Supplementary Marterial

This sections outlines our adaptions employed for homomorphic matching and the relevant evaluations contrasting recent methods including RapidMatch (RM) and Graphflow (GF). Subsection A and B presents our adaptations and evaluations, respectively.

This supplementary material is also publicly available on line: https://github.com/TempestHe/TriFMatch

A. Adaptation for Homomorphic Matching

In this subsection, we introduce how the filters in TriFMatch is adapted for homomorphic matching.

Definition 1 (Graph homomorphism) Given two graphs G_1 and G_2, G_1 is homomorphic to G_2 if there exists a mapping $M : V(G_1) \mapsto V(G_2)$ such that: 1) $\forall v_1 \in V(G_1), L(v_1) = L(M(v_1))$; 2) $\forall \langle v_1, v_1' \rangle \in E(Q), \langle M(v_1), M(v_1') \rangle \in E(G_2)$ and vice versa. The mapping M is referred to as an homomorphism.

Problem statement. Homomorphic subgraph matching aims to identify all the subgraphs in G that are homomorphic to Q. In contrast to subgraph matching based on isomorphisms, homomorphic matching is more relaxed, allowing multiple query vertices to be matched to the same data vertex.

Algorithm 1 can be seamlessly adapted for homomorphic matching by disabling the injectivity checking in Line 9, where multiple query vertices are prevented from being matched to the same data vertex. As for the filters, we have the following adaptions.

Adapting the failure-driven filter. Since homomorphism allows multiple query vertices to be matched to the same data vertex, the *conflict* failure no longer exists. Hence, our conflict filter is not necessarily required for homomorphic matching, and our failure-driven filter consists solely of the *emptyset* filter.

Adapting the containment-driven filter. Similarly, to adapt the containment-driven filter, we disable conditions related to the *conflict* failures. Specifically, we disable condition (E2) within the exclusion contraints, which requires marking conflit failures. Hence, we have the adapted exclusion constraints as follows:

Adapted exclusion constraints. For any two sibling states M_x and M_y, the exclusion constraints are defined as:

- (E1) $\forall u_s \in N_{\varphi}^{-}(u)$, $v_x \notin \pi(M_y)[u_s]$.

In contrast, the completion contraints remain unchanged from isomorphic matching, as they are not relevant to *conflict* failures. However, there is an exclusion case, which is illustrated in Figure 1c. Similar to isomorphic matching, for any state containing v_x (e.g., M_y'), additional contraints are needed to ensure the existence of another state (e.g., M_x') antisymmetric to M_y'. However, as homomorphism allows a data vertex to be matched multiple times, a vertex u_j adjacent to u' can also be matched to v_y. In this case, an antisymmetric state like M_x' can never exist, as no self-loop around v_y can match the query edge $\langle u', u_j \rangle$. Consequently, if M_y' is a solution, M_y will be mistakenly filtered due to the absence of a corresponding solution M_x' in $\mathcal{T}(M_x)$.

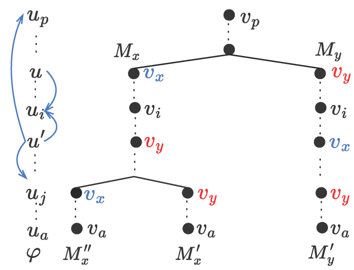


Fig. 1: An exclusion case for the adapted containment-driven filter

To address this discrepancy, we introduce an additional condition, ensuring another state $M_x'' = M_y' - \{(u, v_y), (u', v_x), (u_j, v_y)\} + \{(u, v_x), (u', v_y), (u_j, v_x)\}$ exists (shown in Figure 1c). This condition ensures that if M_y' is a solution, a corresponding solution M_x'' must exists within $\mathcal{T}(M_x)$, preventing the erroneous filtering of M_y. Therefore, we add condition (C3) to the completeion contraints as follows.

Adapted completion constraints. For any two sibling states M_x and M_y, the completion constraints w.r.t. the antisymmetric source u' are defined as:

- (C1) $\forall u_p \in \mathrm{Dm}(M_x) \cap N(u'), \langle v_y, M_x(u_p) \rangle \in E(G)$.
- $\forall u_j \in N(u') - \mathrm{Dm}(M_y) - N(u)$:
  - (C2) $T \cap N(v_y) - \{v_x, v_y\} \supseteq T \cap N(v_x) - \{v_x, v_y\}$
  - (C3) if $v_y \in T$, then $v_x \in T$.

  where $T = \pi(M_y)[u_j]$, if $u_j \in \mathcal{N}(M_y)$, otherwise $T = C(u_j)$

Proposition 1. Let M_x and M_y be sibling states. If M_x and M_y satisfy the adapted completion constraints w.r.t. every $u' \in \alpha(M_y)$, then the subtree $\mathcal{T}(M_x)$ contains the subtree $\mathcal{T}(M_y)$, where $\alpha(M_y)$ denotes the set of antisymmetric sources for M_y.

