MatCo: Computing Match Cover of Subgraph Query over Graph Data

ZHICHAO SHI, College of Cyber Science and Technology, Hunan University, China
YOUHUAN LI*, College of Cyber Science and Technology, Hunan University, China
ZIMING LI, College of Computer Science and Electronic Engineering, Hunan University, China
YUEQUN DOU, College of Cyber Science and Technology, Hunan University, China
XIONGHU ZHONG, College of Computer Science and Electronic Engineering, Hunan University, China
LEI ZOU, Wangxuan Institute of Computer Technology, Peking University, China

Subgraph query can be applied in various scenarios, such as fraud detection and cyberattack pattern analysis. However, computing subgraph queries usually traverses a huge search space. Many efforts have been made to reduce this search space. The size of the answer set can be exponential, providing a substantial lower bound for the search space. Additionally, different answers may overlap, and a single vertex can occur multiple times in different matches. In this paper, we propose a new problem to compute the match cover of a subgraph query. We define the match cover as a subset of answers such that the vertices included are exactly the same as those in the entire set. There can be more than one match covers, however, we only return one, as long as we can avoid the huge overhead of searching the entire set. It is inefficient to apply traditional subgraph query methods for computing match cover. Specifically, existing methods do not prune partial matches that could grow into full matches. For match cover computation, if the vertices in those full matches are already included in previously found matches, continuing the computation over such partial matches is a waste of time. We propose a new framework, called MatCo, to compute the match cover. In MatCo, we design a new data structure, called local candidate space, to determine whether the future search scopes of partial matches have been covered. We can easily maintain local candidate space and efficiently conduct the determination. We also reduce some Cartesian products, which are inevitable in existing methods, into linear enumerations, which significantly improves performance. Extensive experiments over various datasets confirm that our method outperforms comparative ones by 1~3 orders of magnitude. Efficiently computing the minimum match cover could be an interesting future work.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Subgraph Matching, Match Cover, Partial Match Pruning

*Corresponding author: Youhuan Li

Authors' Contact Information: Zhichao Shi, College of Cyber Science and Technology, Hunan University, Changsha, China, shizhichao@hnu.edu.cn; Youhuan Li, College of Cyber Science and Technology, Hunan University, Changsha, China, liyouhuan@hnu.edu.cn; Ziming Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, zimingli@hnu.edu.cn; Yuequn Dou, College of Cyber Science and Technology, Hunan University, Changsha, China, douyuequn@hnu.edu.cn; Xionghu Zhong, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, xzhong@hnu.edu.cn; Lei Zou, Wangxuan Institute of Computer Technology, Peking University, Beijing, China, zoulei@pku.edu.cn.

ACM 2836-6573/2025/6-ART186
https://doi.org/10.1145/3725323

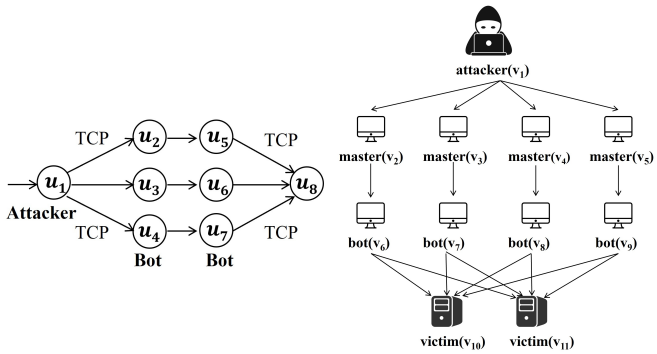
ACM Reference Format:
Zhichao Shi, Youhuan Li, Ziming Li, Yuequn Dou, Xionghu Zhong, and Lei Zou. 2025. MatCo: Computing Match Cover of Subgraph Query over Graph Data. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 186 (June 2025), 24 pages. https://doi.org/10.1145/3725323

1 Introduction

Subgraph query [39] aims to find all matches (isomorphisms) of a query graph in a data graph. As one of the most important problems in graph computation, subgraph queries can be applied in various scenarios, such as detecting financial fraud cycles in transaction graphs [26] and searching for cyberattack patterns in network traffic [20]. Existing works on subgraph queries tend to recursively extend smaller partial matches into larger ones until the final answers are obtained. They typically design index-based filtering and pruning strategies to discard unpromising partial matches as early as possible. However, the answer set can be of exponential size, and since each match corresponds to an independent search branch, many approaches still suffer from the large search space.

In this paper, we propose a new problem: returning a subset of the matches for a subgraph query, such that the vertices in this subset are exactly the same as those in the full set. We call this subset as the match cover of the query over data graph. Specifically, for each data vertex v that exists in some match[1], there must be a match in the cover set containing v. The following two examples demonstrate that, in certain scenarios, returning a match cover instead of the full set may be sufficient.

Example 1: Cyberattack pattern. Fig. 1(a) demonstrates a representative cyberattack pattern [11], where an attacker launches a DDoS attack on a victim machine. Fig. 1(b) depicts the corresponding communication graph formed by DDoS [33] traffic. The full set of 8 matches (i.e., $\{g_1, g_2, \cdots, g_8\}$) is presented in Fig. 1(c), and these matches substantially overlap with each other. In fact, we only need to return a subset of these matches such that for any attack, bot, or victim, it is guaranteed to be included in one of the returned matches. For example, the match cover $\{g_1, g_2\}$, with only 2 matches, includes all related vertices, and for each attacker/bot, there exists at least one match in $\{g_1, g_2\}$ as evidence to indicate its role in the DDoS attack.



(a) Cyberattack pattern (b) DDoS traffic

Full match:

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
g_1	v_1	v_2	v_3	v_4	v_6	v_7	v_8	v_{10}
g_2	v_1	v_2	v_3	v_5	v_6	v_7	v_9	v_{10}
g_3	v_1	v_2	v_4	v_5	v_6	v_8	v_9	v_{10}
g_4	v_1	v_3	v_4	v_5	v_7	v_8	v_9	v_{10}
g_5	v_1	v_2	v_3	v_4	v_6	v_7	v_8	v_{11}
g_6	v_1	v_2	v_3	v_5	v_6	v_7	v_9	v_{11}
g_7	v_1	v_2	v_4	v_5	v_6	v_8	v_9	v_{11}
g_8	v_1	v_3	v_4	v_5	v_7	v_8	v_9	v_{11}

Match cover:

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
g_1	v_1	v_2	v_3	v_4	v_6	v_7	v_8	v_{10}
g_2	v_1	v_3	v_4	v_5	v_7	v_8	v_9	v_{11}

(c) Full match and match cover

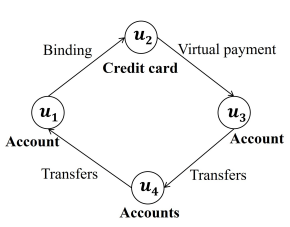
Fig. 1. Query example in cyberattack pattern

Example 2: Financial fraud pattern. Fig. 2(a) presents a credit card fraud pattern involving a fraudulent transaction. Specifically, a criminal (u_1) pays a merchant (u_3) with a credit card (u_2), and the merchant then transfers the money (from a bank) to the criminal through a middleman

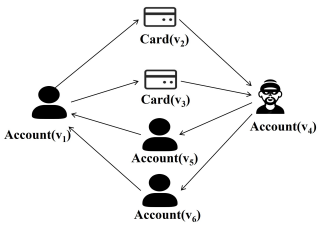
[1]This match may not be included in the match cover

(u_4), completing the illegal cash-out. Fig. 2(b) presents a transaction graph involving this fraud, with multiple credit cards and middlemen[2]. From Fig. 2(c), we see that the match cover $\{g_1, g_2\}$ has 2 matches, while the full set has 4. For each related vertex, there is always a match in $\{g_1, g_2\}$, as evidence, to identify its fraudulent involvement.

It is important to consider whether there is a significant performance benefit to returning a match cover instead of the full set. We have found that a match cover can be much smaller than the full set. We collected statistics on various data graphs from a recent subgraph match survey [35] to compare the size of match covers against full sets. In each graph, we retrieved 50 query graphs by random walk and applied a traditional subgraph match algorithm [21] to compute the full set. Each time we obtained a full answer set for a query, we also computed the minimum k such that the first k obtained matches during the search constitute a match cover. Table 1 presents the size comparison between each full set and the corresponding match cover. We use $|A|$ to denote the average size of the full answer set and $|A'|$ for the match cover size. We also use $|V(A)|$ to denote the number of vertices in A (Clearly, $|V(A)| = |V(A')|$). We observe that $|A|$ is usually larger than $|A'|$ by 5~8 orders of magnitude. Thus, there are many redundant matches with respect to the match cover, and it is promising to design new pruning strategies to reduce redundant computation for performance improvement.



(a) Credit card fraud pattern



(b) Transaction graph

Full match:

	u_1	u_2	u_3	u_4
g_1	v_1	v_2	v_4	v_5
g_2	v_1	v_2	v_4	v_6
g_3	v_1	v_3	v_4	v_5
g_4	v_1	v_3	v_4	v_6

Match cover:

	u_1	u_2	u_3	u_4
g_1	v_1	v_2	v_4	v_5
g_2	v_1	v_3	v_4	v_6

(c) Full match and match cover

Fig. 2. Query example in cyberattack pattern

Table 1. Match cover statistics for different datasets

Dataset	**Answer number** $\|A\|$	**Cover size** $\|A'\|$	$\|A\|/\|A'\|$	$\|V(A)\|$
Yeast	1.72×10^8	1.43×10^2	1.21×10^6	166
Human	5.92×10^{10}	1.18×10^2	5.03×10^8	131
Wordnet	1.72×10^{11}	3.53×10^4	4.86×10^6	3.83×10^4
DBLP	2.08×10^9	2.03×10^3	1.01×10^6	3.07×10^3
YouTube	3.37×10^{11}	1.52×10^4	2.22×10^7	1.62×10^4
EU-2005	4.00×10^{11}	3.90×10^4	1.03×10^7	4.20×10^4
Orkut	1.90×10^{11}	2.28×10^5	8.35×10^5	2.67×10^5
Twitter	3.80×10^{11}	6.68×10^3	5.68×10^7	7.14×10^3

Note that there can be more than one match covers for a query over a data graph. Our goal is to return one of them, as long as the time cost is significantly less than that for the full set of answers. Efficiently computing the minimum match cover could be an interesting direction for future work.

