Contigra: Graph Mining with Containment Constraints

Joanna Che
School of Computing Science
Simon Fraser University
British Columbia, Canada
jca241@cs.sfu.ca

Kasra Jamshidi
School of Computing Science
Simon Fraser University
British Columbia, Canada
kjamshid@cs.sfu.ca

Keval Vora
School of Computing Science
Simon Fraser University
British Columbia, Canada
keval@cs.sfu.ca

Abstract

While graph mining systems employ efficient task-parallel strategies to quickly explore subgraphs of interest (or *matches*), they remain oblivious to *containment constraints* like maximality and minimality, resulting in expensive constraint checking on every explored match as well as redundant explorations that limit their scalability.

In this paper, we develop Contigra for efficient graph mining with containment constraints. We first model the impact of constraints in terms of *dependencies* across exploration tasks, and then exploit the dependencies to develop: (a) *task fusion* that merges correlated tasks to increase cache reuse; (b) *task promotion* that allows explorations to continue from available subgraphs and skip re-exploring subgraphs from scratch; (c) *task cancelations* that avoid unnecessary constraint checking and prioritizes faster constraint validations; and (d) *task skipping* that safely skips certain exploration and validation tasks. Experimental results show that Contigra scale to graph mining workloads with containment constraints, which could not be handled by existing state-of-the-art systems.

CCS Concepts: • **Information systems → Data mining**; **Computing platforms**; • **Computing methodologies → Concurrent computing methodologies**.

Keywords: subgraph exploration, graph mining, maximality, quasi-cliques, nested queries, keyword search, motifs

1 Introduction

Graph mining systems like Peregrine [25], Automine [35], GraphPi [39], and others [4–6, 17, 40, 42] explore subgraphs of interest in large graphs by checking subgraph isomorphisms which is computationally expensive (NP-complete). These systems decompose the subgraph exploration into static, independent *exploration tasks* that traverse through the graph in parallel, finding one subgraph *match* (or embedding) per task at a time. While they support applications like Motif Counting and Frequent Subgraph Mining, complex applications like Maximal Quasi-Cliques and Keyword Search are difficult to run on these systems as they stipulate additional constraints like *maximality* or *minimality*.

Mining with Containment Constraints. We call such constraints *Containment Constraints* since they are in the form of a subgraph being present or absent inside another subgraph. Containment constraints occur in various common graph mining applications. For example, mining Maximal Cliques or Quasi-Cliques [33] requires exploring cliques/quasi-cliques that are not contained inside larger cliques/quasi-cliques. Similarly, queries with Anti-Vertices [26] specify neighborhood constraints, producing subgraphs which are not contained inside a larger subgraph that includes the anti-vertices. This can further be generalized into what we call 'Nested Subgraph Queries' which are important primitives in modern graph query languages (*e.g.*, nested MATCH clause in Cypher/GQL [13, 16]). Finally, Keyword Search [15] is an application with the minimality constraint that aims to find minimal subgraphs with certain specific keywords, *i.e.*, subgraphs that do not contain smaller connected subgraphs with all those keywords.

This paper develops solutions to efficiently support graph mining applications with containment constraints on modern pattern-based graph mining systems. We motivate the scalability challenges in containment constrained applications using the maximal quasi-clique example next.

Scalability Challenges. Consider the problem of finding Maximal Quasi-Cliques which is used in several applications like drug discovery [3, 22], social network analysis [21, 32], and cybersecurity [41, 45]. A quasi-clique is a dense graph structure; it is similar to a clique but it can miss a few edges. Figure 1 shows an example graph and all the quasi-cliques it contains. Notice that while `a-c-d-e` is a valid quasi-clique, it is not maximal since it is contained inside `a-b-c-d-e-i`. Hence, the Maximal Quasi-Clique problem must consider `a-b-c-d-e-i` in its result set and leave `a-c-d-e` out of it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629589

