Understanding High-Performance Subgraph Pattern Matching: A Systems Perspective

Akshit Sharma
akshitsharma@mines.edu
Colorado School of Mines
Golden, CO, USA

Dinesh Mehta
dmehta@mines.edu
Colorado School of Mines
Golden, CO, USA

Bo Wu
bwu@mines.edu
Colorado School of Mines
Golden, CO, USA

Abstract

Subgraph isomorphism is a crucial problem in graph-analytics with wide-ranging applications. This paper examines and compares two high-performance solutions to this problem: backtracking, represented by VF3, and compilation, represented by Dryadic. Despite both strategies being based on vertex-extension mapping, Dryadic significantly outperforms VF3 across all tests, with speed-ups ranging from a minimum of 4.95x to a maximum of 165x. To understand these disparities, the paper identifies and explores five key optimizations in Dryadic: candidate vertices generation, execution specificity, data graph storage, matching order, and redundancy elimination. With these optimizations removed, Dryadic's performance substantially degrades but it is still on average 10x faster than VF3 due to better spatial locality and search-space pruning. With the insights gained from these optimizations, we propose and implement two new techniques: lazy evaluation in Dryadic and connectivity checks in VF3, resulting in performance improvements of up to 1.23x and 1.46x, respectively.

CCS Concepts

• **General and reference** → *Surveys and overviews*; • **Information systems** → **Data mining**.

Keywords

Subgraph Isomorphism, Dryadic, VF3, Subgraph Pattern Matching

ACM Reference Format:
Akshit Sharma, Dinesh Mehta, and Bo Wu. 2024. Understanding High-Performance Subgraph Pattern Matching: A Systems Perspective. In *7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3661304.3661897

1 Introduction

Graph data is abundant in various domains, including social networking [36], cheminfomatics [62], transportation [63], and web graphs [57]. One of the most useful ways to extract value from such data is through subgraph pattern matching (SPM), which finds all instances of a subgraph pattern in the input graph. Unfortunately, SPM is an NP-hard problem [16, 63], demanding heuristic algorithms and corresponding efficient implementations. One popular strategy to solve SPM is vertex extension-based matching, which incrementally builds a map of vertices from pattern graph to data graph, one vertex at a time. An instance of the pattern graph is found if the induced subgraph based on the mapped vertices in the data graph is isomorphic to the pattern graph.

The development of vertex extension-based matching techniques has given rise to two main methodologies, distinguished by their operational frameworks. Query-agnostic systems such as VF3 [10, 12, 17, 66, 70] leverage **backtracking**, which involves a recursive process to align vertices from the pattern graph with those in the data graph, a strategy often adopted in biotechnological contexts. Conversely, Query-specific solutions like Dryadic [14, 15, 38, 39, 50] apply **compilation** approaches, generating executable that does set operations on adjacency lists. This approach focuses on minimizing redundancy and enhancing efficiency in vertex matching.

Our work extends the experimental analysis provided by [54], which classifies SPM systems into direct-enumeration, indexing-enumeration, and preprocessing-enumeration, further dividing these into four components: candidate vertex generation, matching order, enumerating partial results, and additional optimizations like graph compression and infeasible set pruning. Our approach is orthogonal to their method as, (1) Their analysis enables an understanding of the relative strengths of the different systems, while our classification carefully quantifies the relative importance of various factors that contribute to large runtime differences observed between two approaches. (2) They try to understand the ability of different systems to scale with change in the density and number of vertices in the query graph among other factors, whereas we concentrate on the impact of optimization approaches on general performance. This is achieved through the development of several intermediate implementations between representatives of the two approaches: VF3 (fastest available in that group) and Dryadic (shown to be better than CECI [6], a previous state-of-art in the other group). We used six data graphs and ten pattern graphs, with sizes varying from 3 to 6. The use of smaller, more frequent patterns helps to highlight the redundancy reduction more clearly. Using geometric mean, we found Dryadic outperformed VF3 by as much as 165x, found by applying all the pattern graphs on each data graph. We also found Dryadic performed better in each data graph with at least 4.95x geometric mean speedup across all the pattern graphs.

In this paper, we investigate the performance disparity between Dryadic and VF3, both utilizing vertex extension strategies, by examining the impact of various optimizations in Dryadic. We specifically looked into the following: 1). Dryadic pre-compiles a unique, highly optimized executable for each pattern graph, leveraging

GRADES-NDA '24, June 14, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0653-0/24/06.
https://doi.org/10.1145/3661304.3661897

compile-time knowledge for optimization. Conversely, VF3 generates a single executable capable of processing both pattern and data graphs dynamically, lacking pattern-specific optimizations. To align Dryadic's approach with VF3's flexibility, we modified Dryadic to produce a generic matching program that accommodates any subgraph pattern, eliminating the need for compiling a new executable for each pattern (§ 3.2). Comparatively, this adaptation resulted in slower performance for Dryadic, as expected, but offered the convenience of not requiring a new executable for each pattern graph. 2). Further analysis reveals VF3's usage of binary search for edge finding and Dryadic's efficient data structure (§ 3.3) for label processing and set operations, offering linear time operations as opposed to VF3's $O(n \log n)$. By aligning Dryadic closer to VF3's binary search approach (§ 3.1), we assessed the performance implications, finding linear searches more effective in Dryadic's context. 3). Apart from this we also used matching order (§ 3.4) from VF3 to dryadic, and removed code motion (§ 3.5) and determined that these changes did not have a significant impact on the performance. We also improve the system by applying lazy evaluation in Dryadic to avoid unnecessary pre-computation and optimizing VF3's connectivity checks for efficiency. This implementation demonstrates improvements up to 1.23x and 1.46x, respectively.

This paper makes the following contributions:

- Provides a comprehensive evaluation of two subgraph pattern matching systems, Dryadic and VF3, through many subgraph patterns and a set of real-world graphs.
- Implements five variants of Dryadic to understand the benefits of its optimizations.
- Improves the performance of Dryadic through lazy evaluation with only 100 lines of code change.
- Improves the performance of VF3 by modifying the connectivity constraints check with only 10 lines of code change.

2 Background and Motivation

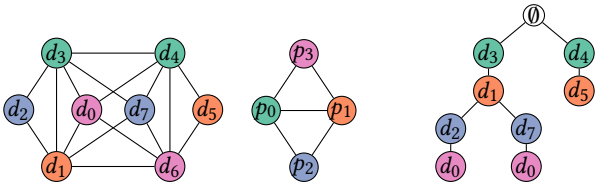
In this section, we provide a brief overview of subgraph pattern matching, followed by an explanation of the two systems, Dryadic and VF3, and a comparison of their performance.

2.1 Subgraph Pattern Matching

Given a data graph $D = (V_d, E_d)$, a subgraph $S = (V_s, E_s)$ of D is defined by $V_s \subseteq V_d$ and $E_s = E_d \cap (V_s \times V_s)$. A pattern graph $P = (V_p, E_p)$ is said to be a subgraph of D if it is isomorphic to a subgraph S of D; i.e., there exists a bijective function $f\colon V_p \to V_s$ such that $(p_1, p_2) \in E_p \iff (f(p_1), f(p_2)) \in E_s$. Further, if each graph vertex v has label $l(v)$, then the *labeled* subgraph pattern matching problem additionally requires that the bijective function f satisfies $l(p) = l(f(p))$ for any vertex p in V_p. In the example of Fig. 1b and Fig. 1a below, there are two such bijective functions: $\{(p_0, d_3), (p_1, d_1), (p_2, d_2), (p_3, d_0)\}$ and $\{(p_0, d_3), (p_1, d_1), (p_2, d_7), (p_3, d_0)\}$.

2.2 Matching through Vertex Extension

Vertex extension is crucial in subgraph pattern matching, where the mapping f is constructed incrementally by matching vertices from pattern graph P to those in data graph D. This process uses recursion-tree-based algorithms (e.g., VF3) or set-operation-based



(a) Data **(b) Pattern** **(c) Matching Tree**

Figure 1: Example of subgraph pattern matching.

algorithms (e.g., Dryadic), employing heuristic strategies to find the optimal for vertex-matching order.

2.3 Dryadic

Dryadic optimizes subgraph pattern matching in two steps: 1). preprocessing the pattern graph to establish vertex matching order and generate nested for-loop C++ code; 2). compiling this code to create an executable for varied data graphs. This approach, bypasses symmetric pattern outputs for comparability with VF3. The nested loops in dryadic iterate over data graph (Fig. 1a) vertices matching the pattern graph's (Fig. 1b) ordered vertices (p_0, p_1, p_2, p_3), employing set operations based on edge presence (intersection) or absence (difference) for efficient pattern matching. A more detailed explanation of this method is provided in Appendix C.