Proof. As proven in Proposition 6.3, in most cases, we can find a state M_x' symmetric or antisymmetric to M_y'. Hence, we focus on proving the existence of M_x'' for this exclusion case. To establish the existence of M_x'', we only need to prove that M_x'' is a homomorphism. Compared to M_y', the data vertices matched to u, u' and u_j are different in M_x''. Therefore, we need to prove that all edges adjacent to these vertices are matched in M_x''. In the proof of Proposition 6.3, we have already proved that all the edges adjacent to u and u' are matched. Hence, we only need to prove that the edges adjacent to u_j are matched. According to condition (C3), u_j must be a antisymmetric source, as $v_x \in T$ is a candidate for u_j. Consequently, condition (C1) and (C2) guarantee that all the edges adjacent to u_j are matched in M_x''.

Summary. Algorithm 3 summarizes the adapted filters for homomorphic matching. The failure-driven filter is simplified with the *emptyset* filter only (Lines 3-4), while the containment-driven filter incorporates the adapted exclusion and completion contraints (Lines 9-12).

B. Evaluation on Homomorphic Matching

Experimental setups. Utilizing the same dataset and hardward platform as the isomorphic matching experiments, we

Algorithm 3: IsFeasible

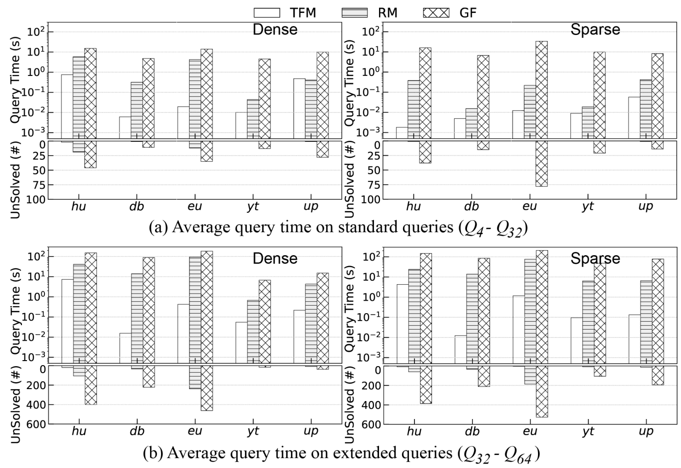
```
Input: the Search State M_y
Output: the Feasibility of M_y
Function IsFeasible(M_y):
    // Failure-driven filter
    if M_y is pruned by the emptyset filter then
        return False;
    // Containment-driven filter
    // 𝓜 is the set of explored sibling
       states without solutions
    for M_x ∈ 𝓜 do
        if π(M_x) contains π(M_y) then
            if M_x and M_y satisfy the adapted exclusion
               constraints then
                return False;
            if ∀u' ∈ α(M_y), M_x and M_y satisfy the
               adapted completion constraints w.r.t. u' then
                return False;
    return True;
```

evalute TriFMatch against the recent homomorphic matching methods. Since most exploration-based methods are desinged for isomorphic matching, we can only compare our method with the recent join-based methods, including GraphFlow (GF) and Rapidmatch (RM). GF, implemented in JAVA, is compiled using JAVAC 11.0.22, while RM is programmed in C++ and compiled by g++-9.4.0 with -O3 flag enabled. Following the default settings of GF, we build the catalogues for the data graphs and we set the java heap size to be the maximum available memory. During evaluation, we also noticed that RM with failing set pruning encounters a result missing issue, due to its incorrect implementation. Hence, in default, we disabled failing set pruning ensuring its enumeration completeness. Throughout the evaluation, we adopt the same evaluation metrics used in the isomorphic experiments.

Overall performance. Figure 2 illustrates the overall performance on both standard and extended queries. In most cases, TFM outperforms RM, achieving speedups of up to $216\times$ and $884\times$ on standard and extended queries respectively. This significant performance advantage stems from TFM's ability to drastically reduce the number of unsolved queries. Compared to isomorphic matching, the unsovled queries of TFM and RM are significantly fewer in the homomorphic matching, as homomorphic solutions are more general and easier to be found.