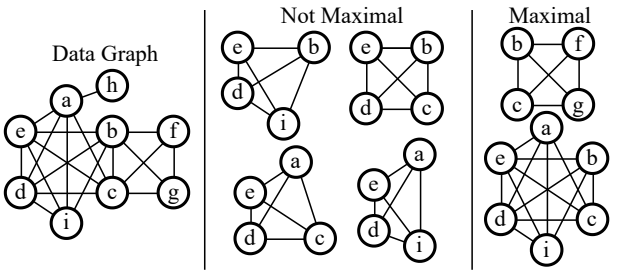


Figure 1. Example data graph and its quasi-cliques with $\gamma = 0.8$. There are five quasi-cliques of size 4, but the ones shown on the left are not maximal due to the size 6 quasi-clique `a-b-c-d-e-i` (shown on the right).

Constraint Checking. Finding maximal quasi-cliques on pattern-based graph mining systems is challenging because the maximality constraint is not specified on the structure of the subgraph alone, *i.e.*, it cannot be directly reduced into finding matches for certain specific quasi-clique patterns. This is visible in our example in Figure 1 where `a-c-d-e` and `f-g-c-b` have the same structure, but only the latter one is maximal. Since exploration tasks in graph mining systems follow static loop schedules (or matching orders), the task exploring `a-c-d-e` can remain independent from the task exploring `a-b-c-d-e-i`, making it difficult to enforce maximality constraint by checking these two matches. The only way to ensure maximality is to examine every individual match *after it has been explored* and ensure it is not contained in a larger matching subgraph. This would require $O(C(n,k))$ matches to be checked (subgraphs of size k in a graph with n vertices), which is inefficient and does not scale on large graphs (*i.e.*, as n grows).

Per-Match Cost. Checking whether a match satisfies maximality is itself not a simple task. Each quasi-clique match can potentially be a part of multiple larger quasi-cliques. For instance in Figure 1 the match `c-d-e` is inside `a-c-d-e` while it is also inside quasi-cliques `b-c-d-e` and `a-c-d-e-i`. This means, given a match for `c-d-e`, checking whether it is maximal would require verifying whether `c-d-e` is contained in any of those quasi-cliques which are being explored by other concurrent tasks. This issue complicates further since maximality checks can span across multiple sizes (*e.g.*, a size k quasi-clique might not be inside any size $k+1$ quasi-clique while still being part of size $k+2$ quasi-clique). Hence, every match must go through multiple checks against matches from other tasks, making the process of satisfying maximality computationally intensive for each match.

Effect on Performance. We verified the scalability bottleneck in exploring Maximal Quasi-Cliques by measuring the time taken to find maximal quasi-cliques in different graphs and comparing it with time taken to only find quasi-cliques (*i.e.*, without maximality constraint). Figure 2 shows the performance for state-of-the-art graph mining systems Peregrine [25] and GraphPi [39]. As we can see, the maximality

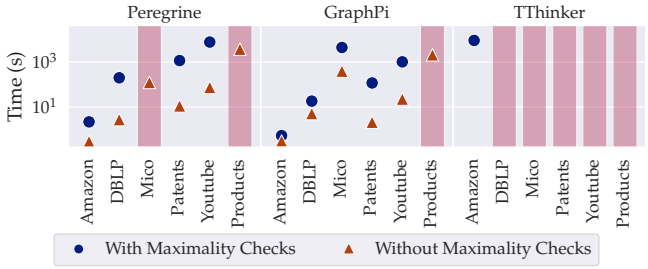


Figure 2. Performance of mining quasi-cliques with and without the maximality checks. Red bars indicate executions with maximality checks that did not complete in 12 hours.

checks often add over an order of magnitude performance penalty compared to executions without those checks. More importantly, the performance difference between exploring just the quasi-cliques compared to exploring maximal quasi-cliques grows as graphs grow large, primarily because the number of matches to be examined increases rapidly; for instance, 453.1 million maximality checks are performed on Patents graph whereas 2.3 billion checks are performed on the larger Youtube graph. Such increase in the number of matches to be checked significantly limits scalability; as seen, both GraphPi and Peregrine fail to complete maximal quasi-cliques on the large Products graph while they finish exploring quasi-cliques without the maximality constraint.