2.4 VF3

VF3 is a recursive backtracking algorithm (Algorithm 4) that expands a match by iteratively pairing pattern vertices with data vertices, considering both graphs to determine the matching order. Unlike Dryadic, VF3 adapts its vertex matching order based on both the pattern and the current data graph, necessitating pre-computation for each data graph. We use VF3Light [10], an enhanced version that optimizes the matching process without a look-ahead step. VF3 iterates through states, checking for goal achievement or dead ends, and utilizes pattern-data vertex pair feasibility to guide recursion, ensuring pattern graph isomorphism with the data graph. A more detailed explanation of this method is provided in Appendix D.

2.5 Performance Comparison

We compared the runtime performance of Dryadic and VF3 on six data graphs, spanning from social networks to citation and collaboration networks, as detailed in Table 1. The data graphs vary in size, with vertices ranging from 96K to 3.7M and edges exceeding 1M, marked with 10 distinct labels according to a uniform distribution. The experiments utilize ten pattern graphs of up to six vertices, differentiated by distinct labels, shown in Appendix A.

Table 1: Datasets used in experiments.

Graphs	Symbol	Vertices	Edges	Description
DBLP [64]	DBLP	317K	1M	Collaboration
MiCo [19]	Mico	96K	1.1M	Co-authorship
YouTube [64]	YT	1.1M	2.9M	Social Network
Cit-Patents [35]	CitPa	3.7M	16.5M	US Patents
LiveJournal [64]	LiJo	4M	34.7M	Social Network
Orkut [64]	Ork	3.1M	117.2M	Social Network

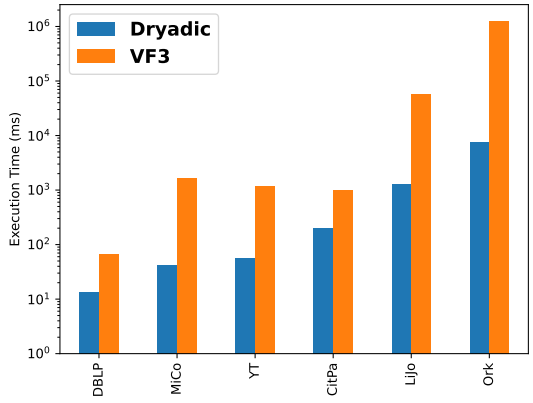


Figure 2: Performance comparison of Dryadic vs VF3.

Our tests were conducted on an Intel Xeon CPU E5-4610, using a single thread and 188 GB DDR3 RAM. Both Dryadic and VF3 compiled with -O3 optimization, running under Ubuntu 18.04 LTS. The timeout for each run was set to 10 hours. The comparison (Fig. 2) uses a logarithmic scale due to the significant performance gap between Dryadic and VF3, highlighting Dryadic's superiority, particularly on larger graphs. Notably, Dryadic's performance on the Orkut dataset outperformed VF3 by up to two orders of magnitude.

3 Quantifying the Performance Benefits of Optimizations

In this section, we conduct a systematic experimental analysis to evaluate the impact of a series of optimizations on the relative performance of the two approaches. We modify Dryadic in each of the first 5 subsections below eliminate one optimization to bring it closer to VF3. For consistency of presentation, each subsection has four parts: 1) Describing details of the Dryadic implementation 2) Describing details of the VF3 implementation 3) Describing changes made to degrade Dryadic and 4) Presenting results and discussion. While these modifications narrow the gap between (the degraded) Dryadic and VF3, there remains a sizeable difference between the two. § 3.6 offers two explanations for the remaining gap.

3.1 Candidate generation

3.1.1 Dryadic Dryadic employs set intersection and difference operations for solving subgraph isomorphism, processing two sorted arrays of vertex indices. This process parallels the "merge" phase of mergesort, where pointers initiate at the beginning of each array and advance based on the comparison outcomes, filling the output array accordingly. These operations exhibit a time complexity of $O(m+n)$, with m and n representing the respective sizes of the two sets involved. Sets may represent neighbors of a vertex with a specific label $\mathrm{N}(v,l)$ or outcomes from preceding intersection and difference operations, typically smaller than the initial sets.

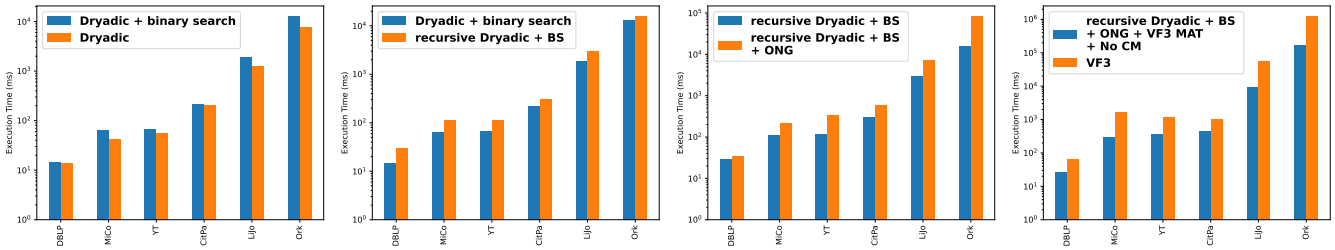
3.1.2 VF3 VF3 performs two checks in IsFeasiblePair(p_i, d_i) (Algorithm 4 Line 10) similar to the two set operations of Dryadic: the edge-presence and edge-absence checks.

1. *Edge-Presence Check*: The first check confirms that if a vertex p_i in the pattern subgraph has edges connecting it to vertices already matched, then the corresponding vertex d_i in the data subgraph must have analogous edges connecting it to the corresponding vertices. This is performed by iterating through p_i's neighbors in the pattern graph. For each neighbor p_j that was previously matched, we identify the corresponding data vertex d_α and perform a binary search for d_α in d_i's adjacency list. *E.g.,*: Consider a partial mapping $(p_0,d_3),(p_1,d_1),(p_2,d_7)$ to which we are adding (p_3,d_6). We must verify that $\overline{p_3p_1}$ has corresponding $\overline{d_6d_1}$ and $\overline{p_3p_0}$ has corresponding $\overline{d_6d_3}$. This is achieved by two binary searches in d_6's adjacency list for d_1 and d_3, respectively. The search for d_1 succeeds whereas the search for d_3 fails and InFeasiblePair() returns false.

2. *Edge-Absence Check*: The edge-absence check is implemented by checking the contrapositive; i.e., the presence of an edge in the *data subgraph* must be confirmed by the presence of the corresponding edge in the pattern subgraph. The algorithm is similar to the edge-presence check, with the roles of the data and pattern graphs reversed. *Example*: To illustrate, consider the data graph of Fig. 1a) augmented with an edge $\overline{d_5d_7}$. Consider a partial mapping $(p_0,d_4),(p_1,d_5),(p_2,d_7)$ to which we are adding (p_3,d_6) via IsFeasiblePair$((p_3,d_6))$. The algorithm iterates over d_6's neighbors. For each neighbor of d_6 that was previously matched, we confirm that an edge exists from p_3 to the corresponding vertex in the pattern subgraph. In the example, vertices d_4, d_5 and d_7 are neighbors of d_6 and mapped to p_0, p_1 and p_2, respectively. Edges $\left(\overline{d_6d_4},\overline{d_6d_5},\overline{d_6d_7}\right)$ correspond to vertex pairs $\left(\overline{p_3p_0},\overline{p_3p_1},\overline{p_3p_2}\right)$ in the pattern subgraph. Three binary searches on the neighbor list of p_3 are executed to perform this check. Since $\overline{p_3p_2}$ is absent, the addition of (p_3,d_6) will give a data subgraph that is not isomorphic to the pattern subgraph and IsFeasiblePair() returns false.

3.1.3 Implementation Changes We modify the Dryadic implementation so that both set operations are implemented *less* efficiently. We iterate on one array and use binary search on the other (sorted) array to mimic the implementation in VF3. In *set intersection*, we iterate on the smaller array: for each element x in the array, we perform a binary search for x in the other (larger) array. The element x is added to the output array if and only if the search is successful. Assuming $m \le n$, the time complexity is $O(m \log n)$. In *set difference*, we iterate on the first array: for each element x in the array, we perform a binary search for x in the second array. Element x is added to the output array if and only if the search is *not* successful. Assuming m and n are the sizes of the first and second sets, the complexity is $O(m \log n)$. Since the first set is the result of previous intersection and difference operations while the second is of the form $\mathrm{N}(v,l)$, we expect $m < n$.