It is non-trivial to extend existing subgraph search methods to match cover computation. Existing methods typically compute subgraph queries by recursively extending small partial matches into

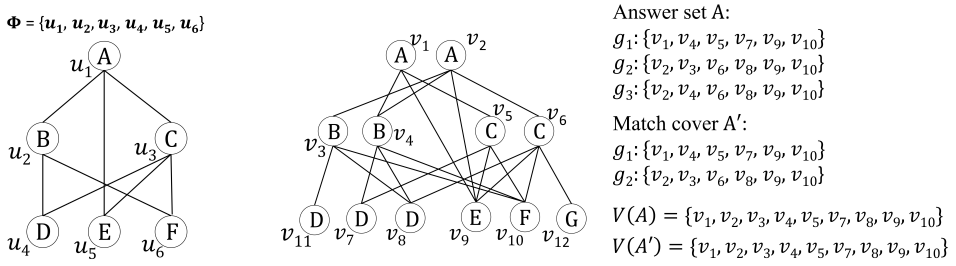
[2]In fraudulent transactions, it is common to have an indefinite number of credit cards and intermediaries [14]

larger ones until final matches are obtained. If a partial match can be extended into one or more full matches, then it is considered promising and cannot be pruned in traditional subgraph search solutions. However, for match cover computation, the situation changes. For example, vertices in those full matches may have already been included in previously found matches. In this case, those full matches are redundant with respect to the current match cover, and the corresponding computation is a waste of time. Note that matches in a match cover may still overlap with each other. Therefore, during the search, even if a vertex v has been included in an existing match, we may still need to visit v in subsequent computations.

We propose a new framework, called MatCo, to efficiently compute the match cover. In MatCo, we design a data structure, called local candidate space, to indicate the future search scope of each partial match. One one hand, if the future search scope indicates empty result, then current partial match can be safely pruned; on the other hand, if the partial match, together with the search scope, has been already covered, then the subsequent computation following this partial match would only lead to redundant matches, and we can prune this search branch immediately. We can easily maintain local candidate space and efficiently perform each determination. And, the space cost for local candidate space is almost constant. Furthermore, we reduce some inevitable Cartesian products in traditional subgraph search into linear enumeration when computing the match cover, which significantly improves performance.

To summarize our contributions:

- We propose a new problem: computing the match cover for a subgraph query. We are the first to define the match cover and reveal, through statistical analysis, that a match cover is usually much smaller than the full answer set.
- We design a framework called MatCo to compute the match cover. MatCo dynamically maintains a new data structure, called local candidate space, for determining unpromising partial matches as early as possible, including those that only lead to redundant matches with respect to the current match cover.
- We reduce some inevitable Cartesian products in traditional subgraph search into linear enumerations for match cover computation, which significantly improves performance.
- Extensive experiments over various datasets confirm the efficiency of our method, which outperforms comparative methods by 1∼3 orders of magnitude.



(a) Query graph (b) Data graph (c) Answer set, match cover and key vertices

Fig. 3. Example of data graph and query graph

2 Problem Definition

Let's formally define our problem to compute match cover for a subgraph query over a data graph.

Definition 1 (Graph). *A graph G consists of a vertex set $V(G)$, an edge set $E(G)$ and a label function L_G that assigns labels for each vertex and edge. We may use V, E and L to denote the corresponding vertex set, edge set and label function, respectively, when the context is clear. For simplicity, we use $L(u)$ and $L(u,v)$ to denote the label of vertex u and that of edge (u,v), respectively. We use $N_G(v)$ to denote the neighbor set of v in G.*

Definition 2 (Subgraph Match). *Consider a data graph G and a query graph Q. A subgraph g of G is called a match (isomorphism) of Q if and only if there exists a bijective mapping F ($\subset V(Q) \times V(g)$) from $V(Q)$ to $V(g)$ such that:*

- $\forall (u,v) \in F, L_Q(u) = L_G(v)$
- $\forall (u_1,v_1),(u_2,v_2) \in F$, *then:*
  - $(u_1,u_2) \in E(Q) \Leftrightarrow (v_1,v_2) \in E(g)$
  - $(u_1,u_2) \in E(Q) \Rightarrow L_Q(u_1,u_2) = L_G(v_1,v_2)$

A data vertex v matches a query vertex u if and only if $L_G(v) = L_Q(u)$, while a data edge (v_1,v_2) matches a query edge (u_1,u_2) if and only if v_1 (v_2, resp.) matches u_1 (u_2, resp.) and $L_G(v_1,v_2) = L_Q(u_1,u_2)$.

For the sake of presentation only, we assume that graphs are undirected. Actually, our method can be easily applied over directed ones, since we can trivially incorporate edge direction constraints into edge matching semantic during query execution.

Definition 3 (Answer Set & Key Vertex). *Consider a data graph G and a query graph Q. We denote answer set of Q over G as $A_{G,Q}$, and for simplicity, when the context is clear, we may sometimes use A in place of $A_{G,Q}$. Additionally, a vertex $v \in V(G)$ is referred to as a key vertex if there exists a match $g \in A$ such that $v \in V(g)$. The set of all key vertices under Q and G is denoted as $V(A)$, namely:*

$$V(A) = \bigcup\nolimits_{g \in A} V(g)$$

The core concept in our problem is the match cover.

Definition 4 (Match Cover). *Consider a data graph G, a query graph Q, and the corresponding answer set A. We define match cover of Q over G as a subset $A' \subseteq A$ such that the set of vertices in A' exactly matches the set of all key vertices, i.e., $V(A') = V(A)$.*

For example, consider the running example in Fig. 3. The set A contains all answers, i.e., $\{g_1, g_2, g_3\}$, and the key vertex set $V(A)$ is $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. We can see that $A' = \{g_1, g_2\} \subset A$ is a match cover since $V(A') = V(A)$.

Clearly, there could be more than one match covers for Q, while our problem is to find one of them. Computing minimum match cover is not our focus, since it may cause extra time cost to make the result set being minimal. Actually, our target is the efficiency, as long as the results cover all key vertices. Efficiently computing minimum match cover could be an interesting future work. Since A' is a subset of A, then $|A'| \leq |A|$. Also, there may be only one match cover, i.e., the full set. For example, it is possible there were no overlap between different matches, and only the full set could cover all key vertices. In this way, $A' = A$. Also, since A' covers all key vertex and each match contains $|V(Q)|$ vertices, in this way, $|V(A)| \leq |A'| \cdot |V(Q)|$, namely, $|A'| \geq \frac{|V(A)|}{|V(Q)|}$. Therefore:

$$\left\lceil \frac{|V(A)|}{|V(Q)|} \right\rceil \leq |A'| \leq |A| \tag{1}$$

Our problem is to efficiently find a match cover, which is NP-hard as indicated in the following Theorem 1.

Theorem 1. *Given a data graph G and a query Q, computing a match cover is NP-hard.*

Proof. We can reduce the k-clique problem (NP-Complete) to match cover computation in polynomial time. Given an arbitrary parameter k and a data graph $G = (V_G, E_G)$, consider the problem to determine whether there exists a clique of size k in G. Let's construct a match cover problem based on this k-clique one. We build a query graph $Q = (V_Q, E_Q, L)$ where $|V_Q| = k$, and each vertex in V_Q has edge with any other vertex, namely,

$$E_Q = \{(u_i, u_j) \mid u_i, u_j \in V_Q \ \wedge u_i \neq u_j\}$$

And, L is a constant label function such that $\forall u \in V_Q$, $L(u) = l$ where l is a constant label. We also build a new graph $G_{new} = (V_{new}, E_{new}, L_{new})$ where $V_{new} = V_G$, $E_{new} = E_G$, and $\forall v \in V_{new}$, $L_{new}(v) = l$. For this match cover problem, if the returned match cover is not empty set, then each returned match is exactly a k-clique, which means there exists a k-clique in the original G since G and G_{new} have the same topology structure ($V_{new} = V_G$, $E_{new} = E_G$). Conversely, there would be no such k-clique matching Q in G_{new} and hence there would be no k-clique in G. Apparently, the problem reduction costs only polynomial time. Overall, match cover computation is NP-hard. □

We also define some concepts that would be frequently used when presenting our method.

Definition 5 (Matching Order and Left/Right Neighbor). *A matching order of a query Q, denoted as Φ_Q is a sequence of all query vertices: $\{u_1, u_2, \cdots, u_{|V(Q)|}\}$. For u_i, u_j in Φ_Q, if u_i is a neighbor of u_j in Q and $i < j$, then u_i is referred to as a left neighbor of u_j under Φ_Q, and, u_j is a right neighbor of u_i. We use $LN_{\Phi_Q}(u_i)$ ($RN_{\Phi_Q}(u_i)$, resp.) to denote the left (right, resp.) neighbor sets of u_i under Φ_Q.*

We use q^i to denote the subquery induced by the first i vertices in Φ_Q. For the sake of presentation only, we may use Φ, $LN(u_i)$ and $RN(u_i)$ for Φ_Q, $LN_{\Phi_Q}(u_i)$ and $RN_{\Phi_Q}(u_i)$, respectively, when the context is clear. For example, in Fig. 3(a), left neighbors of u_4 are $\{u_2, u_3\}$. Also, u_1 is the only neighbor at the left of u_3, and hence $LN(u_3) = \{u_1\}$. Note that we do not discuss how to design a matching order since it is a well-studied problem [5], and our method simply builds a matching order that is consistent with the deceasing order of query vertex degree.

Definition 6 (i-match). *Consider a query Q and a matching order $\Phi = \{u_1, u_2, \cdots, u_{|V(Q)|}\}$, we may use a data vertex sequence $\{v_1, v_2, \cdots, v_{|V(Q)|}\}$ to denote a match if the pair-wise mapping $\{(u_1, v_1), (u_2, v_2), \cdots, (u_{|V(Q)|}, v_{|V(Q)|})\}$ is a bijective one satisfying the subgraph matching constraints in Definition 2. Similarly, the i-length prefix $\{v_1, v_2, \cdots, v_i\}$ $(1 \leq i \leq |V(Q)|)$ could be used to denote the partial match matching q^i. We call this i-length sequence as a i-match of Q under Φ, which is usually denoted as g^i.*

For example, in Fig. 3, with $\Phi = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, we use g_1=$\{v_1, v_4, v_5, v_7, v_9, v_{10}\}$ to denote a match, and $\{v_1\}$, $\{v_1, v_4\}$ and $\{v_1, v_4, v_5\}$ represent 1-match, 2-match and 3-match, respectively.

Definition 7 (Dependent Vertex). *For a data graph G, query Q and $\Phi = \{u_1, u_2, \cdots, u_k\}$ $(k = |V(Q)|)$, consider a partial match g^{i-1} matching q^{i-1}. For a left neighbor $u_{i'}$ of u_i $(i' < i)$, the data vertex $v_{i'} \in g^{i-1}$ matching $u_{i'}$ is a dependent vertex of u_i over g^{i-1}.*

For example, consider a partial match $g^3 = \{v_1, v_4, v_5\}$ of the running example in Fig. 3. We know that the left neighbors of u_4 are $\{u_2, u_3\}$. Since v_4 and v_5 match u_2 and u_3, respectively, then dependent vertices of u_4 over g^3 are $\{v_4, v_5\}$.

Definition 8 (Query Edge Constrained Neighbor Set). *Consider a data graph G, a query Q and a data vertex v matching u. Assume that (u, u') is a query edge in Q, then the query edge constrained neighbor set of v under u', denoted as $N^{(u,u')}(v)$, is defined as the maximum subset of $N_G(v)$ such that for each $v' \in N_G(v)$, data edge (v, v') matches query edge (u, u'), namely,*

$$N^{(u,u')}(v) = \{v' \in N_G(v) \mid (v, v') \text{ matches } (u, u')\}$$

For example, consider a partial match $g^3 = \{v_1, v_4, v_5\}$ of the running example in Fig. 3. We know that g^3 matches $q^3 = \{u_1, u_2, u_3\}$. For query edge (u_2, u_4), adjacent edges of v_4 (matching u_2) that match (u_2, u_4) are (v_4, v_7) and (v_4, v_8), and hence, $N^{(u_2,u_4)}(v_4) = \{v_7, v_8\}$. Similarly, $N^{(u_3,u_4)}(v_5) = \{v_7\}$.