Memory cost. Table I presents the peak memory consumption during enumeration. Since our memory monitoring thread is implemented in C++, we cannot measure the memory cost of GF precisely. Nevertheless, the memory cost of GF is modest compared to capacity of the memory. As observed with isomorphic matching, RM generally incurs lower memory cost than TFM. However, given the capacity of modern memory devices, this difference is also negligible in practice.

Enumeration completeness Table II presents the number of queries with missing results (Row "Missed") and the speedups after enabling failing set pruning in RM. We validate the enumeration completeness of each method by focusing on



(a) Average query time on standard queries (Q_4 - Q_{32})

(b) Average query time on extended queries (Q_{32} - Q_{64})

Fig. 2: Overall performance comparison

Dataset		*hp*	*ye*	*wd*	*hu*	*db*	*eu*	*yt*	*up*
Standard Queries	TFM	0.02	0.09	70.57	0.49	1.04	2.76	1.38	13.60
	RM	**0.004**	**0.01**	**0.40**	**0.02**	**0.02**	**0.20**	**0.18**	**1.81**
Extended Queries	TFM	0.09	0.14	71.55	1.57	0.82	3.22	2.40	**8.03**
	RM	**0.01**	**0.10**	**11.63**	**0.17**	**0.06**	**0.93**	**2.16**	84.14

TABLE I: Peak memory consumption (MB)

queries with fewer than 10^5 solutions, as shown in Row "Checked". Our validation confirms that TFM, GF and RM without failing set pruning consistently produce the correct number of solutions. RM with failing set pruning, however, processes up to 100% standard queries and 71% extended queries with missing results. We attribute this discrepancy to RM's incorrectly implemented failing set pruning technique, as our TFM is also equipped with failing set pruning correctly adapted for homomorphic matching.

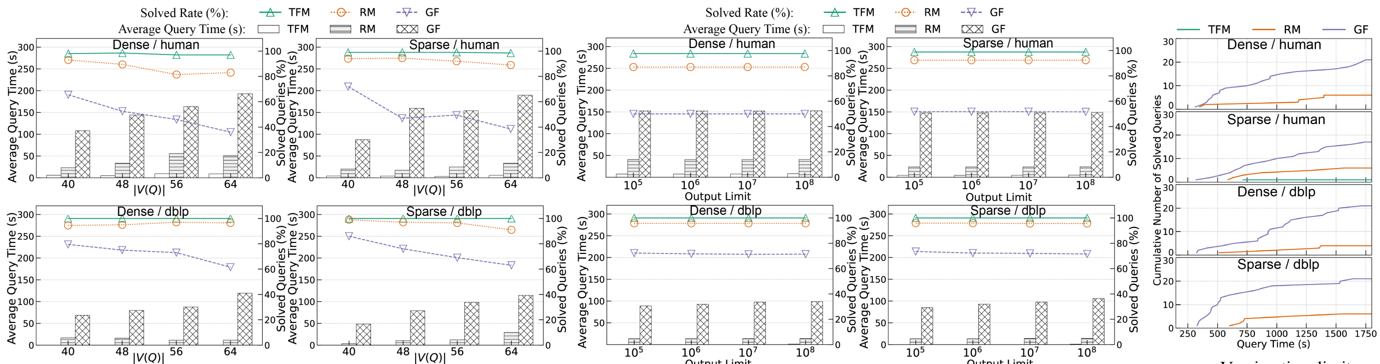
Although failing set pruning is originally proposed dealing with isomorphic matching, it can be seamlessly applied to homomorphic matching. Specifically, failing set pruning primarily involves computing a failing set for each state M, which records the query vertices associated with the *emptyset* and *conflict* failures encountered in $\mathcal{T}(M)$. As *conflict* failures do not occur during homomorphic matching, failing set pruning can be seamlessly integrated.

Regarding the time efficiency, the performance of RM can be degraded significantly on datasets such as *hu* and *db*, as missing results postpone the algorithm's termination.