3.1.4 Results Discussion Figure 3a shows a comparison using the geometric mean across various patterns, highlighting that 'dryadic' outperforms its 'dryadic + binary search' counterpart. Our asymptotic analysis, assuming m and n are not small constants, reveals that for arrays of similar size ($m = n$), the time complexity for set operations in 'dryadic' is $O(m+n) = O(n)$, as opposed to $O(m \log n) = O(n \log n)$ in the binary search approach. The comparison becomes less clear when array sizes significantly differ or are relatively small. Notably, the largest performance drop was observed in the Ork dataset (1.70x), and the least in the DBLP dataset (1.06x). It's important to recognize that Dryadic's set operations differ fundamentally from VF3's pairwise-matching computations,



(a) Binary search. **(b) recursive version.** **(c) Online label matching.** **(d) All the optimizations removed.**

Figure 3: Quantifying the performance benefits of optimizations.

except for triangle patterns where they align. Dryadic employs intersections for candidate data vertex identification, while VF3 checks for edge presence between the last data vertex and the two previously mapped vertices. For non-triangle patterns, Dryadic's intermediate arrays are smaller, reducing the search space. This difference complicates direct performance comparisons for other patterns, a topic further explored in § 3.6.

3.2 Iterative vs Recursive Implementations

3.2.1 Dryadic Since Dryadic computes an execution plan (nested for-loops) that is specific to a pattern graph, a new execution plan must be created and compiled whenever a different pattern graph is used. This gives one executable per pattern. If a new pattern graph is used, then it should be compiled first. The executable generated should then be run on the data graph.

3.2.2 VF3 VF3 is more flexible as it uses the same executable for different combinations of pattern and data graphs. It uses recursion, with each recursive call attempting to match a new pair of vertices. VF3's recursive nature prevents the compiler from inlining the recursive call because the depth of recursion is unknown at compile time. Another source of inefficiency in VF3 is the overhead due to the number of function calls, one for each node in the matching tree. The size of the tree is loosely related to its height and "width". Its height is bounded by the number of vertices in the pattern graph; i.e., $|V_p|$. It's "width" depends on the average number of available matching vertex pairs at each node. This is specific to each example and not easy to characterize analytically.

3.2.3 Implementation Changes To facilitate a comparison not dependent on compilation and execution optimizations, we revised Dryadic's architecture, substituting for-loops with recursive calls. The pattern graph's computation plan generates an instruction list instead of an executable file. A recursive function uses these instructions alongside the data graph to identify matches, mitigating the need for recompilation for each pattern but increasing function call overhead. Additionally, these function calls cannot be inlined.

Tail recursion optimization, which can minimize recursive call stack storage, is inapplicable due to post-recursive call computations. In this recursive setup, the framework selects the set operation, array segment and recursion level dynamically. Conversely, in the original iterative model, these elements were known at compile time. The specific set operation, target array, and loop position are established at compile time, enabling the compiler to apply pattern graph-specific optimizations, such as inlining set operations, rather than dynamic runtime selection.

3.2.4 Results Discussion In comparing 'dryadic + binary search' with 'recursive dryadic + BS' (Fig. 3b), where BS denotes binary search, we noted performance drops with recursion. The slowdown reached from 1.23x for Ork to 2.11x for DBLP against a binary Drydic version. As expected, the for loop version outperforms the recursive one due to pattern-specific code optimization.

3.3 Graph Representation

3.3.1 Dryadic Dryadic stores the data graph using binary compressed sparse row (CSR) format [38], augmented to efficiently store and retrieve vertex label information. Table 2 illustrates the data graph organization by labels Fig. 1a. The *vertices* array stores vertex indices sorted by their labels. It is indexed by the *labels* array so that vertices with label l are stored in locations $labels[l], \ldots, labels[l+1]-1$ in *vertices*. In the example, vertices with label 2 are stored in locations $vertices[4 \ldots 5]$, containing vertices 2 and 7. Fig. 1a confirms that both d_2 and d_7 have label 2. This data organization permits each vertex in L(label) to be computed in constant time. The modified-CSR representation in Dryadic similarly stores vertex adjacencies (edges) organized by labels. Although Table 3a uses two dimensions for convenience, it is stored and understood as the 1D array obtained by traversing the 2D array in row-major order.

Table 2: Vertex storage by label

labels	0	2	4	6	8
vertices	3	4	1	5	**2**	**7**	0	6

Thus, N(v, l), the neighbors of vertex v with label l are stored in locations $vertIdx[v*nL+l], \ldots, vertIdx[v*nL+l+1]-1$ in the *neigh* array. Here, nL is the number of distinct labels in the data graph. In the example of Fig. 1a, $nL = 4$. Accordingly, the neighbors of vertex 6 with label 1 may be found in locations 24 and 25 in *neigh*. The 1D *neigh* array is depicted using two rows in row-major order in Table 4, with the elements in locations 24 and 25 in boldface.

3.3.2 VF3 VF3 stores the labels per vertex id in an array. The labels are mapped to vertices by their index. In the array (3, 1, 2, 0, 0, 1, 3, 2), the label of vertex id 3 is 2. This array allows VF3 to check the label of a particular vertex in constant time ($O(1)$). VF3 stores neighbors of the vertices in the data graph in the form of an adjacency list (Table 3b). Each adjacency list is stored as an array with vertex id in increasing order. This helps VF3 to perform a binary search to check if a vertex is a neighbor of another or not (explained in § 3.1).

Table 3: Combined tables of Vertex Index and Adjacency List

(a) Vertex index table

Ln refers to the labels

vertIdx				
Idx	L0	L1	L2	L3
0	0	2	3	3
1	4	5	5	7
2	9	10	11	11
3	11	12	13	15
4	16	17	18	19
5	21	22	22	22
6	23	24	26	27
7	28	30	31	31

(b) Storage of adjacency lists in VF3

Idx	Adjacency list				
0	1	3	4	6	
1	2	3	6	7	
2	1	3			
3	0	1	2	4	7
4	0	3	5	6	7
5	4	6			
6	0	1	4	5	7
7	1	3	4	6	

Table 4: Neighbors table: indices shown for first and last elements

0															15
3	4	1	6	3	2	7	0	6	3	1	4	1	2	7	0
3	5	7	0	6	4	6	4	**1**	**5**	7	0	3	4	1	6
16															31

Picking next candidate data vertex: The VF3 implementation of NEXTPAIR() performs a linear search through the adjacency list of the last mapped data graph vertex for a data vertex d_α with the same label as the pattern vertex to be matched. Assuming labels are uniformly distributed, the average time for this search is $\Theta(nL)$, where nL is the number of labels. (Recall $nL = 10$, in our benchmark graphs.) In contrast, the label-aware CSR-based data structure described earlier can perform this search in $O(1)$ time.

As VF3 runs binary search on both pattern and data graphs, it stores the neighbors in both graphs in a sorted adjacency list. In the beginning, while finding the vertices to match the first pattern vertex (in the matching order), VF3 has a time complexity of $O(|V_d|)$. Otherwise the first and third step, VF3 has a time complexity of $O(V_{deg})$ and $O(V_{deg} log(|V_p|))$ (where V_{deg} is the maximum degree of a vertex in data graph). For Dryadic, the step similar to the first has constant time complexity to retrieve and linear to iterate over (it also iterates over vertices with match labels). Also, the set operations in the original Dryadic are linear but have been changed to one using binary search from version 'binary' since § 3.1.

3.3.3 Implementation Changes We downgraded Dryadic to construct the label-specific arrays online. Every time, L() or N() is called, the program iterates over the vertices and inserts ones with matched labels into an intermediate array. The new version 'recursive dryadic + BS + ONG' (ONG is Online Neighbor Generation) is compared to the previous version 'recursive dryadic + BS'.

3.3.4 Results Discussion The results between the two versions can be seen in Fig. 3c. This modification degraded Dryadic's performance by maximum 5.29x on Ork with the geometric mean across pattern graphs. As expected, calculating the neighbor list on runtime is more expensive and thus leads to worse execution time.

3.4 Matching Order

Matching order, the sequence in which pattern graph vertices are matched against data graph candidates, significantly influences execution time. We compare two matching sequences, in Fig. 1b: $\{p_0, p_1, p_2, p_3\}$ and $\{p_0, p_2, p_3, p_1\}$. The latter sequence identifies a wedge in the data graph after matching the first three vertices, whereas the former finds a triangle. Triangles necessitate a fully connected trio of vertices, unlike wedges, which require only a shared vertex between two edges. Given real-world graphs' sparsity, wedges are more prevalent than triangles, making triangle formation a more stringent criterion for pruning candidate vertices.