3 Approach

In this section, we present our MatCo framework for computing match cover. We introduce the traditional subgraph search in Section 3.1, where we discuss in detail why existing methods can not be applied for match cover computation. In Section 3.2, we propose a new data structure: local candidate space to compute match cover. We define a future search scope for each partial match with the local candidate space, and once we find that the future search scope has been covered, we can prune the corresponding partial match safely. We also propose an innovative optimization in Section 3.3 to accelerate MatCo, where we reduce time-consuming Cartesian products that are inevitable in traditional method into lightweight linear enumeration. We discuss our overall complexity in Section 3.4.

3.1 Traditional Subgraph Search & Its Inefficiency for Match Cover Computation

Consider a query graph Q, a data graph G and a matching order $\Phi = \{u_1, u_2, \cdots, u_{|V(Q)|}\}$. We firstly illustrate a traditional search process for subgraph query with some frequently-used concepts, which is beneficial to our later presentation for MatCo.

Existing frameworks for subgraph query usually grow each small partial match into full ones by a series extensions. Consider an i-match g^i matching q^i and a data vertex v. If $g^i \cup \{v\}$ matches q^{i+1} (with v matching u_{i+1}), then we say that v is a feasible candidate of u_{i+1} over g^i. We use $P(g^i)$ to denote the set of feasible candidate vertices under Φ. In this way, for each g^i (matching q^i), we need to compute the corresponding $P(g^i)$, based on which we can extend g^i into several g^{i+1} (matching $q^{i+1} = q^i \cup \{u_{i+1}\}$). And then each g^{i+1} would similarly be extended into a series of g^{i+2}. One can recursively conduct such extension to obtain target final answers.

In an extension, one can apply worst-case optimal join (WCOJ) [21] to compute the set of feasible candidates $P(g^i)$, which is similar to many existing works [3, 17, 19, 23, 27, 36]. Specifically, for a partial match $g^i = \{v_1, v_2, \cdots, v_i\}$ matching $q^i = \{u_1, u_2, \cdots, u_i\}$, according the definition of subgraph match, any feasible candidate of u_{i+1} over g^i must be a common neighbor of dependent vertices (see Definition 7) under u_{i+1} in g^i. Assume that $LN(u_{i+1}) = \{u_{i_1}, u_{i_2}, \cdots, u_{i_y}\}$, where $y = |LN(u_{i+1})|$ and each dependent vertex v_{i_x} matches u_{i_x} $(1 \le x \le y)$. According to Definition 8, we know that $N^{(u_{i_x},u_{i+1})}(v_{i_x})$ is the subset of $N_G(v_{i_x})$ $(1 \le x \le y)$, such that for each vertex $v \in N^{(u_{i_x},u_{i+1})}(v_{i_x})$, data edge (v_{i_x}, v) matches query edge (u_{i_x}, u_{i+1}). Then, according to the WCOJ mechanism, we can compute $P(g^i)$ by conducting a series of intersections over all such $N^{(u_{i_x},u_{i+1})}(v_{i_x})$, followed by removing visited vertices (i.e., those in $V(g^i)$) from the result set:

$$P(g^i) = \left(\bigcap_{u_{i_x} \in LN(u_{i+1})} N^{(u_{i_x},u_{i+1})}(v_{i_x}) \right) \setminus V(g^i) \tag{2}$$

It is inefficient to apply existing subgraph matching work to compute match cover, since we may not need to find out the full set of answers to constitute a match cover. An important problem is that, during our search, how to determine current partial match is unpromising? In traditional methods, a partial match is unpromising only if it could not be extended into any full answer. While, things are different in match cover computation. Consider a partial match g^i that could be extended into several full matches. Assume that A_{g^i} is the set of full matches that are grown from g^i $(A_{g^i} \neq \emptyset)$, and A'' is the set of matches already found at the first sight of g^i during the search. Then,

g^i could still be unpromising under our problem if $V(A_{g^i}) \subseteq V(A'')$, i.e., subsequent search over g^i would not contribute to covering new key vertices (see Definition 3). None of previous work could determine such unpromising partial match, since they have no way to figure out $V(A_{g^i})$ in advance, let alone to determine whether $V(A_{g^i})$ has been covered or not. Hence, existing algorithms would not terminate unless they figure out the full set of matches, which is inefficient.

For example, consider the data graph G, query graph Q and the corresponding Φ in Fig. 3. With the approach indicated in Algorithms 1 and 2, when we encounter the partial match $g^2 = \{v_2, v_4\}$, two full matches g_1 and g_2 (see Fig. 3) have been found and the covered vertex set is $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. In this way, the third full match g_3, the only one that would be grown from g^2, would only contain vertices that have already been covered. Hence, computation for g_3 is just a waste of time and the corresponding partial match g^2 should be pruned once encountered. In the following Section 3.2, we propose an auxiliary data structure, called local candidate space, to effectively determine such special unpromising partial match.

3.2 Local Candidate Space for MatCo

We propose a new data structure, i.e., local candidate space, together with an innovative pruning strategies to efficiently compute match cover. Note that this is one of the two important parts in MatCo, and the other novel optimization for MatCo will be illustrated in the subsequent Section 3.3. We will finally present the complexity analysis of our method in Section 3.4.

For each partial match g^i, the corresponding local candidate space is essentially an array, where each element is a set of candidate for a query vertex, which limits the search range over g^i. We formally define local candidate space in the following Definition 9.

Definition 9 (Local Candidate Space). *Consider a partial match $g^i = \{v_1, v_2, \cdots, v_i\}$ matching $q^i = \{u_1, u_2, \cdots, u_i\}$ under Φ. The local candidate space of g^i, denoted as C_{g^i}, is an array of size $|V(Q)|$. Each element $C_{g^i}[j]$ is a set of candidates matching u_j $(1 \le j \le |V(Q)|)$, which is defined as follows:*

- *if $j \le i$, $C_{g^i}[j] = \{v_j\}$.*
- *if $j > i$:*
  - *if there is no dependent vertex in g^i for u_j, (i.e., $LN(u_j) \cap V(q^i) = \emptyset$), then $C_{g^i}[j]$ is marked as undefined, i.e., $C_{g^i}[j] = null$. Note that $null \neq \emptyset$;*
  - *if there exists a dependent vertex in g^i for u_j, (i.e., $LN(u_j) \cap V(q^i) \neq \emptyset$), then $C_{g^i}[j]$ can be formed by the following intersections with a duplication to remove visited vertices:*

$$C_{g^i}[j] = \left(\bigcap_{u_{i'} \in LN(u_j) \cap V(q^i)} N^{(u_{i'}, u_j)}(v_{i'}) \right) \setminus V(g^i) \tag{3}$$

where $v_{i'}$ (matching $u_{i'}$) is a dependent vertex in g^i for u_j.

With Definition 9, we have the following important Lemma 1 and Theorem 2.

Lemma 1. *For a partial match g^i, $P(g^i) = C_{g^i}[i+1]$.*

Proof. Since we know that each prefix of matching order always constitutes a connected subquery (to avoid unnecessary Cartesian products), and hence, u_{i+1} would always have at least one left neighbor in q^i, and meanwhile, there would be at least one dependent vertex in g^i of u_{i+1}. In this way, $C_{g^i}[i+1]$ would not be *null* and hence $C_{g^i}[i+1]$ can be computed according to Equation 3. Also, when $j = i+1$, then $LN(u_j) \cap V(q^i)$ is exactly equivalent to $LN(u_{i+1})$, therefore, definition of $P(g^i)$ (see Equation 2) is the same as that of $C_{g^i}[i+1]$ in Equation 3. □

Theorem 2. *For two partial matches $g^{i'}$ and g^i, where g^i is grown from $g^{i'}$ ($i' < i$). Assume that $g^{i'} = \{v_1, v_2, \cdots, v_{i'}\}$ matching $q^{i'}$, and $g^i = \{v_1, v_2, \cdots, v_{i'}, v_{i'+1}, \cdots, v_i\}$ matching q^i. Then, for $1 \le j \le |V(Q)|$, if $C_{g^i}[j]$ is not null, we can conclude that $C_{g^i}[j] \subseteq C_{g^{i'}}[j]$.*

Proof. We can prove this theorem as follows:

- ① if $1 \le j \le i'$, apparently,
$$C_{g^i}[j] = \{v_j\} = C_{g^{i'}}[j]$$
- ② if $j > i$,
  - if dependent vertices of u_j in $g^{i'}$ are the same as those in g^i (i.e., $LN(u_j) \cap V(q^{i'}) = LN(u_j) \cap V(q^i)$), then
$$C_{g^i}[j] = C_{g^{i'}}[j] \setminus V(g^i)$$
  - if the set of dependent vertices of u_j in $g^{i'}$ is different from that in g^i (i.e., $LN(u_j) \cap V(q^{i'}) \neq LN(u_j) \cap V(q^i)$), assume that $LN(u_j) \cap (V(q^i) \setminus V(q^{i'}) = \{u_{i_1}, u_{i_2}, \cdots, u_{i_y}\}$, and the corresponding dependent vertices are $\{v_{i_1}, v_{i_2}, \cdots, v_{i_y}\}$ ($i' < i_1 < i_2 < \cdots < i_y \le i$), then, according to the definition of local candidate space, we have:
$$C_{g^i}[j] = C_{g^{i'}}[j] \cap \left(\bigcap_{x=1}^{y} N^{(u_{i_x}, u_j)}(v_{i_x}) \right) \setminus V(g^i)$$
  in this way, $C_{g^i}[j] \subseteq C_{g^{i'}}[j]$;
- ③ if $i' < j \le i$, then according to ②,
$$C_{g^{j-1}}[j] \subseteq C_{g^{i'}}[j]$$
  while, together with Lemma 1, we have:
$$P(g^{j-1}) = C_{g^{j-1}}[j] \subseteq C_{g^{i'}}[j]$$
  since $C_{g^i}[j] = \{v_j\}$, and $v_j \in P(g^{j-1})$ (according to the definition of $P(g^{j-1})$), then $C_{g^i}[j] \subseteq P(g^{j-1}) \subseteq C_{g^{i'}}[j]$.

