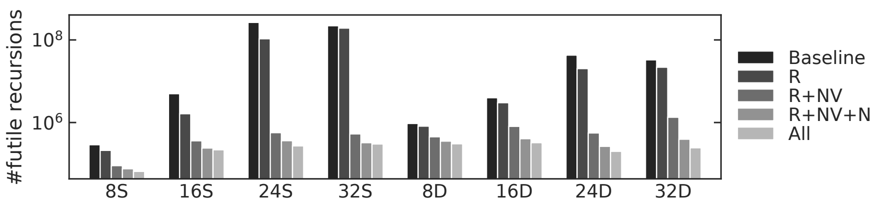


Fig. 9. Number of futile recursions on Yeast.

to process each query set of Yeast. We omitted DAF and RM because they are not completely based on recursive search; DAF uses leaf decomposition [3] besides backtracking, and RM is a join-based method. As shown in the figure, GuP produced the fewest number of recursions for most of the query sets. This result shows that the high performance of GuP is derived from the reduced search space. Note that, owing to overheads related to guards, GuP showed longer average processing time in Fig. 6 despite fewer recursions.

We also counted the number of local candidate vertices adaptively filtered out by guards during backtracking. While we omit the detailed results, 11.5% of local candidate vertices were pruned on average. This may seem a slight reduction but greatly impacts the number of recursions because it is determined by the multiplication of the number of local candidate vertices. For example, if we have a 32-vertex query graph, and 11.5% of local candidate vertices are pruned for every query vertex, the number of recursions decreases to 2% ($(1 - 0.115)^{32} = 0.02$).

4.3 Detailed Analysis of GuP

Next, we show the results of the experiments to understand the characteristics of GuP.

4.3.1 Reservation Size. GuP requires parameter r, which specifies the maximum size of reservation guards. Fig. 8 shows the total number of recursions needed to process 1,000 query graphs in each query set of Yeast. "r = ∞" indicates no limitation on the size. We disabled the pruning techniques except for reservation guards. The results show that the pruning power of reservation guards increases as r increases, but it almost saturates at $r = 3$. This is because large reservation guards are rarely matched by partial embeddings and so ineffective in pruning. We also found almost the same trends regarding the other data graphs. From this result, we recommend $r = 3$ as the default setting because r is preferred to be small to reduce the computational costs of the reservation guard generation and a matching test with reservation guards. Remember that we always used $r = 3$ except for this experiment.

4.3.2 Effectiveness of Each Guard. Next, we investigated the effectiveness of each guard. To better understand the contribution of guards, here we focused on the number of *futile* recursions, which are recursive calls leading to a deadend. Fig. 9 shows the number of futile recursions offered by the

Table 3. Peak memory consumption

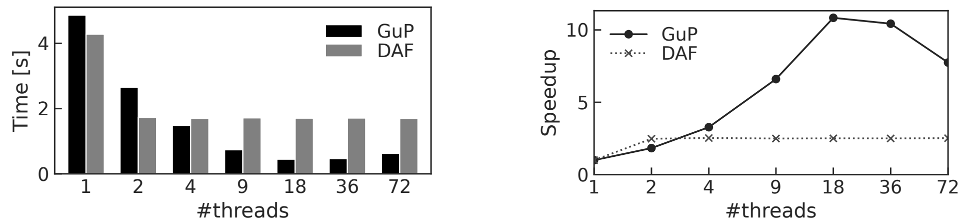
Graph	Query	Whole	Guard *R*	Guard *NV*	Guard *NE*	Guard/Whole
Yeast	8S	2.91 MB	0.07 MB	0.09 MB	0.61 MB	26.49%
	32S	4.21 MB	0.11 MB	0.13 MB	0.83 MB	25.36%
	8D	3.37 MB	0.04 MB	0.05 MB	0.81 MB	26.59%
	32D	4.27 MB	0.07 MB	0.08 MB	1.00 MB	26.93%
Patents	8S	1.51 GB	1.43 MB	1.86 MB	2.14 MB	0.36%
	32S	1.51 GB	2.53 MB	3.09 MB	4.31 MB	0.66%
	8D	1.51 GB	0.50 MB	0.67 MB	0.68 MB	0.12%
	32D	1.51 GB	1.39 MB	1.66 MB	3.04 MB	0.40%