3.4.1 Dryadic Dryadic establishes matching order by analyzing the pattern graph, generating various enumerations that differ in their sequence of set operations (intersection and difference) based on vertex order. These enumerations are evaluated excluding symmetric cases (automorphisms), converting each to an execution plan. An analytical model approximates the cost of each matching order using the $G(n, p)$ randomized graph model, with the best order found by ranking (based on intermediate array size) the outcomes.

3.4.2 VF3 VF3 integrates characteristics of both the pattern and data graphs, estimating the likelihood of matching data graph vertices to pattern graph vertices. This process involves computing label occurrence probabilities, as well as the indegree and outdegree ratios between the graphs [12]. By independently evaluating these factors and combining their probabilities, VF3 prioritizes vertices with the lowest match likelihood for early matching attempts. This strategy aims to minimize the number of vertices considered, enhancing matching efficiency.

3.4.3 Implementation Changes We modified Dryadic to use the same matching order as VF3.

3.4.4 Results Discussion This new version is referred to in our experiments as 'recursive dryadic + BS + ONG + vf3 matching order' (mat) and is compared to the previous version 'recursive dryadic + BS + ONG' (vector) (shown in Appendix B). In our results, the mat version performed better in a few datasets such as citPa and DBLP while the vector performed better in MiCo and Ork. The mat version performed better in a few datasets such as citPa and DBLP while the vector performed better in MiCo and Ork. Some datasets such as LiJo and YT performed better with different matching order techniques based on different pattern graphs.

Over the geometric mean of all the pattern graphs, mat performed worse than vector for Ork which had a speed degradation of 0.98x. While for all the other datasets, the mat version performed better than the vector version with speedup of 1.26x for DBLP, 1.01x for Mico, 1.04x for YT, for 1.13x for CitPa and 1.06x for LiJo. VF3's matching order considers both pattern and data graph. It is able to prune the candidate vertices better with its matching order.

3.5 Code Motion

3.5.1 Dryadic In § 2.3, we presented a simplified description of Dryadic in Algorithm 3 but the generated program may lead to substantial computation redundancy. Consider the operation $N(a, 3) \cap N(b, 3)$ in Line 7. This operation is performed *once* in Line 7 to compute the set of vertices variable d will iterate over. However, the identical $N(a, 3) \cap N(b, 3)$ computation is repeated for each value of the c variable in the for loop of Line 6. The redundancy can be removed by moving the $N(a, 3) \cap N(b, 3)$ computation to before the for loop of Line 6. This technique to avoid redundant computations called *code motion* is incorporated in Dryadic, with the revised implementation of Algorithm 3 shown in Algorithm 1.

As before, this is based on the pattern graph of Fig. 1b and assumes pattern vertices will be matched in the order (p_0, p_1, p_2, p_3). Also, as before, the algorithm consists of nested for-loops in Lines 5, 9, 12 and 14, but these are interspersed with set computations to remove redundant computations. The code generation is careful not to perform unnecessary computations (e.g., $N(a,1) \cap N(b,1)$).

Algorithm 1 Dryadic CM

```
1:  Input Data Graph D = (V_d, E_d), Order PO = (p_0, p_1, p_2, p_3)
2:  Output All bijections f
3:  procedure processGraph(g)
4:      L0 ← L(0)
5:      for all a ∈ L0 do
6:          y1l1 ← N(a, 1)
7:          y1l2 ← N(a, 2)
8:          y1l3 ← N(a, 3)
9:          for all b ∈ y1l1 do
10:             y1y2l3 ← y1l3 ∩ N(b, 3)
11:             y1y2l2 ← y1l2 ∩ N(b, 2)
12:             for all c ∈ y1y2l2 do
13:                 y1y2n3l3 ← y1y2l3 − N(c, 3)
14:                 for all d ∈ y1y2n3l3 do
15:                     output {(p_0, a), (p_1, b), (p_2, c), (p_3, d)}
```

Observe specifically that Lines 8 and 10 together perform the $N(a,3) \cap N(b,3)$ partial computation corresponding to Line 7 from Algorithm 3.

3.5.2 VF3 VF3 checks candidate data vertex when required (i.e. IsFeasiblePair()) and doesn't store any intermediate results.

3.5.3 Implementation Changes Code motion was disabled by modifying the generated plans, in Dryadic. These generated plans are read by the recursive program as input and converted to internal representation. This representation is used by the recursive algorithm to perform subgraph isomorphism.

3.5.4 Results Discussion We compared the recursive version of Dryadic with a computation pattern similar to (Algorithm 3) vs previous section (§ 3.4) with code motion disabled (Algorithm 1). We compared between 'recursive dryadic + BS + ONG + VF3 MAT' vs 'recursive Dryadic BS + ONG + VF3 + No CM' (No code Motion) (Appendix B). The results are mixed with 'recursive dryadic + BS + ONG + VF3 MAT' performing better than 'recursive Dryadic BS + ONG + VF3 + NO CM' on four data graphs (MiCo, YT, LiJo, and Ork) and worse on two (DBLP, and CitPa). The geometric mean of speedups for 'recursive dryadic + BS + ONG + VF3 MAT' over 'recursive Dryadic BS + ONG + VF3 + No CM' over the ten pattern graphs are as follows: MiCo (1.35x), YT (1.14x), LiJo (1.43x), and Ork (2.01x), while 'recursive Dryadic BS + ONG + VF3 + No CM' performs better in DBLP (1.07x) and CitPa (1.16x). The results reflect that while pre-computed set operations have the potential to remove computation redundancy, they may incur unnecessary computation. For e.g., when the set $y1y2l2$ in line 12 of Algorithm 1 is empty, the operation to compute $y1y2l3$ in line 10 is useless.

3.6 Discussion

In the preceding sections, we started with Dryadic and made a series of 5 modifications to align it with VF3. The resulting Dryadic implementation 'recursive dryadic + BS + ONG + vf3 matching order' (MO), done to align Dryadic with VF3, was compared with VF3, as shown in Fig. 3d. We see that 'recursive dryadic + BS + ONG + VF3 MAT + No CM' beats 'vf3' in each case and sometimes as large as 10x (e.g., Ork). We believe that the remaining difference in the performance of the two approaches can be explained as follows,

3.6.1 Spatial locality The sorted array is fundamental to Dryadic's efficiency, enabling quick set operations through batched computations that leverage spatial locality by combining results into a single array. In contrast, VF3 iterates over a data graph's neighboring vertices with the NextPair(p_i, d_i) function, requiring a subsequent call to IsFeasiblePair(p_i, d_i) to confirm matches. This recursive approach might lead to cache eviction for the neighbor list of d_i, necessitating data re-loading for subsequent NextPair(p_i, d_i) calls. Conversely, Dryadic's cache-aware intermediate set operations prevent unnecessary data eviction, offering a more efficient computation compared to VF3's method involving IsFeasiblePair() and AddPair().

3.6.2 Better pruning of search space Dryadic effectively prunes candidate vertices from the data graph more efficiently than VF3, due to its intermediate representation. As Dryadic progresses in mapping vertices, the size of intermediate arrays reduces, enhancing pruning efficiency (discussed in § 3.1). For instance, when mapping (p_0, d_3), (p_1, d_1), (p_2, d_2) (Fig. 1) with p_3 as the next vertex, Dryadic's set operations (intersection and difference) pinpoint d_0 as the sole candidate. Conversely, VF3, iterating over d_1's neighbors, initially considers d_0, d_6 based on label matches. However, d_6 is eliminated after failing the feasibility check. This example demonstrates Dryadic's superior pruning capability. When applied to larger graphs, the disparity in the algorithms' search spaces becomes even more pronounced.

Thus, we attribute Dryadic's superior performance to the better graph representation, spatial locality, search-space pruning, and due to the generation of query-specific executables. Whereas, the matching order and code motion optimizations, while beneficial, do not significantly impact performance.

4 Improving Both Systems

4.1 Dryadic Improvements

4.1.1 Lazy Evaluation Even though code motion avoids redundant computations, it sometimes performs pre-computations that are not used: Suppose $y1y2l2$ computed on Line 11 in Algorithm 1 is the empty set. Then, the for loop of Line 12 is not executed and the result of $y1y2l3$ (computed on Line 10) is not used. This makes the computation of $y1y2l3$ wasteful. We address this by using *lazy evaluation* as shown in Algorithm 2. This is an approach similar to memoization in dynamic programming that only performs a computation if the results are needed, and having done so, stores the results in case they are needed again later. This is different from [37], in the fact that Graphmini uses an auxiliary graph data structure to prune the search space, while we are using lazy evaluation to avoid unnecessary computation.