Above all, for any $C_{g^{i'}}[j]$ ($1 < j \le |V(Q)|$), either $C_{g^{i'}}[j] = \textit{null}$ or $C_{g^i}[j] \subseteq C_{g^{i'}}[j]$. □

$g^2 = \{v_1, v_4\}$

C_{g^2}	$C_{g^2}[1]$	$C_{g^2}[2]$	$C_{g^2}[3]$	$C_{g^2}[4]$	$C_{g^2}[5]$	$C_{g^2}[6]$
Candidate set	$\{v_1\}$	$\{v_4\}$	$\{v_5\}$	$\{v_7, v_8\}$	$\{v_9\}$	$\{v_{10}\}$

$g^3 = \{v_1, v_4, v_5\}$

C_{g^3}	$C_{g^3}[1]$	$C_{g^3}[2]$	$C_{g^3}[3]$	$C_{g^3}[4]$	$C_{g^3}[5]$	$C_{g^3}[6]$
Candidate set	$\{v_1\}$	$\{v_4\}$	$\{v_5\}$	$\{v_7\}$	$\{v_9\}$	$\{v_{10}\}$

Fig. 4. Examples for local candidate space

For instance, consider the running example in Fig. 3 and the examples for local candidate space in Fig. 4. $g^3 = \{v_1, v_4, v_5\}$ is grown from $g^2 = \{v_1, v_4\}$, and we can see from Fig. 4 that for $1 \le j \le |V(Q)| = 6$, $C_{g^3}[j] \subseteq C_{g^2}[j]$.

With Theorem 2, let's discuss our pruning strategy over each partial match g^i. Consider a full match $g = \{v_1, v_2, \cdots, v_{|V(Q)|}\}$ that is grown from $g^i = \{v_1, v_2, \cdots, v_i\}$. According to Theorem 2, for each v_j matching u_j ($i{+}1 \le j \le |V(Q)|$) in g, if $C_{g^i}[j]$ is not null, then v_j must be in $C_{g^i}[j]$. This

motivates us to define a future search scope for each partial match g^i where the scope covers all full matches that g^i could grow into.

Definition 10 (Future Search Scope). *Consider a query Q, data graph G and a matching order $\Phi = \{ u_1, u_2, \cdots, u_{|V(Q)|} \}$. For a partial match g^i, the future search scope, denoted as $Scope(g^i)$, is a set formed by the following Cartesian product:*

$$Scope(g^i) = S_1 \times S_2 \times \cdots S_{|V(Q)|}$$

where for $1 \leq j \leq |V(Q)|$:

$$S_j = \begin{cases} C_{g^i}[j], & if\, C_{g^i}[j] \neq null. \\ V(G), & if\, C_{g^i}[j] = null. \end{cases}$$

According to Theorem 2 and Definition 10, if a full match $g = \{ v_1, v_2, \cdots, v_{|V(Q)|} \}$ is grown from a partial match g^i, then g (i.e., $\{ v_1, v_2, \cdots, v_{|V(Q)|} \}$) must be within $Scope(g^i)$. Hence, $Scope(g^i)$ covers all full matches that g^i could grow into.

One straightforward pruning strategy based on future search scope is that, for each partial match g^i, once there exists a j ($i < j \leq |V(Q)|$) such that $C_{g^i}[j] = \emptyset$ (not *null*), then $Scope(g^i)$ would be empty[3]. In this way, g^i could not grow into any full match and it can be safely pruned.

More importantly, for each partial match g^i, we can utilize local candidate space of g^i to test whether the future search scope $Scope(g^i)$ has already been covered. Specifically, if for $\forall 1 \leq j \leq |V(Q)|$, $|C_{g^i}[j]| > 0$ and every vertex $v \in C_{g^i}[j]$ has been covered[4], then all vertices in $Scope(g^i)$ accordingly had already been covered, including those in any possible full match g that would be grown from g^i. Therefore, for current match cover computation, each such full match g should not be returned since it contributes nothing to covering new key vertices, and accordingly, we need to prune g^i immediately to save computation. Apparently, the determination on whether $Scope(g^i)$ has been covered need only one linear scan over local candidate space, which is efficient since the number of vertices in a local candidate space is $O(|V(Q)| * d_G)$ and d_G is the average degree of the data graph G.

Found matches : $g_1 = \{v_1, v_4, v_5, v_7, v_9, v_{10}\}$
$g_2 = \{v_2, v_3, v_6, v_8, v_9, v_{10}\}$

Covered vertices : $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$

$g^2 = \{v_2, v_4\}$

C_{g^2}	$C_{g^2}[1]$	$C_{g^2}[2]$	$C_{g^2}[3]$	$C_{g^2}[4]$	$C_{g^2}[5]$	$C_{g^2}[6]$
Candidate set	$\{v_2\}$	$\{v_4\}$	$\{v_6\}$	$\{v_7, v_8\}$	$\{v_9\}$	$\{v_{10}\}$
Is covered	✓	✓	✓	✓	✓	✓

Fig. 5. Example of covered search scope

For instance, consider the running example in Fig. 3. When we encounter $g^2 = \{v_2, v_4\}$ during the search where two full matches g_1 and g_2 have been obtained and the covered vertex set is $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. Fig. 5 presents the corresponding search scope of g^2, and we can see that every item in C_{g^2} has been covered. Hence, we could safely prune g^2 once it is encountered.

Now let's consider our overall search process, including how to maintain local candidate space when we grow each g^i into g^{i+1}. Algorithm 1, together with the procedure in Algorithm 2, presents the pseudo codes of our search for MatCo.

[3]The corresponding Cartesian products for $Scope(g^i)$ involve a $\emptyset$

[4]Namely, v has been included in some match that is already found during current search.

Algorithm 1: MatCo over local candidate space

Input: A query graph Q and data graph G
Input: $\Phi = \{u_1, \cdots, u_{|V(Q)|}\}$
Output: Report a match cover M
1 **foreach** v_1 *matching* u_1 **do**
2 Let $g^1 = \{v_1\}$ under Φ
3 **foreach** $j \leftarrow 1$ *to* $|V(Q)|$ **do**
4 Initialize $C_{g^1}[j] = null$
5 **if** $j = 1$ **then**
6 $C_{g^1}[j] = \{v_1\}$
7 **if** $u_1 \in RN(u_j)$ **then**
8 $C_{g^1}[j] = N^{(u_1,u_j)}(v_1) \setminus \{v_1\}$
9 Set v_1 as visited
10 Extension$(G, Q, \Phi, g^1, C_{g^1})$
11 /* See Algorithm 2 for Extension procedure */
12 Set v_1 as unvisited
13 Report match cover M
14 **return**

Initially, for each candidate v_1 of u_1, we conduct our search over partial match $g^1 = \{v_1\}$ (Line 2 in Algorithm 1). According to the definition of local candidate space (see Definition 9), we set $C_{g^1}[1]$ as $\{v_1\}$, and for each right neighbor u_j of u_1, v_1 is the only one dependent vertex (see Definition 7) of u_j over g^1, and hence, we also set $C_{g^1}[j]$ as $N^{(u_1,u_j)}(v_1) \setminus \{v_1\}$ according to Equation 3 (Lines 3-8 in Algorithm 1).

We can easily conduct extensions with local candidate space. According to Lemma 1, for each g^i, we can just take $C_{g^i}[i+1]$ as the feasible candidate set $P(g^i)$. Therefore, for each $v_{i+1} \in C_{g^i}[i+1]$, we would obtain a $g^{i+1} = g^i \cup \{v_{i+1}\}$, which would be recursively extended into a series of g^{i+2} (Lines 11-12 in Algorithm 2).

Let's discuss how to construct $C_{g^{i+1}}$ based on C_{g^i} when we grow $g^i = \{v_1, v_2, \cdots, v_i\}$ into $g^{i+1} = g^i \cup \{v_{i+1}\}$. According to the definition of local candidate space, $\forall 1 \le j \le i$, $C_{g^{i+1}}[j] = C_{g^i}[j] = \{v_j\}$. Also, we need to set $C_{g^{i+1}}[i+1]$ as $\{v_{i+1}\}$. Now let's consider $C_{g^{i+1}}[j]$ where $i+1 < j \le |V(Q)|$:

- if u_j is not a neighbor of u_{i+1}, then dependent vertices (see Definition 7) of u_j over g^{i+1} are exactly the same as those over g^i, and hence $C_{g^{i+1}}[j] = C_{g^i}[j] \setminus \{v_{i+1}\}$;
- if u_j is a neighbor of u_{i+1} and $C_{g^i}[j] = null$, then u_{i+1} must be the only mapped left neighbor of u_j, and v_{i+1} is the only dependent vertex of u_j over g^{i+1}, hence, we need to set $C_{g^{i+1}}[j] = N^{(u_{i+1},u_j)}(v_{i+1}) \setminus V(g^i)$ (Line 17 in Algorithm 2);
- if u_j is a neighbor of u_{i+1} and $C_{g^i}[j] \neq null$, then v_{i+1} is a new dependent vertex of u_j over g^{i+1}, and hence we need to set $C_{g^{i+1}}[j] = C_{g^i}[j] \cap N^{(u_{i+1},u_j)}(v_{i+1}) \setminus V(g^i)$ (Line 19 in Algorithm 2).

Theorem 3. *Given a data graph G, query graph Q and a matching order $\Phi = \{u_1, u_2, \cdots, u_{|V(Q)|}\}$, our method (Algorithms 1 and 2) could successfully output a match cover of Q over G.*

Proof. We prove this theorem in following two steps.

- ① Each output $g^{|V(Q)|}$ is a match of Q (Lines 2-3 in Algorithm 2). Specifically, our search starts with each g^1 that matches $q^1 = \{u_1\}$ (Lines 1-2 in Algorithm 1). And then we recursively extend each partial match g^i into several g^{i+1} with feasible candidates in $C_{g^i}[i+1]$ (see Lemma

1), where each g^{i+1} exactly matches q^{i+1} according to the definition of feasible candidate set. Hence, each output $g^{|V(Q)|}$ is a match of $q^{|V(Q)|}$, namely, Q.

- ② All output matches of our algorithm constitute a match cover. Let A denote the full set of matches and A_{output} is the output set of ours. If $A \setminus A_{output} = \emptyset$, then A_{output} is a match cover. Let's consider the case when $A \setminus A_{output} \neq \emptyset$.
  - According to ①, A_{output} is a subset of A, i.e., $A_{output} \subseteq A$. Thus, $V(A_{output}) \subseteq V(A)$.
  - Assume that a match $g \in A \setminus A_{output}$. According to Algorithm 2, g is not in A_{output} only if there exists a partial match $g^i \subseteq g$ such that $Scope(g^i)$ has already been covered when our search encounters g^i (Lines 8-10 in Algorithm 2). Hence, $V(g)$ has been covered at that time, which means $V(g) \subseteq V(A_{output})$. In this way, vertices of matches in $A \setminus A_{output}$ are covered by those in A_{output}, thus, $V(A \setminus A_{output}) \subseteq V(A_{output})$. Since $V(A_{output}) \subseteq V(A_{output})$, then:

$$V(A) = \big(V(A \setminus A_{output}) \cup V(A_{output})\big) \subseteq V(A_{output})$$

  namely, $V(A) \subseteq V(A_{output})$.

Hence, $V(A_{output}) = V(A)$, and A_{output} is a match cover.