Sensitivity evaluation. Figure 3 presents the effect of query size, output limit and time limit on query performance. Firstly, in Figure 3a, as the query size increases, the query time also increases, while the percentage of unsolved queries decreases consistently for RM and GF. However, TFM maintains rapid

Dataset		*hp*	*ye*	*wd*	*hu*	*db*	*eu*	*yt*	*up*
Standard Queries	Missed	401	328	4	37	171	43	134	355
	Checked	1751	690	5	43	200	43	168	538
	Speedup	0.7	134.3	0.9	0.1	0.9	0.5	3.7	6.6
Extended Queries	Missed	361	94	449	0	0	0	78	355
	Checked	1336	146	632	0	0	0	254	1000
	Speedup	0.8	4.7	2.0	1.0	1.1	1.0	13.5	1.8

TABLE II: Effect of *failing set pruning* on enumeration completeness and query performance in RM



(a) Varying query size $|V(Q)|$ (on extended queries)

(b) Varying output limit (on extended queries)

(c) Varying time limit (on extended queries)

Fig. 3: Sensitivity evaluation on dataset *hu* and *db* varying (a) query size; (b) output limit (c) time limit

query performance by solving almost all queries across all cases. Secondly, Figure 3b shows the query performance when expanding the output limit up to 10^9. Compared to isomorphic matching, increasing the output limit does not significantly affect the time cost, as homomorphic solutions are relatively easier to find. Thirdly, Figure 3c displays the cumulative number of queries solved between 5 and 30 minutes. GF incrementally solves the most queries, as it originally solves the fewest queries within the initial 5-minute setting, leaving more queries to be solved potentially. In constrast, TFM solves almost no additional queries incrementally, as nearly all queries are solved in the original 5-minute setting. Nevertheless, TFM solves the most queries overall within 30 minutes.

Ablation study. Figure 4 presents the results of TFM by disabling each filter individually. The baseline configuration, denoted as *Base*, utilizes all filters, while *w/o* ☐ represents the configuration with the filter specified in ☐ disabled. Our evaluated filters include *CDFilter* (containment-driven filter), *FDFilter* (failure-driven filter) and *FSPruning* (failing set pruning). In most cases, *CDFilter* demonstrates the most significant impact, particularly on dense queries. Without *CDFilter*, the number of unsolved queries grows substantially. *FSPruning* is relatively more effective on sparse queries and datasets like *yt* and *up*.

Table III further details the percentage of queries where each filter can effectively reduce at least one state. consistent with the results observed in isomorphic matching, *FDFilter* demonstrate effectiveness across a majority of queries. Both of our filters, *FDFilter* and *CDFilter*, outperforms *FSPruning* in terms of the number of queries where they exhibit effectiveness. Interestingly, on dense queries, *FSPruning* exhibits effectiveness on fewer queries than on sparse queries.

Dataset		*hu*	*db*	*eu*	*yt*	*up*
Dense Extended Queries	CDFilter	9.1%	11.7%	32.6%	75.7%	47.1%
	FDFilter	**23.5%**	**31.7%**	**56.6%**	**100%**	**81.7%**
	FSPruning	5.2%	7.2%	18.1%	70.8%	7.7%
Sparse Extended Queries	CDFilter	7.7%	14.8%	33.3%	89.5%	79.1%
	FDFilter	**22.5%**	**41.5%**	**67.2%**	**99.8%**	**99.1%**
	FSPruning	7.0%	11.3%	26.3%	85.6%	40.8%

TABLE III: Percentage of queries, where each filtering technique is effective

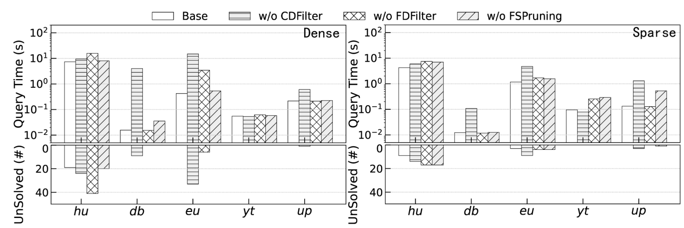
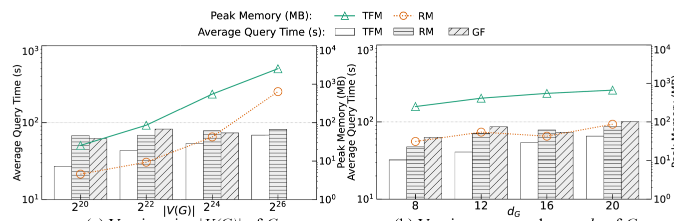


Fig. 4: Ablation study on filtering techniques



(a) Varying size $|V(G)|$ of G

(b) Varying average degree d_G of G

Fig. 5: Scalability evaluation on synthetic graphs

Scalability evaluation. Figure 5 illustrates the performance of the algorithms on large synthetic graphs varying sizes $|V(G)|$ and average degree d_G. In most cases, TFM outperforms other baselines, achieving minimal query time. While TFM incurs higher memory cost than RM, both methods demonstrate strong scalability on these large graphs. However, GF encounters a memory overflow issue on the largest graph instance, comprising 2^{26} vertices and over $1B$ edges.