Algorithm 2 has a similar structure (four nested for-loop) to that of Algorithm 1. Each intermediate result is stored in a variable which is only calculated when it is required. The algorithm uses the

'OP' class to separate initialization from computation. Code with set operations is replaced with 'OP' class constructors. (For example, $y1l3 \cap \mathrm{N}(b,3)$ is replaced by $\mathrm{OP}(\cap, y1l3, \mathrm{N}(b,3))$.) The OP class stores references to the inputs in variables mA and mB. It also stores the type and results of operation in mT and mR respectively. When the result of an operator is required, then the Compute() method is called. Compute() returns the result if it is already computed or (if not) computes, stores and returns the result.

Algorithm 2 Dryadic Lazy

```
1:  Input Data Graph D = (V_d, E_d), Order PO = (p_0, p_1, p_2, p_3)
2:  Output All bijections f
3:  procedure OP::OP(type, a, b)
4:      storemA ← &a                          ▷ stores reference
5:      storemB ← &b                          ▷ stores reference
6:      storemR ← ∅                           ▷ stores result
7:      storemT ← type                        ▷ value: ∩ or −
8:      return this
9:  procedure OP::COMPUTE
10:     if mR ≠ ∅ then return mR
11:     if mT = ∩ then
12:         return mR ← mA.COMPUTE() ∩ mB.COMPUTE()
13:     return mR ← mA.COMPUTE() − mB.COMPUTE()
14: procedure OP::LOOP
15:     return this.COMPUTE()
16: procedure PROCESSGRAPH(g)
17:     L0 ← L(0)
18:     for all a ∈ L0.LOOP() do
19:         y1l1 ← N(a, 1)
20:         y1l2 ← N(a, 2)
21:         y1l3 ← N(a, 3)
22:         for all b ∈ y1.LOOP() do
23:             y1y2l3 ← OP(∩, y1l3, N(b, 3))
24:             y1y2l2 ← OP(∩, y1l2, N(b, 2))
25:             for all c ∈ y1y2l2.LOOP() do
26:                 y1y2n3l3 ← OP(−, y1y2l3, N(c, 3))
27:                 for all d ∈ y1y2n3l3.LOOP() do
28:                     output {(p_0, a), (p_1, b), (p_2, c), (p_3, d)}
```

Consider the data graph (Fig. 1a) and pattern graph (Fig. 1b). For the matching order p_0, p_1, p_2, p_3, we have already mapped $\{(p_0, d_3), (p_1, d_1), (p_2, d_2)\}$. At this point, our program is about to execute Line 27. When $y1y2n3l3$.LOOP() is called, Compute() will be called on the object. Since this is the first time, the function is called on an uninitialized object mR. The $y1y2n3l3$.Compute() requires its inputs ($y1y2l3$ and $\mathrm{N}(c,3)$). As $y1y2l3$.Compute() is also called for the first time, its result is also uninitialized. So, $y1y2l3$ will first compute the intersection ($mT = \cap$) between $y1l3$ and $\mathrm{N}(b,3)$. The result $\{d_0\}$ is stored in mR and returned. Next, $y1y2n3l3$.Compute() computes the difference ($mT = -$) between $y1y2l3$ and $\mathrm{N}(c,3)$. The result of this operation ($y1y2n3l3$.Compute()) is also $\{d_0\}$. So, $d = d_0$.

Later, we have mapping $\{(p_0, d_3), (p_1, d_1), (p_2, d_7)\}$, and our program is about to execute Line 27: Similarly to before $y1y2n3l3$.LOOP() is called which calls Compute(). The Compute() requires results from $y1y2l3$ and $\mathrm{N}(c,3)$. Since the result for $y1y2l3$ is already populated, we do not need to re-compute the intersection operation. The difference operation has to be calculated again (its result is again $\{d_0\}$). This technique helps us reuse the calculated set operations.

Now, consider mapping $\{(p_0, d_4), (p_1, d_5)\}$, with the program about to execute Line 25. The result for $y1y2l2$ is $\emptyset$ and the program does not iterate over the loop at Line 25. Therefore, the operation at Line 10 is never executed. So, we will not calculate set operations unless they are required.

4.1.2 Performance Analysis Fig. 4a shows all three executable versions of Dryadic, with and without code motion along with lazy evaluation. Let's first compare 'dryadic' and 'no cm' versions. These versions are similar to disabling code motion as mentioned in § 3.5. The geometric mean of speedups for 'dryadic' over 'no cm' over the ten pattern graphs are as follows: MiCo (1.48x), YT (1.21x), LiJo(1.59x) and Ork (2.12x), while 'no cm' outperforms 'dryadic' in DBLP (1.09x) and CitPa (1.19x). These results are similar to the recursive versions discussed previously.

Lazy evaluation ('lazy') simultaneously eliminates the drawbacks of both 'dryadic' and 'no cm.' Thus, 'lazy' can be seen (Fig. 4a) to be consistently faster than both versions. Comparing 'lazy' to the *better* of 'dryadic' and 'no cm' gives the following speedups: 2.12x (DBLP), 1.66x (CitPa) ('lazy' wrt 'no cm'), and 1.60x (MiCo), 1.91x (YT), 1.52x (LiJo), 1.23x (Ork) (wrt 'dryadic').

4.2 VF3 Improvements

Recall (from § 2.4) that VF3 performs an edge-absence check in IsFeasiblePair(p_i, d_i). This considers edges from the most recently matched data vertex (d_i) to previously matched vertices in the matched data subgraph; and verifies that each such edge has a corresponding edge in the matched pattern subgraph. VF3 implements this by iterating over the adjacency list of d_i and for each vertex in the adjacency list, determining whether it belongs to the data subgraph in $O(1)$ time. (VF3 maintains an array indexed on data vertices that is initialized to NULL. If a data vertex is matched, the array entry contains the corresponding pattern vertex.) If the vertex does belong to the data subgraph, it verifies that the pattern subgraph contains the corresponding edge.

Typically, one would expect the degree of a vertex in the data graph to be much larger than the number of vertices in the pattern graph. For example, in our data graphs, the average maximum degree of any data vertex over the six data graphs is 26K (the maximum degree overall is 66K in Orkut). The average degree of vertices in DBLP, MiCo, YT, CitPa, LiJo, and Ork is 3.31, 11.17, 2.63 4.37, 8.67 and 38.14 respectively. In contrast, our pattern graphs have at most six vertices (so the number of already mapped vertices is at most five). In these situations, VF3 searches a very large adjacency list for a handful of mapped data vertices.

We modified the implementation of VF3 to first identify the (at most five) already-mapped data vertices from a mapping table (e.g., Table 5). For each such data vertex x, we perform a binary search for x in d_i's adjacency list (recall that the adjacency list is implemented as a sorted array). If x is found, we must confirm that the pattern subgraph has the corresponding edge.

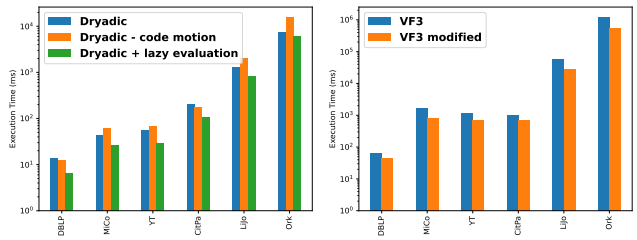
Example: In Fig. 1a, suppose we have mapped $\{(p_0, d_3), (p_1, d_1)\}$ and are executing IsFeasiblePair(p_2, d_7). The original VF3 algorithm iterates over *all of the neighbors* of d_7 (i.e., d_1, d_3, d_4, d_6) and

Table 5: Mapped vertices storage

indices	0	1	2	3
patt	p_0	p_1	p_2	p_3
data	d_3	d_1	d_7	d_0

checks if they were already mapped. In the modification, we maintain a list of mapped pairs and only iterate over them (i.e. d_3 and d_1). We check whether these vertices are neighbors of d_7 (via two binary searches on d_7's adjacency list). If yes, then the edge-absence step proceeds as before.

The number of matched pairs increases as VF3 goes deeper into the matching tree. In Fig. 1c, d_3 and d_4 are the roots of the tree. When d_3 and d_4 are mapped, there are no previous vertices to consider. When d_1 or d_6 are being mapped at depth 1, VF3 only has to check if edges to d_3 and d_4 exist. For the matching pairs in Table 5, Original VF3 explores the neighboring vertices of each data vertex in the table $\{5, 5, 4, 4\}$ number of times. Modified VF3 explores neighboring data vertices $\{0, 1, 2, 3\}$ number of times (same as depth of matching pair).