□

Algorithm 2: Extension for MatCo

```
1  Procedure Extension(G, Q, Φ, g^i, C_{g^i})
2      if |g^i| = |V(Q)| then
3          Add g^i into M
4          Mark each vertex in V(g^i) covered
5          return
6      if There exists one element in C_{g^i} is ∅ then
7          return                                   ▷ Scope(g^i) must be empty
8      if None of elements in C_{g^i} is null then
9          if Every vertex in ⋃_{j=1}^{|V(Q)|} C_{g^i}[j] has been covered then
10             return                               ▷ Scope(g^i) must be already covered
11     foreach v_{i+1} ∈ C_{g^i}[i + 1] do
12         Let g^{i+1} = g^i ∪ {v_{i+1}}
13         Initialize C_{g^{i+1}} with C_{g^i}
14         Set C_{g^{i+1}}[i + 1] = {v_{i+1}}
15         foreach u_j ∈ RN(u_{i+1}) do
16             if C_{g^{i+1}}[j] = null then
17                 C_{g^{i+1}}[j] = N^{(u_{i+1},u_j)}(v_{i+1}) \V(g^i)
18             else
19                 C_{g^{i+1}}[j] = C_{g^i}[j] ∩ N^{(u_{i+1},u_j)}(v_{i+1}) \V(g^i)
20         Set v_{i+1} as visited and remove v_{i+1} (if exists) from C_{g^{i+1}}
21         Extension(G, Q, Φ, g^{i+1}, C_{g^{i+1}})
22         Set v_{i+1} as unvisited
23     return
```

Our search for MatCo is efficient. For each partial match g^i, we need to scan the entire local candidate space C_{g^i}, to determine whether $Scope(g^i)$ has been covered. The first i elements in C_{g^i} contain totally i vertices, and each of the remaining $|V(Q)| - i$ elements contains $O(d_G)$ vertices

where d_G is the average degree of G. Hence, the scan costs $O(i + (|V(Q)| - i) \cdot d_G)$ time, namely, $O((|V(Q)| - i) \cdot d_G)$, which is almost constant. While, once we found the scope has already covered, we would not access search branches over g^i, and the search space we prune may be of the same size as that of $Scope(g^i)$, namely:

$$O(|Scope(g^i)|) = O\left(\left|\bigtimes_{j=1}^{|V(Q)|} C_{g^i}[j]\right|\right) = O(d_G^{|V(Q)|-i})$$

Overall, assume that the probability for $Scope(g^i)$ being covered is p, then the expected time cost we save on g^i is $O(p \cdot d_G^{|V(Q)|-i})$. Since the time we conduct the determination is $O((|V(Q)| - i) \cdot d_G)$, then our strategy is effective if the following inequality holds:

$$\frac{O((|V(Q)| - i) \cdot d_G)}{O(d_G^{|V(Q)|-i})} = \frac{O(|V(Q)| - i)}{O(d_G^{|V(Q)|-i-1})} < p$$

Usually, $(d_G^{|V(Q)|-i-1})$ is far larger than $(|V(Q)| - i)$, and our experiments indicate that p is usually larger than 40%, more details are available in Section 4.

There is also an optimization to reduce the time cost for determining whether scopes are covered. In each determination, we would scan the entire local candidate space, which could be optimized. Recall the Theorem 2 that non-null local candidate sets usually shrink when we grow small partial matches into larger ones. In this way, once some local candidate set $C_{g^i}[j]$ for partial match g^i is covered, then for any g^{i+1} grown from g^i, vertices in $C_{g^{i+1}}[j]$ must be covered ($C_{g^{i+1}}[j] \subseteq C_{g^i}[j]$). Therefore, we can set an extra flag to mark whether a local candidate set has already covered, and this flag would not change until backtrack happens in current search branch. With our search going on, more and more vertices would be covered (Line 4 in Algorithm 2), and the check over these flags would be more frequent, and hence, this acceleration is significant.

In worst case, the match cover returned may contain all matches, since different matches may not overlap with each other.

3.3 Optimization with Linear Enumeration

We propose an acceleration strategy in MatCo to reduce time-consuming Cartesian products into efficient linear enumeration.

Let's discuss the Cartesian products. Consider a matching order Φ of Q. If $u \in \Phi$ has no right neighbor, then u is an isolated vertex [19]. Search over this matching order could possibly result in Cartesian products. For example, consider the case when u_{i+1} is an isolated vertex ($i + 1 < |V(Q)|$). We know that for each partial match g^i, the set of $(i + 1)$-matches that are grown from g^i is $\{g^i\} \times C_{g^i}[i + 1]$. Since there is no edge between u_{i+1} and u_{i+2}, then $(i+1)$-th vertex in g^{i+1} is not a dependent vertex of u_{i+2}, and hence, for each such g^{i+1}, the feasible candidate set, i.e., $C_{g^{i+1}}[i + 2]$, would be usually the same as $C_{g^i}[i + 2]$. Thus, the set of $(i + 2)$-matches that are grown from g^i is nearly $\{g^i\} \times C_{g^i}[i + 1] \times C_{g^i}[i + 2]$, which means Cartesian product happens.

Existing work [6, 17, 19] usually move isolated vertices at the right side of matching order to postpone the Cartesian products. However, these postponements can only delay the increase of search space, while Cartesian products may not be avoided. The following Theorem 4 indicates that these Cartesian products is usually hard to avoided in both existing subgraph matching work and our local candidate space based MatCo search (Section 3.2).

Theorem 4. *Consider G, Q and a matching order $\Phi = \{u_1, u_2, \cdots, u_{|V(Q)|}\}$, where $u_{i+1}, u_{i+2}, \cdots, u_{|V(Q)|}$ are isolated vertices. For a partial match $g^i = \{v_1, v_2, \cdots, v_i\}$, a $|V(Q)|$-tuple $g \in Scope(g^i)$ is a match of Q if there is no duplicated vertex in g.*

Proof. • ① We prove that the following equation holds:

$$Scope(g^i) = \{g^i\} \times C_{g^i}[i+1] \times C_{g^i}[i+2] \cdots \times C_{g^i}[|V(Q)|]$$

Actually, since $u_{i+1}, u_{i+2}, \cdots u_{|V(Q)|}$ are isolated vertices, then these vertices have no edge between each other, and their left neighbors must be within $\{u_1, u_2, \cdots, u_i\}$. Hence, according to the definition of local candidate space, $C_{g^i}[j]$ $(i+1 \le j \le |V(Q)|)$ would not be *null*. Also, according to the definition of future search scope, $Scope(g^i)$ can be exactly formed by the above Cartesian products.

- ② Let's prove a $|V(Q)|$-tuple $g \in Scope(g^i)$ with no duplicated vertex is a match of Q. Assume that $g = \{v_1, v_2, \cdots, v_i, \cdots, v_{|V(Q)|}\}$, then it is easy to know that $L_G(v_j) = L_Q(u_j)$ $(1 \le j \le |V(Q)|)$ since v_j is one of the candidates of u_j. Also, since there is no duplicated vertex in g, then the pair-wise mapping $\{(u_1, v_1), (u_2, v_2), \cdots, (u_{|V(Q)|}, v_{|V(Q)|})\}$ is bijective. Furthermore, $\forall (u_{j_1}, u_{j_2}) \in E_Q$:
  - if $(u_{j_1}, u_{j_2}) \in E_{q^i}$, since g^i matches q^i, then we have $(v_{j_1}, v_{j_2}) \in E_{g^i}$, and $L_Q(u_{j_1}, u_{j_2}) = L_G(v_{j_1}, v_{j_2})$;
  - if $(u_{j_1}, u_{j_2}) \notin E_{q^i}$, since there is no query edge between those in $\{u_{i+1}, u_{i+2}, \cdots, u_{|V(Q)|}\}$, then u_{j_1} must be in q^i while u_{j_2} in $Q \setminus q^i$. According to the definition of $C_{g^i}[j_2]$, v_{j_2} is a neighbor of v_{j_1} and $L_Q(u_{j_1}, u_{j_2}) = L_G(v_{j_1}, v_{j_2})$.

  Thus, recall the definition of subgraph match (see Definition 2), g is a match of Q.

□

With Theorem 4, for each g^i when $\{u_{i+1}, u_{i+2}, \cdots, u_{|V(Q)|}\}$ are isolated vertices, each tuple g in $Scope(g^i)$ tends to be a match of Q, our extension-based search (Algorithm 2) may traverse the entire $Scope(g^i)$ (of exponential scale) to cover vertices in C_{g^i}.

Actually, Theorem 4 also motivates us to replace Cartesian products with linear enumeration, so that we can reduce search space of exponential scale into that of linear one. For each g^i, our enumeration strategy would cover vertices in C_{g^i} with only linear time cost. Specifically, assume that $\{u_{i+1}, u_{i+2}, \cdots, u_{|V(Q)|}\}$ are isolated vertices. Consider $g^i = \{v_1, v_2, \cdots, v_i\}$ and $g \in Scope(g^i)$, where $g = \{v_1, v_2, \cdots, v_i, \cdots, v_{|V(Q)|}\}$ and g has no duplicated vertex. According to Theorem 4, g is a match of Q. We first mark each vertex in g as visited. For each unvisited vertex v in $C_{g^i}[j]$ $(i < j \le |V(Q)|)$, $g \cup \{v\} \setminus \{v_j\}$ is still a tuple in $Scope(g^i)$ that is of no duplicated vertex, and according to Theorem 4, $g \cup \{v\} \setminus \{v_j\}$ is also a match of Q. In this way, we can easily generate $|C_{g^i}[j]|$ matches that cover vertices in $C_{g^i}[j]$ based on g. Each of such match is generated in constant time.

Let's present the detailed process for our linear enumeration. Algorithm 3 presents the corresponding pseudo codes. Assume that $\{u_{i+1}, u_{i+2}, \cdots, u_{|V(Q)|}\}$ are isolated vertices. For each g^i, we firstly retrieve unvisited vertices from $C_{g^i}[j]$ $(i < j \le |V(Q)|)$ to form a match $g^{|V(Q)|}$ of Q (Lines 3-8 in Algorithm 3). We would add g in set M if there exists uncovered vertices in g (Lines 9-11 in Algorithm 3). Then, for each $v \in C_{g^i}[j]$ $(i < j \le |V(Q)|)$ where v is neither visited nor covered, we can replace v with the j-th vertex in $g^{|V(Q)|}$ to form a new match that covers v (Lines 12-17 in Algorithm 3). After this process, all key vertices in C_{g^i} would be covered according to our discussions. We can see that time-consuming Cartesian products are safely reduced into efficient linear enumeration, which could significantly improve the performance of MatCo.