different combinations of techniques in GuP. "Baseline" means a conventional backtracking search, "R", "NV", and "NE" mean the use of reservation guards, nogood guards on vertices, and nogood guards on edges, respectively. Finally, "All" means complete GuP, equivalent to "R+NV+NE" with backjumping. We can see the overall trend that nogood guards on vertices contributed the most to reducing futile recursions. The contribution of nogood guards on edges is the second largest, and backjumping offers slightly more improvement. Although the contribution of reservation guards varied among the query sets, they substantially decreased the number of futile recursions for 16S, 24S, and 24D by 77%, 60%, and 53%, respectively. Thus, all the techniques in GuP contribute to achieving efficient backtracking, leading to the high robustness of GuP.

4.3.3 Memory Consumption. Since GuP needs additional memory space for guards, we evaluated its memory consumption using Yeast and Patents, the smallest and largest data graphs in our experiment. Table 3 shows the results observed during the process of each query set. The "Whole" column shows the peak heap memory consumption[4], the columns under "Guard" shows the maximum memory consumption of each guard, and "Guard/Whole" shows the percentage of the total memory consumption of the guards in the whole memory consumption. While guards occupied about one-fourth of the whole memory consumption for Yeast, the percentage decreased to under 1% for Patents. This is because the memory consumption for Patents is dominated by the data graph. The program needs much temporary memory for buffering data read from files and constructing a data structure of the data graph. In contrast, guards consume little memory because they are attached to candidate vertices and edges, which are much fewer than the vertices and edges of the data graph. Guards are generated after releasing memory for the temporary data, and hence the peak memory consumption for Patents was 1.51 GB regardless of the size of query graphs. Note that this seems a reasonable memory consumption because we observed that GQL-G and GQL-R also allocated about 1.5 GB of memory for Patents. As shown by these results, guard-based pruning can be applied to large-scale graphs.

4.3.4 Parallelization. We parallelized our implementation by the approach described in Section 3.4 and compared its performance with DAF, which is the only parallelized method among the methods used in the experiment. Parallel search often offers superlinear speedups [14] when a thread encountered a search space where it could easily find embeddings more than the limit, which is 10^5 in our experiment. This is beneficial in practice but becomes noise in a study of parallel scalability. To mitigate this effect, we increased the limit to 10^8 in this experiment. Section 4.3.4 shows average processing time and speedup for 1,000 query graphs in 32D of Yeast with different numbers of threads. The 1-thread performance differs from Fig. 6 because of the different limit

[4]We used heaptrack to obtain these values: https://github.com/KDE/heaptrack.



(a) Average processing time

(b) Speedup

Fig. 10. Performance in parallel executions.

on the number of embeddings. For 1- and 2-thread executions, GuP performed worse than DAF due to guard overheads and the superlinear speedup of DAF. However, the performance of DAF does not scale to more than two threads. This is because DAF parallelizes the search only at the candidate vertices of u_0 [14] and thus failed at load balancing. In contrast, thanks to work stealing, GuP offered speedup almost in proportion to the number of threads and outperformed DAF with threads more than two. Since our machine consists of four NUMA nodes each of which has 18 cores, communication costs degraded the 36- and 72-thread performances. NUMA optimization will improve the performance, but it is not the focus of this paper.

In the parallel execution, each thread of GuP individually maintains nogood guards and does not share them with the other threads. Since this may affect the performance, we counted the total number of recursions in parallel execution. Perhaps counterintuitively, the parallel execution slightly decreased the number of recursions: the 1- and 72-thread executions produced 38.6 billion and 38.5 billion recursions, respectively. As mentioned above, a parallel search can find search space that is easy to find many embeddings, which leads to fewer recursions. Compared with this phenomenon, the thread-local maintenance of nogood guards has only an unobservable impact, so pruning with guards preserves effectiveness in parallel search.