(a) Dryadic **(b) VF3**

Figure 4: Benefits of optimization

VF3 performed better with this optimization (Fig. 4b). Iterating over neighbors of the data vertex is more expensive than iterating over matched vertices and checking if they are the neighbors of the data vertex. We saw up to 1.46x improvement from the CitPa dataset in latency. For other data graphs DBLP, MiCo, YT, LiJo, and Ork, we saw improvement by 1.45x, 1.96x, 1.67x, 2.08x, and 2.36x respectively. We note that the modification may not always outperform the original VF3. Let vertex d_i have degree n and let k denote the number of matched pairs. The complexity of the edge-absence check in original VF3 is $O(n)$ while modified VF3 has complexity $k \log n$. When $k << n$ (as in our benchmarks), modified VF3 is superior. However, when k and n are comparable, the original VF3 is expected to perform better. Although this scenario is not expected in real-world benchmarks, the edge-absence check can be implemented using a hybrid strategy that switches between the two approaches based on an empirical comparison of k and n.

5 Related Work

There are many approaches for subgraph pattern matching, the oldest is Ullmann's algorithm [59], but it doesn't exploit pruning and hence does not scale well for larger graphs. GraphQL [22], TurboIso [21], CFL [7] use candidate set-based filtering to prune the search space for subgraph matching. EmptyHeaded [1] and GraphFlow [28] use intersection set operations to generate candidate sets. As these do not handle missing edges, we consider Dryadic [38] which is a similar algorithm that comes under this category. More works for filtering are [20, 30, 45, 55]. On the other hand, VF2++ [27], SPath [70], QuickSi [48], and RI [8] generate candidate set from the neighbors of matched vertices and pass them through filters. This approach is similar to VF3 [12], which is an advanced version of VF2 with multiple sophisticated optimizations. Pruning can also be done by setting up constraints on the vertices and edges and then stopping the search when the constraints are violated [5, 23]. Apart from these methods, historical data can also be used to prune the search space [24]. Another way is to use effective graph editing or splitting of graphs to reduce the search space [25, 37].

GraphIt [68] is a domain-specific language (DSL) for graph analytics. GG [9] is an extension of GraphIT that achieves high performance on both CPUs and GPUs. Compilation-based algorithms reduce the burden on programmers as the system can perform optimizations like tiling on the graph [67] to increase cache locality. Dryadic uses such optimizations specialized for subgraph pattern matching. Other methods that are specifically developed to improve sub-graph matching on a specific hardware platform, such as FPGA [51], GPU [3, 56], but that is not the focus of this paper.

Different systems propose various methods to generate matching orders. A good heuristic for matching order can reduce the exploration space. Dryadic ranks the enumerated matching orders based on an analytical model and chooses the best one. Instead, VF3 uses both pattern and data graphs to generate probabilities of the least possible matches and a matching order based on the estimate. Kim et al. [29] use static equivalences of vertices to generate matching orders and dynamic equivalences of vertices to prune redundant search space. Sun et al. [54] compare the matching order generation methods of different systems.

There are different implementations of shared-memory parallel algorithms for subgraph isomorphism [1, 11, 13, 18, 26, 28, 31, 40, 44, 52, 53, 69]. They typically focus on optimized pruning [6], load balancing [60], or compilation based algorithms to operate on subpatterns [14]. Subgraph isomorphism is widely researched in distributed systems [2, 4, 6, 32–34, 41–43, 46, 47, 49, 50, 57, 58, 61, 65] by considering communication and computation constraints, a good e.g., being a comprehensive study [34] of various matching algorithms implemented in a generic distributed computing framework.

6 Conclusion

In this paper, we compared two state-of-the-art subgraph pattern matching systems, VF3 and Dryadic, through comprehensive experiments. They represent two of the most popular implementation approaches of the same algorithmic strategy for subgraph pattern matching. However, Dryadic is roughly two orders of magnitude faster than VF3. We implemented multiple variants of Dryadic to understand the reasons for the gigantic performance gap between the two systems. Based on the understanding, we improved Dryadic via lazy evaluation and VF3 by narrowing the search scope for connectivity checks. We tested the single-threaded version and will test the multi-threaded version in future work.

References

[1] Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS) 42*, 4 (2017), 20.

[2] Afrati, F. N., Fotakis, D., and Ullman, J. D. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (Piscataway, NJ, USA, 2013), IEEE, Insitute of Electrical and Electronics Engineers, pp. 62–73.

[3] Ahmad, A., Yuan, L., Yan, D., Guo, G., Chen, J., and Zhang, C. Accelerating k-core decomposition by a gpu. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)* (Piscataway, NJ, USA, 2023), IEEE, Insitute of Electrical and Electronics Engineers, pp. 1818–1831.

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

[5] Arai, J., Fujiwara, Y., and Onizuka, M. Gup: Fast subgraph matching by guard-based pruning. *Proc. ACM Manag. Data 1*, 2 (jun 2023).

[6] Bhattarai, B., Liu, H., and Huang, H. H. CECI: compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (New York, NY, USA, 2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 1447–1462.

[7] Bi, F., Chang, L., Lin, X., Qin, L., and Zhang, W. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (New York, NY, USA, 2016), F. Özcan, G. Koutrika, and S. Madden, Eds., ACM, pp. 1199–1214.

[8] Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics 14*, 7 (2013), 1–13.

[9] Brahmakshatriya, A., Zhang, Y., Hong, C., Kamil, S., Shun, J., and Amarasinghe, S. P. Compliation techniques for graphs algorithms on gpus. *CoRR abs/2012.07990* (2020).

[10] Carletti, V., Foggia, P., Greco, A., Vento, M., and Vigilante, V. Vf3-light: a lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters 125* (2019), 591–596.

[11] Carletti, V., Foggia, P., Ritrovato, P., Vento, M., and Vigilante, V. A parallel algorithm for subgraph isomorphism. In *Graph-Based Representations in Pattern Recognition: 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19–21, 2019, Proceedings 12* (Berlin Heidelberg, 2019), Springer, Springer, pp. 141–151.

[12] Carletti, V., Foggia, P., Saggese, A., and Vento, M. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE transactions on pattern analysis and machine intelligence 40*, 4 (2017), 804–818.

[13] Chen, H., Liu, M., Zhao, Y., Yan, X., Yan, D., and Cheng, J. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[14] Chen, J., and Qian, X. Dwarvesgraph: A high-performance graph mining system with pattern decomposition, 2020.

[15] Chen, J., and Qian, X. Kudu: An efficient and scalable distributed graph pattern mining engine. *arXiv preprint arXiv:2105.03789* (2021).

[16] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), Association for Computing Machinery, pp. 151–158.

[17] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell. 26*, 10 (2004), 1367–1372.

[18] Dias, V., Teixeira, C. H. C., Guedes, D., Meira, W., and Parthasarathy, S. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1357–1374.

[19] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB 7*, 7 (2014), 517–528.

[20] Han, M., Kim, H., Gu, G., Park, K., and Han, W. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (New York, NY, USA, 2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 1429–1446.

[21] Han, W., Lee, J., and Lee, J. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (New York, NY, USA, 2013), K. A. Ross, D. Srivastava, and D. Papadias, Eds., ACM, pp. 337–348.

[22] He, H., and Singh, A. K. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), Association for Computing Machinery, pp. 405–418.

[23] Jamshidi, K., Mariappan, M., and Vora, K. Anti-vertex for neighborhood constraints in subgraph queries. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (New York, NY, USA, 2022), Association for Computing Machinery, pp. 1–9.

[24] Jian, X., Li, Z., and Chen, L. Suff: Accelerating subgraph matching with historical data. *Proc. VLDB Endow. 16*, 7 (mar 2023), 1699–1711.

[25] Jiang, Z., Zhang, S., Liu, B., Hou, X., Yuan, M., and You, H. Fast subgraph matching by dynamic graph editing. *IEEE Transactions on Services Computing* (2023), 1–12.

[26] Jin, X., Yang, Z., Lin, X., Yang, S., Qin, L., and Peng, Y. Fast: Fpga-based subgraph matching on massive graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (Piscataway, NJ, USA, 2021), IEEE, Insitute of Electrical and Electronics Engineers, pp. 1452–1463.

[27] Jüttner, A., and Madarasi, P. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics 242* (2018), 69–81.

[28] Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., and Salihoglu, S. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), ACM, Association for Computing Machinery, pp. 1695–1698.

[29] Kim, H., Choi, Y., Park, K., Lin, X., Hong, S.-H., and Han, W.-S. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 925–937.