3.4 Complexity Analysis

Let's discuss the overall complexity of our approach. Let d_G and d_Q denote the average degree of data and query graphs, respectively. For space cost, during the search, the only auxiliary data structure we maintain is the local candidate space, which costs only $O(|V(Q)| \cdot d_G)$ space. Usually,

Algorithm 3: Linear Enumeration

```
1  Procedure LinerEnumeration(G, Q, Φ, g^i, C_{g^i})
2      /*{ u_{i+1}, u_{i+2}, ··· u_{|V(Q)|}} are isolation vertices */
3      for j = i+1 ← |V(Q)| do
4          Let v denote the first unvisited vertex in C_{g^i}[j]
5          if v does not exist then
6              return
7          Let g^{i+1} = g^i ∪ {v}
8          Mark v as visited
9      if there exists a uncovered vertex in g^{|V(Q)|} then
10         Add g^{|V(Q)|} into M
11         Mark every vertex in g^{|V(Q)|} as covered
12     for j = i+1 ← |V(Q)| do
13         foreach v ∈ C_{g^i}[j] where v is neither visited nor covered do
14             Let v' denote the j-th vertex in g^{|V(Q)|}
15             Update g^{|V(Q)|} = g^{|V(Q)|} ∪ {v} \ {v'}
16             Mark v as covered and v' as unvisited
17             Add g^{|V(Q)|} into M
```

this cost is much less than that for storing the data graph, and hence, our method is significantly efficient on space. For time cost, assume that the pruning rate (the probability that a search scope is covered) of a partial match is p, and there are k isolated vertices in Φ. Then, search with depth less than $|V(Q)| - k$ is based on the recursive process as Algorithms 1 and 2, while, for search with depth larger than $|V(Q)| - k$ (included), we apply linear enumeration as indicated in Algorithm 3. Let's discuss the time cost from these two different parts.

For the search with depth less than $|V(Q)| - k$, we first compute the number of search branches of different depth. Actually, a search branch of depth i exactly corresponding to a partial match of length i, i.e, g^i. We use n_i to denote the number of g^i, and hence n_1 is $O(|V(G)|)$. Since the pruning rate is p, and for each g^i, it would be extended into $O(d_G)$ (i.e., $O(|C_{g^i}[i+1]|)$) i+1-matches, i;e, g^{i+1}. Hence, $n_{i+1} = n_i \cdot (1-p) \cdot d_G$, and accordingly, n_i is:

$$O\left(|V(G)| \cdot (1-p)^{i-1} \cdot d_G^{i-1}\right) = O\left(|V(G)| \cdot (d_G - d_G{\cdot}p)^{i-1}\right)$$

where $1 \le i \le |V(Q)| - k$. Additionally, as we can see, when $i = 1$, $O\left(|V(G)| \cdot (d_G - d_G{\cdot}p)^{i-1}\right)$ is exactly $O(|V(G)|)$. Now let's analyze the time cost for each g^i. As previous discussion, the time for $Scope(g^i)$ being covered determination is $O(d_G{\cdot}(|V(Q)| - i))$. And the time for maintaining C_{g^i} (Lines 15-19 in Algorithm 2) is $O(|RN(u_i)| \cdot d_G)$, namely, $O(d_Q \cdot d_G)$ where d_Q and d_G are the average degree of query and data graphs, respectively. Therefore, time cost of g^i ($i \le |V(Q)| - k$) is

$$O\left(d_G \cdot (|V(Q)| - i + d_Q)\right)$$

And for a specific i, the time cost for all g^i is

$$O\left(|V(G)| \cdot d_G^i \cdot (1-p)^{i-1} \cdot (|V(Q)| - i + d_Q)\right)$$

Since both i and d_Q is less than $|V(Q)|$, we use $O(|V(Q)|)$ for $O(|V(Q)| - i + d_Q)$. Also, $|V(G)| \cdot d_G = |E(G)|$, then the time cost for all i-match could be simplified as follows:

$$O\left(|V(Q)| \cdot |E(G)| \cdot (d_G - d_G{\cdot}p)^{i-1}\right)$$

And the time for the search with depth less than $|V(Q)| - k$ is that

$$O\left(|V(Q)| \cdot |E(G)| \cdot \Sigma_{i=1}^{|V(Q)|-k}(d_G - d_G{\cdot}p)^{i-1}\right)$$

namely[5],

$$O\left(|V(Q)| \cdot |E(G)| \cdot (d_G - d_G{\cdot}p)^{|V(Q)|-k}\right)$$

For the search with depth not less than $|V(Q)| - k$, for each $g^{|V(Q)|-k}$, the time for linear enumeration is $O(k \cdot d_G)$, and hence, the total time cost for this part is $O(n_{|V(Q)|-k} \cdot k \cdot d_G)$, i.e.,

$$O\left(|V(G)| \cdot (d_G - d_G{\cdot}p)^{|V(Q)|-k-1} \cdot k \cdot d_G\right)$$

namely,

$$O\left(|E(G)| \cdot k \cdot (d_G - d_G{\cdot}p)^{|V(Q)|-k-1}\right)$$

After combining the two parts of time cost, we can see that the time complexity of our approach is

$$O\left((|V(Q)| + \frac{k}{d_G - d_G{\cdot}p}) \cdot |E(G)| \cdot (d_G - d_G{\cdot}p)^{|V(Q)|-k}\right)$$

4 Experimental Evaluation

4.1 Setup

We conduct the evaluation on a CentOS machine with two Intel Xeon Silver 4210R, 2.40 GHz and 512G memory. Our codes, query sets, as well as datasets are available in anonymous repository [1].

Comparative works. We compare our method (MatCo) with various state-of-the-art subgraph matching methods, including VEQ [23], DP-iso [17] (also known as DAF), NewSP [27], RapidMatch [36], GuP [3] and IVE [19]. We also compare our method with several counterparts in Section 4.4, to explicitly demonstrate the effectiveness of our different optimization strategies.

Table 2. Summary of datasets

Dataset	|V|	|E|	Degree	|L(V)|
Yeast	3,112	12,519	8	71
Human	4,674	86,282	36.9	44
WordNet	76,853	120,399	3.1	5
DBLP	317,080	1,049,866	6.6	15
YouTube	1,134,890	2,987,624	5.3	25
EU-2005	862,644	16,138,468	37.4	40
Orkut	3,072,441	117,185,083	76.2	100
Twitter	41,652,230	1,468,365,182	57	1000

Datasets. We apply 8 datasets in our evaluation. Both **Yeast** [17] and **Human** [4] are biological networks for human protein-protein interactions. **WordNet** [22] is a lexical dataset on semantic relationships between words. **DBLP** [17] is an academic social network dataset on co-authorships in computer science. **YouTube** [4] is a large social network dataset built over user interactions on the video-sharing platform. **EU-2005** [35] is a web graph under the .eu domain. **Orkut** [7] dataset is created from a free online social network where users form friendships with each other. **Twitter** [2] dataset is built over following relationships between Twitter users. Table 2 summarizes the statistical

[5]We assume that $d_G - d_G{\cdot}p > 1$

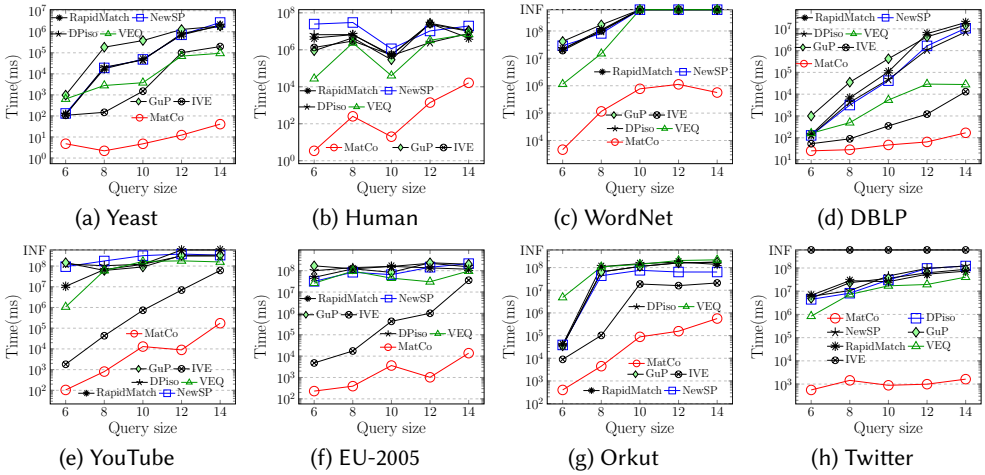


Fig. 6. Time efficiency comparison

information of these datasets. The first 6 datasets are also used in a recent subgraph matching survey [35]. The original versions of Yeast, Human, and WordNet are labeled datasets. While, DBLP, YouTube and EU-2005 were already assigned with labels in the survey. We directly download and apply these 6 labeled versions in our evaluation. For the remaining two large unlabeled datasets, i.e., Orkut and Twitter, we randomly generate their labels in the same way as the recent survey did. We set the label number of Twitter as 1000, which is much larger than that of Orkut (i.e., 100), since we find that every query could not finish in one week when we set the label number of Twitter as 1000.

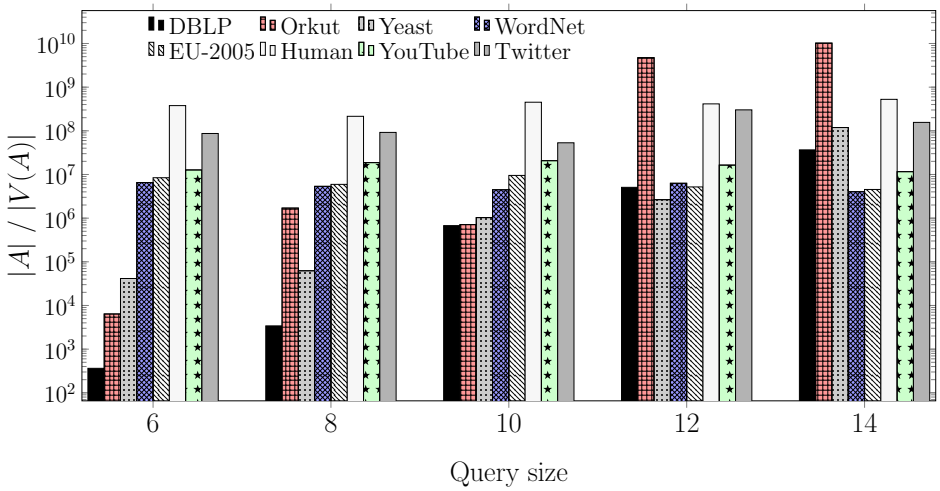


Fig. 7. The ratio $|A|$ / $V(A)$ varying query size

Query generation. We generate query graphs by randomly extracting subgraphs from the corresponding data graphs, which is similar to many previous subgraph matching works [27, 32, 35]. For each dataset, we set five different query sizes based on the number of query edges (i.e., $|E(Q)|$):

6, 8, 10, 12, and 14. Under each query size, we generate 30 query graphs with random walks over the data graph. We set the maximum execution time of each query as one week, and results of timeout queries would be removed. The reported time and space under a given group setting are obtained by averaging those from the corresponding generated queries.

4.2 Time Efficiency

We evaluate our method against comparative ones on overall time efficiency. Fig. 6 presents the results. We can see that our method outperforms comparative ones by 1~3 orders of magnitude. We obtain the greatest performance advantage on Twitter, which is the largest dataset. Note that, IVE fails to construct the index since memory would exceed, and we have no results for IVE over Twitter on time or space cost. Comparative works are all timeout on WordNet over queries of size larger than 10, which may be because WordNet has smallest number of different labels and candidate scale of each query vertex tends to be large. There is also a slight increase of time cost with query size grows, since the search space over large query tends to be deeper than that of small ones.