5 CONCLUSION

We proposed GuP, an efficient algorithm for subgraph matching. GuP utilizes guards on candidate vertices and candidate edges to filter them out adaptively to a partial embedding. Our contributions are (i) a pruning approach based on guards, (ii) the propagation of the injectivity constraint by a reservation, (iii) the nogood discovery rules for effective pruning, and (iv) search-node encoding of a nogood guard. The experimental results showed that GuP can answer difficult queries that the state-of-the-art methods could not answer within the time limit and also can answer other queries in comparable processing time.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Transactions on Database Systems* 42 (2017), 1–44.

[2] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. *Proceedings of the 2019 International Conference on Management of Data*, 1447–1462.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. *Proceedings of the 2016 International Conference on Management of Data* 1, 1199–1214.

[4] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. 2010. Enhancing Graph Database Indexing by Suffix Tree Structure. *Proceedings of the 5th IAPR International Conference on Pattern Recognition in Bioinformatics*, 195–203.

[5] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14 (2013), S13.

[6] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. 2007. Fg-index: Towards Verification-free Query Processing on Graph Databases. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 857–872.
[7] Fan Chung. 2010. Graph theory in the information age. *Notices of the American Mathematical Society* 57 (2010), 726–732.
[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26 (2004), 1367–1372.
[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.).
[10] Eugene C. Freuder and Richard J. Wallace. 1995. Generalizing Inconsistency Learning for Constraint Satisfaction. *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, 563–569.
[11] J. Gaschnig. 1978. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. *Proceedings of the Canadian Artificial Intelligence Conference* (1978), 268–277.
[12] Matthew L. Ginsberg. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence Research* 1 (1993), 25–46.
[13] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. 2013. GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures. *PLoS One* 8 (2013), 1–11.
[14] 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. *Proceedings of the 2019 International Conference on Management of Data*, 1429–1446.
[15] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. TurboISO: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 337–348.
[16] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 405–418.
[17] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *Proceedings of the VLDB Endowment* 4 (2011), 1123–1134.
[18] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. 2000. Maintaining Arc-Consistency within Dynamic Backtracking, Rina Dechter (Ed.). *Principles and Practice of Constraint Programming – CP 2000*, 249–261.
[19] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms. *Proceedings of 20th International Conference on Extending Database Technology*, 25–36.
[20] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2022. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB Journal* 32 (2022), 343–368.
[21] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. *Proceedings of the 2016 International Conference on Management of Data*, 1231–1245.
[22] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming Subgraph Isomorphism for RDF Query Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1238–1249.
[23] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proceedings of the VLDB Endowment* 8 (2015), 974–985.
[24] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment* 10 (2016), 217–228.
[25] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment* 6 (2012), 133–144.
[26] Ciaran McCreesh and Patrick Prosser. 2015. A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs, Gilles Pesant (Ed.). *Principles and Practice of Constraint Programming*, 295–312.
[27] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2020. The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants, Fabio Gadducci and Timo Kehrer (Eds.). *Graph Transformation*, 316–324.
[28] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12 (2019), 1692–1704.
[29] Krzysztof Michalak and Jerzy Korczak. 2011. Graph mining approach to suspicious transaction detection. *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 69–75.
[30] Patrick Prosser. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence* 9 (1993), 268–299.
[31] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 11 (2018), 1876–1888.

[32] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming*.

[33] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment* 1 (2008), 364–375.

[34] Richard M. Stallman and Gerald J. Sussman. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9 (1977), 135–196.

[35] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1083–1098.

[36] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching with Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* 34 (2022), 491–505.

[37] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proceedings of the VLDB Endowment* 14 (2020), 176–188.

[38] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23 (1976), 31–42.

[39] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A Fast and Compact System for Large Scale RDF Data. *Proceedings of the VLDB Endowment* 6 (2013), 517–528.

[40] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment* 3 (2010), 340–351.

Received October 2022; revised January 2023; accepted February 2023