[30] Kim, K., Seo, I., Han, W., Lee, J., Hong, S., Chafi, H., Shin, H., and Jeong, G. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (New York, NY, USA, 2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 411–426.

[31] Kimmig, R., Meyerhenke, H., and Strash, D. Shared memory parallel subgraph enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Piscataway, NJ, USA, 2017), IEEE, Insitute of Electrical and Electronics Engineers, pp. 519–529.

[32] Lai, L., Qin, L., Lin, X., and Chang, L. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment 8*, 10 (2015), 974–985.

[33] Lai, L., Qin, L., Lin, X., Zhang, Y., and Chang, L. Scalable distributed subgraph enumeration. *PVLDB 10*, 3 (2016), 217–228.

[34] Lai, L., Qing, Z., Yang, Z., Jin, X., Lai, Z., Wang, R., Hao, K., Lin, X., Qin, L., Zhang, W., Zhang, Y., Qian, Z., and Zhou, J. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow. 12*, 10 (2019), 1099–1112.

[35] Leskovec, J., Kleinberg, J., and Faloutsos, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (New York, NY, USA, 2005), Association for Computing Machinery, pp. 177–187.

[36] Leskovec, J., and Sosič, R. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST) 8*, 1 (2016), 1.

[37] Liu, J., Polisetty, S., Guan, H., and Serafini, M. Graphmini: Accelerating graph pattern matching using auxiliary graphs. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Piscataway, NJ, USA, 2023), IEEE, Insitute of Electrical and Electronics Engineers, pp. 211–224.

[38] Mawhirter, D., Reinehr, S., Han, W., Fields, N., Claver, M., Holmes, C., McClurg, J., Liu, T., and Wu, B. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Piscataway, NJ, USA, 2021), IEEE, Insitute of Electrical and Electronics Engineers, pp. 289–303.

[39] Mawhirter, D., and Wu, B. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), ACM, Association for Computing Machinery, pp. 509–523.

[40] Mhedhbi, A., and Salihoglu, S. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow. 12*, 11 (2019), 1692–1704.

[41] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (New York, NY, USA, 2013), Association for Computing Machinery, pp. 439–455.

[42] Plantenga, T. Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing 73*, 2 (2013), 164–175.

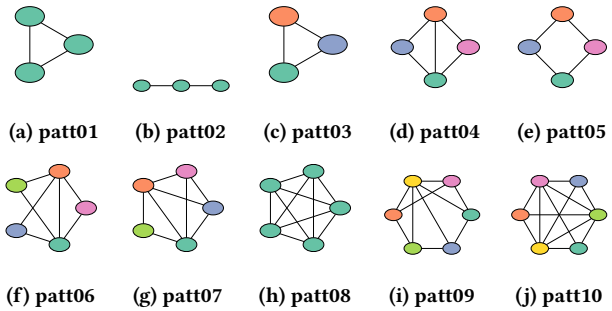
[43] Qiao, M., Zhang, H., and Cheng, H. Subgraph matching: on compression and computation. *PVLDB 11*, 2 (2017), 176–188.

[44] Raman, R., van Rest, O., Hong, S., Wu, Z., Chafi, H., and Banerjee, J. Pgx.iso: Parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems* (New York, NY, USA, 2014), GRADES'14, Association for Computing Machinery, p. 1–6.

[45] Ren, X., and Wang, J. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment 8*, 5 (2015), 617–628.
[46] Ren, X., Wang, J., Han, W.-S., and Yu, J. X. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment 12*, 11 (2019), 1344–1356.
[47] Serafini, M., De Francisci Morales, G., and Siganos, G. Qfrag: distributed graph search via subgraph isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, Association for Computing Machinery, p. 214–228.
[48] Shang, H., Zhang, Y., Lin, X., and Yu, J. X. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow. 1*, 1 (2008), 364–375.
[49] Shao, Y., Cui, B., Chen, L., Ma, L., Yao, J., and Xu, N. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (New York, NY, USA, 2014), C. E. Dyreson, F. Li, and M. T. Özsu, Eds., ACM, pp. 625–636.
[50] Shi, T., Zhai, M., Xu, Y., and Zhai, J. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2020), Insitute of Electrical and Electronics Engineers, pp. 1–14.
[51] Su, X., Lin, Y., and Zou, L. Fasi: Fpga-friendly subgraph isomorphism on massive graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)* (Piscataway, NJ, USA, 2023), IEEE, Insitute of Electrical and Electronics Engineers, pp. 2099–2112.
[52] Sun, S., Che, Y., Wang, L., and Luo, Q. Efficient parallel subgraph enumeration on a single machine. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019* (Piscataway, NJ, USA, 2019), IEEE, pp. 232–243.
[53] Sun, S., and Luo, Q. Parallelizing recursive backtracking based subgraph matching on a single machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)* (Piscataway, NJ, USA, 2018), IEEE, Insitute of Electrical and Electronics Engineers, pp. 1–9.
[54] Sun, S., and Luo, Q. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Piscataway, NJ, USA, 2020), Insitute of Electrical and Electronics Engineers, pp. 1083–1098.
[55] Sun, S., and Luo, Q. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering 34*, 1 (2020), 491–505.
[56] Sun, X., and Luo, Q. Efficient gpu-accelerated subgraph matching. *Proc. ACM Manag. Data 1*, 2 (jun 2023).
[57] Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment 5*, 9 (2012).
[58] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 425–440.
[59] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM 23*, 1 (1976), 31–42.
[60] Vora, K., Xu, G., and Gupta, R. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 507–522.
[61] Wang, Z., Gu, R., Hu, W., Yuan, C., and Huang, Y. BENU: distributed subgraph enumeration with backtracking-based framework. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019* (Piscataway, NJ, USA, 2019), Insitute of Electrical and Electronics Engineers, pp. 136–147.
[62] Willett, P. Chemoinformatics: a history. *Wiley Interdisciplinary Reviews: Computational Molecular Science 1*, 1 (2011), 46–56.
[63] Willey, L. C., and Salmon, J. L. A method for urban air mobility network design using hub location and subgraph isomorphism. *Transportation Research Part C: Emerging Technologies 125* (2021), 102997.
[64] Yang, J., and Leskovec, J. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems 42*, 1 (2015), 181–213.
[65] Yang, Z., Lai, L., Lin, X., Hao, K., and Zhang, W. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 2049–2062.
[66] Zhang, S., Li, S., and Yang, J. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (New York, NY, USA, 2009), EDBT '09, Association for Computing Machinery, p. 192–203.
[67] Zhang, Y., Kiriansky, V., Mendis, C., Amarasinghe, S., and Zaharia, M. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)* (Piscataway, NJ, USA, 2017), IEEE, Insitute of Electrical and Electronics Engineers, pp. 293–302.
[68] Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S. P. Graphit: a high-performance graph DSL. *PACMPL 2*, OOPSLA (2018), 121:1–121:30.
[69] Zhao, C., Zhang, Z., Xu, P., Zheng, T., and Guo, J. Kaleido: An efficient out-of-core graph mining system on a single machine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (Piscataway, NJ, USA, 2020), Insitute of Electrical and Electronics Engineers, pp. 673–684.
[70] Zhao, P., and Han, J. On graph query optimization in large networks. *Proc. VLDB Endow. 3*, 1 (2010), 340–351.

A Pattern Graphs

The 10 pattern graphs used in the paper.



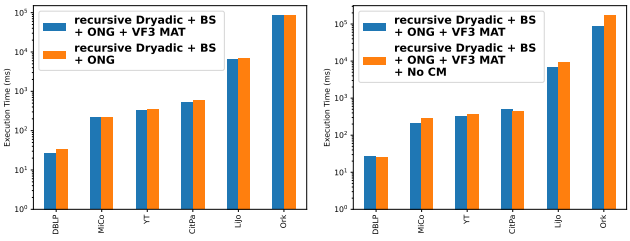
(a) patt01 **(b) patt02** **(c) patt03** **(d) patt04** **(e) patt05**

(f) patt06 **(g) patt07** **(h) patt08** **(i) patt09** **(j) patt10**

Figure 5: Pattern Graphs. Each distinct color represents a unique label.

B Additional results

In this section, we present the results that are omitted in the paper due to space constraints. Specifically we show the results for the impact that the VF3 matching order has on latency, by adding matching order to 'recursive dryadic + BS + ONG' (§ 3.4). As can be seen in Fig. 6a, the matching order does not make a significant difference. In Fig. 6b, we show the impact of code motions in latency (discussed in § 3.5), which varies depending on the data graph and pattern graph considered.