An interesting observation is that our advancement over comparative ones is smallest on DBLP dataset with queries of size 6 and 8. To analyze this, we conduct a statistics to present the distribution of $|A|/|V(A)|$ over different datasets, where A is the full set and $V(A)$ is the set of all key vertices. On average, an existing method (those computing the full set) need to find $|A|/|V(A)|$ matches to cover each key vertex. As indicated in Fig. 7, we can see that over queries of size 6 and 8, the ratio (i.e., $|A|/|V(A)|$) of DBLP is smaller than those of other datasets by nearly 2 orders of magnitude, which weakens our advantages since comparative ones could terminate much more early on DBLP than they do on other datasets. For queries of size 10, 12 and 14, the ratio ($|A|/|V(A)|$) of DBLP turns large and we can see from Fig. 6d that our method significantly outperforms comparative ones under these queries.

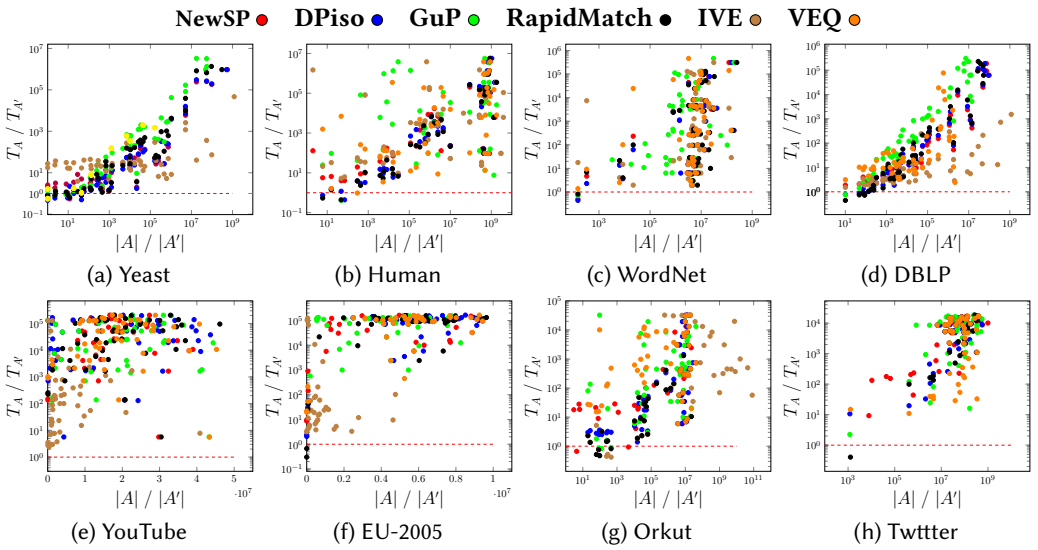


Fig. 8. Speedup distribution

We also present the speedup of our method against all comparative ones under different ratios of answer number to match cover size (i.e., $|A|/|A'|$). The speedup is $T_A/T_{A'}$ where T_A is the running time of the corresponding comparative work and $T_{A'}$ is that of our method. Fig.8 demonstrates

the specific distribution. Each point corresponding to a query, and points over red dotted line indicate speedups larger than 1, namely, cases when our method outperforms comparative ones. To avoid scattered points being too dense, we only randomly retrieve 10 queries under each size in this speedup evaluation. We can see that most points are over the red dotted line, and the very tiny portion of points that are below the red line gather at the area when the ratio (i.e., $|A|/|A'|$) is relative small. Since the running of IVE fails over Twitter dataset due to the memory limits, there is no (brown) points for IVE in Fig. 8h. Note that, on average, our method still outperforms comparative ones, as indicated in Fig. 6.

4.3 Space Efficiency

We evaluate the space efficiency of our method against comparative approaches. Fig. 9 presents the space cost for indexes/auxiliary data structures of different solutions. We can see that our method significantly outperforms comparative ones, by 1∼2 orders of magnitude under most settings. Space costs of these comparative methods are relatively stable with the query size increases, while our method exhibits a slight growth trend. A possible explanation is that, in our method, the number of vertex sets in local candidate space usually grows with query size, and hence, generally, the larger a query is, the more space our method costs. Nevertheless, the growth trend is very slight and our method is much more efficient than comparative ones. Noticeably, our advantages is most obvious on Twitter, i.e., the largest dataset, where our method outperforms comparative ones by almost 4 orders of magnitude.

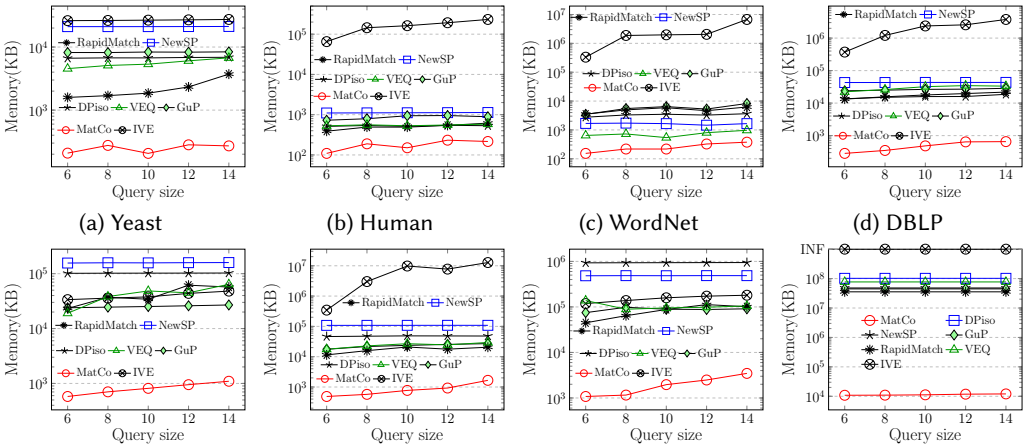


Fig. 9. Space efficiency comparison

4.4 Counterparts Comparison

We also set several counterparts of our method for evaluation, to demonstrate the efficiency of different optimizations in MatCo. The baseline, denoted as MatCo-Basic, is the basic search framework without local candidate space or linear enumeration optimization. The second counterpart, denoted as MatCo-LCS, is the version to incorporates local candidate space based pruning strategy into baseline. The third, denoted as MatCo-Linear, is the version combining baseline with the strategy to replace Cartesian products with linear enumeration. The final one, denoted as MatCo-Final, contains both local candidate space and linear enumeration optimization. Fig. 10 presents the results of counterparts evaluation. MatCo-Basic runs slowest on all datasets, and it is timeout

over half of the datasets, i.e., WordNet, YouTube, EU-2005 and Orkut. Our final version obviously outperforms all other counterparts. Also, MatCo-Linear runs faster than MatCo-LCS in most cases, which means our linear enumeration strategy is more effective than that the pruning strategy with local candidate space. Actually, the linear enumeration optimization reduces all search space deeper than i into efficient enumeration when the last $|V(Q)| - i$ query vertices are isolated, which is a significant acceleration.

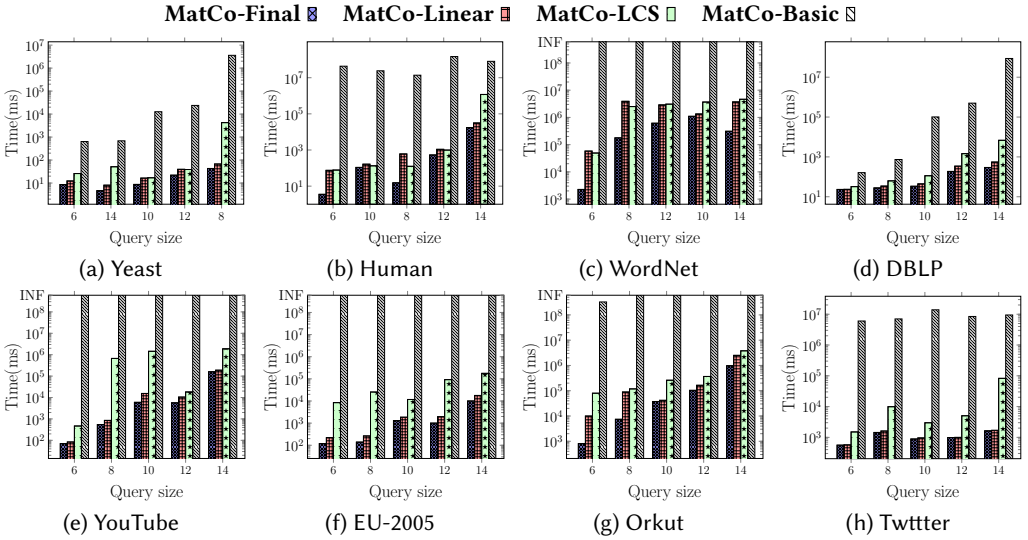


Fig. 10. Couterparts comparison

4.5 Pruning Rate with Local Candidate Space

We also present the pruning rate of our method over the given datasets. Pruning rate is the ratio of determinations that indicates covered search scope to all determinations conducted during our search. The higher the pruning rate is, the more efficient our method would be. Specifically, each determination costs linear time, while once we found a covered search scope, the search space that we prune would be of exponential scale. We can see from Fig. 11 that pruning rate is more than 20% over these datasets, which is relatively high, because, according to our previous discussion, the expected time cost we save on each g^i is $O(p \cdot {d_G}^{|V(Q)|-i})$, where p is the pruning rate while d_G is the average degree of data graph.

5 Related Work

Most existing subgraph matching works focus on static graphs [3, 17, 19, 23, 36, 46, 47], while some recent works consider the maintenance of matches over dynamic graphs [13, 24, 27, 32, 37, 44]. These works tend to return the full set of answers under the corresponding scenario. There are also many variants of subgraph matching, some of which return only a subset of answers, for example, TopK subgraph matching [10, 12, 15, 16, 45]. In this section, we discuss the related work in three categories: (1) traditional subgraph matching work (include both static and dynamic methods), (2) TopK subgraph matching and (3) other subgraph matching variants.

Traditional subgraph matching. There are many works on subgraph matching, and most of them focus on static graphs [3, 17, 19, 23, 36]. These static method usually constructs complicated

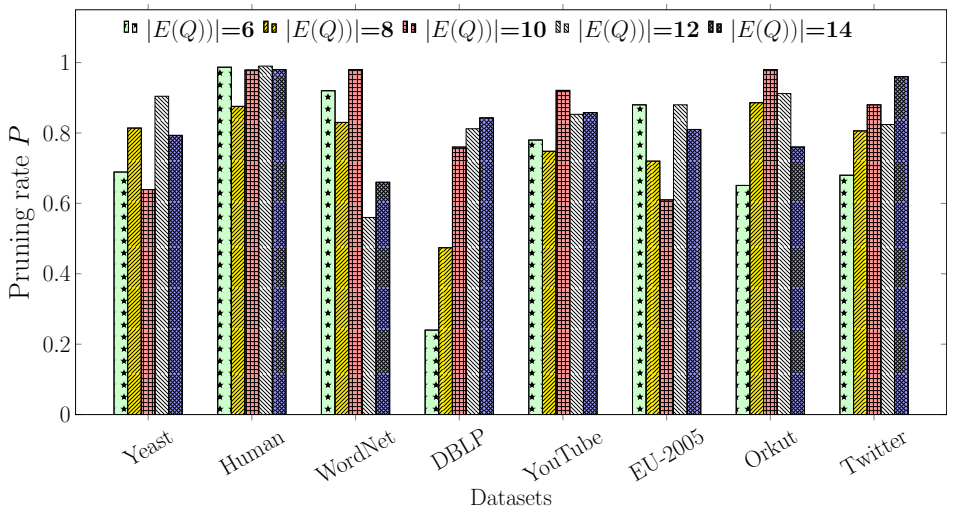


Fig. 11. Pruning rate over different datasets

indexes to filter unpromising candidates as many as possible, and then they would conduct search over the remaining candidates following a certain matching order. Indexes are usually built only once in offline way and queries for static subgraph matching are ad-hoc, hence, time cost for indexes construction is usually tolerable. Recently, there are also some works on dynamic subgraph matching, where they maintain all matches constantly when updates (usually, edge insertions/deletions) happen [13, 24, 27, 32, 37, 44]. These works tend to incorporate lightweight indexes/auxiliary data structures to reuse previous computation results, to accelerate current search for each update. To avoid repeated computation, they would conduct the search (or construct some local indexes) starting from the update parts of the dynamic graph. There is no essential difference between the search frameworks of static methods and dynamic ones. Both of them return all matches and have a common technical strategy: pruning partial matches that can not be extended into full ones as early as possible. Applying existing subgraph matching works for match cover would always lead to the maximum result, which maybe inefficient. Actually, existing works have no indexes/data structure to indicate a tight future search scope, and they accordingly have no way to determine whether future scope of some search branch has been covered. Hence, it is difficult to terminate existing algorithms in advance for match cover computation.

TopK subgraph matching. There are also some works that only return top k matches under certain constraints. KiSD [16] firstly proposes to compute top k densest matches of a given query graph over static weighted graphs. PBSM [10] also studies this TopK subgraph matching and propose graph compression based strategy to improve the performance. CSM-TopK [15] is the first to extend TopK subgraph matching to dynamic scenario, and they maintain density of star-structured partial matches to accelerate the search. These works focus on returning top k densest matches and their pruning strategies are usually based on density constraints, which is different from ours. Fan et al. [12] firstly propose diversified subgraph querying problem, where they compute all vertices that are not only from a set of k matches but also candidates of a specific query vertex. Also, they consider graph simulation [28] instead of isomorphism. DSQL [45] focuses on returning only k matches that cover the largest number of vertices. There is no guarantee that the returned k matches would constitute a match cover. Their solution iteratively search matches until k results are obtain.

In each iteration, they prioritize search matches of small overlapping with those already found. We can see that, none of these work could solve match cover computation.

Other variants. There are various variants of subgraph matching. Some works may break the generality of query graph where they search matches of paths [42, 43], cycles [30, 34], trees [18, 38, 40], DAGs [8, 9], Knowledge Graph [41] and so on. There are also some different matching semantics, for example, graph simulation [29], timing order constrained matching [26] and temporal subgraph matching [25, 31]. These works are far different from match cover computation and they are not the focus in this paper.

Overall, we are the first to propose match cover and existing related methods are difficult to be extended for computing match cover efficiently.

6 Conclusion

In this paper, we propose a new problem to compute match cover of a subgraph query over a data graph. A match cover is a subset of matches that collectively includes all vertices of the full set. Our target is to cover all target vertices as soon as possible during our search for matches. Existing subgraph matching methods tend to return a full set of matches, and there is no auxiliary data structure or mechanism to determine whether all target vertices have been covered. Hence, for match cover computation, existing work would always return the full set of answers, i.e., the maximum match cover, which is inefficient. We propose a new framework, called MatCo, to compute match cover efficiently. We design a local candidate space to indicate a tight future search scope of each partial match, with which we can efficiently determine whether the future search scopes have been covered. We can promptly prune partial matches that would not contribute to covering target vertices, to save computation. We also reduce time-consuming Cartesian products into linear enumeration for match cover computation. Extensive experiments over various datasets indicate that our solution significantly outperform comparative works. Additionally, computing minimal match cover could be an interesting future work.

Acknowledgments

This work was supported by NSFC(No.62472154).

References

[1] 2024. MatCo Codes. https://github.com/hnuGraph/MatCo.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. *arXiv preprint arXiv:1802.03760* (2018).

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[4] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[7] Hongtai Cao, Qihao Wang, Xiaodong Li, Matin Najafi, Kevin Chen-Chuan Chang, and Reynold Cheng. 2024. Large Subgraph Matching: A Comprehensive and Efficient Approach for Heterogeneous Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2972–2985.

[8] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Efficient Algorithms for Pattern Matching on Directed Acyclic Graphs. In *Proceedings of the 21st International Conference on Data Engineering*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 384–385. doi:10.1109/ICDE.2005.56

[9] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 493–504.

[10] Wei Chen, Jia Liu, Ziyang Chen, Xian Tang, and Kaiyu Li. 2018. PBSM: an efficient top-K subgraph matching algorithm. *International Journal of Pattern Recognition and Artificial Intelligence* 32, 06 (2018), 1850020.

[11] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849* (2015).

[12] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified Top-k Graph Pattern Matching. *Proc. VLDB Endowment* 6, 13 (2013), 1510–1521. doi:10.14778/2536258.2536263

[13] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

[14] Qingshuai Feng, You Peng, Wenjie Zhang, Ying Zhang, and Xuemin Lin. 2022. Towards Real-Time Counting Shortest Cycles on Dynamic Graphs: A Hub Labeling Approach. In *38th IEEE International Conference on Data Engineering, Kuala Lumpur, Malaysia, May 9-12, 2022*. 512–524. doi:10.1109/ICDE53745.2022.00043

[15] Chuchu Gao, Youhuan Li, Zhibang Yang, and Xu Zhou. 2024. CSM-TopK: Continuous Subgraph Matching with TopK Density Constraints. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3084–3097. doi:10.1109/ICDE60146.2024.00239

[16] Manish Gupta, Jing Gao, Xifeng Yan, Hasan Cam, and Jiawei Han. 2014. Top-k interesting subgraph discovery in information networks. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 820–831.

[17] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[18] Christoph M. Hoffmann and Michael J. O'Donnell. 1982. Pattern Matching in Trees. *J. ACM* 29, 1 (1982), 68–95. doi:10.1145/322290.322295

[19] Zite Jiang, Shuai Zhang, Xingzhong Hou, Mengting Yuan, and Haihang You. 2024. IVE: Accelerating Enumeration-Based Subgraph Matching via Exploring Isolated Vertices. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4208–4221.

[20] Cliff Joslyn, Sutanay Choudhury, David Haglin, Bill Howe, Bill Nickless, and Bryan Olsen. 2013. Massive scale cyber traffic analysis: a driver for graph database research. In *First International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1695–1698.

[22] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. (2017).

[23] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.

[24] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the*

2018 international conference on management of data. 411–426.
[25] Faming Li, Zhaonian Zou, and Jianzhong Li. 2023. Durable Subgraph Matching on Temporal Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4713–4726. doi:10.1109/TKDE.2022.3148995
[26] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1082–1093.
[27] Ziming Li, Youhuan Li, Xinhuan Chen, Lei Zou, Yang Li, Xiaofeng Yang, and Hongbo Jiang. 2024. NewSP: A New Search Process for Continuous Subgraph Matching over Dynamic Graphs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3324–3337.
[28] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2014. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.* 39, 1 (2014), 4:1–4:46. doi:10.1145/2528937
[29] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2014. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.* 39, 1 (2014), 4:1–4:46. doi:10.1145/2528937
[30] Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. 2011. Approximate Counting of Cycles in Streams. In *Algorithms - ESA 2011 - 19th Annual European Symposium (Lecture Notes in Computer Science, Vol. 6942)*, Camil Demetrescu and Magnús M. Halldórsson (Eds.). Springer, 677–688. doi:10.1007/978-3-642-23719-5_57
[31] Seunghwan Min, Jihoon Jang, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2024. Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3257–3269. doi:10.1109/ICDE60146.2024.00252
[32] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proc. VLDB Endow.* 14, 8 (2021), 1298–1310. doi:10.14778/3457390.3457395
[33] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. 2006. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)* 24, 2 (2006), 115–139.
[34] 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. *Proc. VLDB Endowment* 11, 12 (2018), 1876–1888. doi:10.14778/3229863.3229874
[35] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.
[36] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endowment* 14, 2 (2020), 176–188.
[37] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. Rapidflow: An efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.
[38] Mohammed Amin Tahraoui, Karen Pinel-Sauvagnat, Cyril Laitang, Mohand Boughanem, Hamamache Kheddouci, and Lei Ning. 2013. A survey on tree matching and XML retrieval. *Comput. Sci. Rev.* 8 (2013), 1–23. doi:10.1016/J.COSREV.2013.02.001
[39] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.
[40] Jason Tsong-Li Wang, Kaizhong Zhang, Karpjoo Jeong, and Dennis E. Shasha. 1994. A System for Approximate Tree Matching. *IEEE Trans. Knowl. Data Eng.* 6, 4 (1994), 559–571. doi:10.1109/69.298173
[41] Bo Wei, Xi Guo, Xiaodi Li, Ziyan Wu, Jing Zhao, and Qiping Zou. 2024. Construct and Query A Fine-Grained Geospatial Knowledge Graph. *Data Sci. Eng.* 9, 2 (2024), 152–176. doi:10.1007/S41019-023-00237-4
[42] Sun Wu and Udi Manber. 1992. Path-Matching Problems. *Algorithmica* 8, 2 (1992), 89–101. doi:10.1007/BF01758837
[43] Qingwu Yang and Sing-Hoi Sze. 2007. Path Matching and Graph Matching in Biological Networks. *J. Comput. Biol.* 14, 1 (2007), 56–67. doi:10.1089/CMB.2006.0076
[44] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[45] Zhengwei Yang, Ada Wai-Chee Fu, and Ruifeng Liu. 2016. Diversified Top-k Subgraph Querying in a Large Graph. In *Proceedings of the 2016 International Conference on Management of Data*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1167–1182. doi:10.1145/2882903.2915216
[46] Peipei Yi, Jianping Li, Byron Choi, Sourav S. Bhowmick, and Jianliang Xu. 2022. FLAG: Towards Graph Query Autocompletion for Large Graphs. *Data Sci. Eng.* 7, 2 (2022), 175–191. doi:10.1007/S41019-022-00182-8
[47] Kangfei Zhao, Zongyan He, Jeffrey Xu Yu, and Yu Rong. 2023. Learning with Small Data: Subgraph Counting Queries. *Data Sci. Eng.* 8, 3 (2023), 292–305. doi:10.1007/S41019-023-00223-W

Received October 2024; revised January 2025; accepted February 2025