(a) Different matching orders. **(b) Dryadic rec. w/o code motion.**

Figure 6: Rest of graphs quantifying the performance of Dryadic versions.

C Dryadic

Algorithm 3 shows the nested-for-loop code for the pattern graph in Fig. 1b based on the pattern vertex matching order (p_0, p_1, p_2, p_3). Thus, the outermost for loop corresponds to p_0, the first vertex in

the pattern order and the innermost for loop corresponds to p_3, the last vertex in the pattern order. In the algorithm, $L(l)$ returns all vertices in the data graph with label l and $N(v, l)$ returns all neighboring vertices of v in the data graph with label l. Set operations are determined by the relationship of the new pattern vertex with previous vertices in the pattern order. A set *intersection* is used when an edge is *present* and a set *difference* when an edge is *absent*.

Algorithm 3 Set operation-based matching in Dryadic.

```
1: Input Data Graph D = (V_d, E_d), Order PO = (p_0, p_1, p_2, p_3)
2: Output All bijections f
3: procedure processGraph(g)
4:     for all a ∈ L(0) do
5:         for all b ∈ N(a, 1) do
6:             for all c ∈ N(a, 2) ∩ N(b, 2) do
7:                 for all d ∈ N(a, 3) ∩ N(b, 3) − N(c, 3) do
8:                     output {(p_0, a), (p_1, b), (p_2, c), (p_3, d)}
```

The first vertex in the pattern order is p_0, which has the label 0. The algorithm therefore calls $L(0)$, which returns all data vertices with label 0 (Line 4) and iterates over these vertices using the variable a. The second vertex in the pattern order is p_1, which has label 1 and has an edge to p_0. Thus, Line 5 considers all data vertices with label 1 that are adjacent to a and iterates over them using variable b. The third vertex in the pattern order is p_2 which has label 2 and is adjacent to both p_0 and p_1. Thus, in Line 6, we use variable c to iterate over vertices in the data graph that are adjacent to both a and b and have label 2. The last vertex in the pattern order is p_3, which has label 3 and is connected to p_0 and p_1 but not p_2. Hence, we see our first use of the difference operation. Line 7 subtracts the neighbors of a and b from the neighbors of c (all with label 3) and iterates over these vertices using d. Finally, inside the innermost loop, the algorithm outputs a matching.

We illustrate the algorithm (Algorithm 3) on the data graph (Fig. 1a). Line 4 iterates a over all data vertices with label 0 ($L(0) = \{d_3, d_4\}$). With $a = d_3$, Line 5 sets b to d_1 ($N(d_3, 1) = \{d_1\}$). Next, Line 6 computes the intersection $N(d_3, 2) \cap N(d_1, 2)$, which gives $\{d_2, d_7\}$. With $c = d_2$, Line 7 subtracts $N(d_2, 3)(\emptyset)$ from $N(d_3, 3)(\{d_0\}) \cap N(d_1, 3)(\{d_0, d_6\})$, giving $\{d_0\}$. d is set to d_0 and the first match is the output: $\{(p_0, d_3), (p_1, d_1), (p_2, d_2), (p_3, d_0)\}$. In the next iteration, c is set to d_7 on Line 6. The intersection $N(d_3, 2)(\{d_0\}) \cap N(d_1, 2)(\{d_0\})$ is computed and followed by subtracting $(d_7, 2)(\emptyset)$. This leads to the second match: $\{(p_0, d_3), (p_1, d_1), (p_2, d_7), (p_3, d_0)\}$. After exhausting the inner loops, the system next iterates on Line 4 with $a = d_4$. Line 5 sets b to d_5. Line 6 computes the intersection of $N(d_4, 2)(\{d_7\})$ and $N(d_5, 2)(\emptyset)$, which is $\emptyset$. Thus, the for loop on Line 6 is not executed and no further matches are obtained. The matching paths mentioned in the above procedure are shown in Fig. 1c.

D VF3

Algorithm 4 utilizes the following functions.

IsGoal() returns true if all pattern vertices are mapped. IsDead() analyzes edges leaving the pattern and data subgraphs of state s and returns "true" if this analysis shows that any matches that build on s will surely fail. In this case, the recursion is terminated.

Algorithm 4 Backtracking-based matching algorithm in VF3.

```
1: Input State s
2: Output (v_p, v_d) ∈ M where v_p ∈ V_p and v_d ∈ V_d
3: procedure processGraph(s)
4:     if s.IsGoal() then
5:         output s.matches
6:         return
7:     if s.IsDead() then    return
8:     p_i ← d_i ← NULL
9:     while s.NextPair(p_i, d_i) do
10:        if s.IsFeasiblePair(p_i, d_i) then
11:            s_new ← s
12:            s_new.AddPair(p_i, d_i)
13:            processGraph(s_new)
```

NextPair(p_i, d_i) returns a candidate vertex d_i from the data graph whose label matches the label of p_i. The pattern vertex p_i is uniquely determined from the matching order and the depth of the recursive call. IsFeasiblePair(p_i, d_i) checks whether the pattern and data subgraphs resulting from adding p_i and d_i, respectively, are isomorphic. This entails two checks: (1) the *edge presence* check ensures that any edge in the pattern subgraph from p_i to a previous pattern vertex p_j has a corresponding edge from d_i to the data vertex that was matched to p_j in an earlier recursive step. (2) the *edge absence* check similarly ensures if there is not an edge (p_i, p_j) in the pattern subgraph, then the corresponding edge from d_i is also absent. Note that the edge absence check is implemented by checking the contrapositive: i.e., if there is an edge from d_i to a previously matched vertex in the data subgraph, then there is a corresponding edge from p_i to the corresponding vertex in the pattern graph.

AddPair(p_i, d_i) adds the matching pair (p_i, d_i) to state s in preparation for the next recursive call.

We illustrate the operation of VF3 using the examples from Fig. 1a and Fig. 1b. The pattern vertex matching order is (p_0, p_1, p_2, p_3), as before. First, NextPair() (Line 9) initializes p_i and d_i to p_0 and d_3. IsFeasiblePair() (Line 10) returns "true" since the data and pattern subgraphs do not yet contain edges. The next state s_{new} is obtained by adding (p_0, d_3) to state s (Line 12) and Line 13 makes a recursive call.

At level 2 of the recursion, p_i is set to p_1 (which has label 1). The only possible candidate vertex for d_i is d_1 (neighbors of d_3 with label 1). So, NextPair() sets d_i to d_1. This time, IsFeasiblePair() also confirms that since $\overline{p_1 p_0}$ exists in the pattern graph, then there must be a corresponding edge $\overline{d_1 d_3}$ in the data graph. Since this is true, the algorithm adds this pair and makes the next recursive call.

At level 3, NextPair() maps p_2 to d_2 (neighbor of d_1). IsFeasiblePair(), cross-checks the pattern subgraph edges from p_2 (i.e., $\overline{p_2 p_1}, \overline{p_2 p_0}$) with corresponding edges from d_2 ($\overline{d_2 d_1}, \overline{d_2 d_3}$). After passing the feasibility test, the algorithm adds a matching pair (p_2, d_2).

At level 4, in NextPair(), p_3 will be mapped and the function will return candidate vertex d_0 first. Inside IsFeasiblePair(), the algorithm will check if all the edges of pattern subgraph with pattern vertex p_3 ($\overline{p_3 p_0}, \overline{p_3 p_1}$) are present in the data subgraph with data vertex d_0 as well ($\overline{d_0 d_3}, \overline{d_0 d_1}$). It also checks that the data subgraph

does *not* contain any edges from d_0 to already mapped vertices in the data subgraph that is *not* present in the pattern subgraph. Since d_0 satisfies both of these conditions, (p_3, d_0) are added.

At level 5, IsGoal() (Line 4) returns true. So, the first output generated will be $\{(p_0, d_3), (p_1, d_1), (p_2, d_2), (p_3, d_0)\}$, after which the the program returns to level 4. At level 4, the next iteration of the while loop considers d_6 as a possible match for p_3 in Line 9. IsFeasiblePair() returns false, as there is no edge from d_6 in the data graph that corresponds to $\overline{p_3 p_0}$.

At level 3, d_7 is mapped to p_2 and a series of recursive calls outputs $\{(p_0, d_3), (p_1, d_1), (p_2, d_7), (p_3, d_0)\}$ and unwinds the recursion.

At level 1, NextPair()(Line 9) maps p_0 to d_4 and at level 2, maps d_5 to p_1. Since d_5 has no neighbors with label 2, the recursion terminates. The program also terminates as there are no other matching paths. The matching tree followed by the algorithm is shown in Fig. 1c.