Custom maximal quasi-clique solvers like TThinker [31] and others [18, 33, 38] reduce the number of matches to be checked by pruning the sparse regions of the graph (*i.e.*, reduce n in $O(C(n,k))$ checks). However, similar to previous pattern-based systems, the issue remains: matches are examined for maximality individually *after they are explored* by comparing them with other matches. We measured how TThinker scales in Figure 2; as we can see, it can find maximal quasi-cliques for the one small graph, but it does not finish execution for the remaining five graphs.

The above scalability limitations are present for all graph mining applications with containment constraints. For example, Minimal Keyword Search performs multiple minimality checks for each match against other subgraph matches that are explored by other tasks. Similarly, Nested Subgraph Queries require checking the specified constraints about absence or presence of subgraphs within/around the matches that are explored. Existing graph mining solutions cannot efficiently handle containment constrained graph mining because they examine the explored matches against the required constraints *after those matches have been explored.*

Our Solution. Our goal is to enrich the pattern matching strategies in state-of-the-art graph mining systems to enable graph mining applications with containment constraints. We develop Contigra, a novel execution model for containment constrained graph mining that actively leverages the containment constraints to enforce dependencies across concurrent exploration tasks. By doing so, constraint checking

is performed naturally *during exploration*, hence avoiding expensive checking after matches are explored while also limiting the number of constraint checks and unnecessary subgraph explorations. We make the following contributions.

- We model the effect of containment constraints in terms of dynamic *dependencies* across concurrent exploration tasks (Section 4). We identify three key dependency types based on the different semantics of constraints. These dependencies lay the foundation for task management strategies in Contigra to efficiently explore matches that satisfy all the required containment constraints.
- We develop novel strategies in Contigra to actively validate dependencies during execution. Contigra employs a new kind of task called *validation tasks* that focus on validating constraints by exploring matches containing specific subgraphs (Section 5). While validation tasks are spawned dynamically from exploration tasks, we enable *task fusion* in Contigra to minimize the dynamic overheads of task creation and synchronization as well as leverage the explored subgraph and associated caches for faster validation. Furthermore, Contigra uses *task promotion*, a technique that allows validation tasks to subsequently continue as exploration tasks, hence skipping other exploration tasks that would otherwise re-explore the same subgraphs from scratch.
- We enable Contigra to automatically infer and impose dependencies across validation tasks to capture the dynamic progress of validation during execution, and to *cancel validation tasks* based on the dynamic progress so that unnecessary constraint checking is avoided (Section 6). As different validation tasks involve different amounts of computation, Contigra automatically generates a scheduling order for validation tasks to prioritize quicker validations and higher cancelation of validation tasks.
- We further develop strategies in Contigra to skip certain exploration tasks as well as speed up other exploration tasks by analyzing the constraints across potential matches they would explore. Our analysis buckets exploration tasks into different categories based on different possibilities of constraint violations, using which Contigra either safely *skips exploration tasks*, *skips validation* of constraints, or performs *eager filtering* to actively check constraints (Section 7).
- We demonstrate the effectiveness of techniques by incorporating Contigra in Peregrine, a state-of-the-art graph mining system, and evaluating its performance across multiple containment constrained graph mining applications using a variety of graph datasets. Our evaluation demonstrates that our techniques deliver high performance for mining with containment constraints compared to existing state-of-the-art, and it further scales to larger graphs that existing graph mining systems as well as a custom solution failed to process (Section 8).

2 Background

2.1 Graph Terminology

A *graph* is a tuple $G = \langle V, E, L\rangle$, consisting of a vertex set V, an edge set $E \subset V \times V$, and a vertex labeling function $L : V \rightarrow \mathcal{L}$ where $\mathcal{L}$ is an arbitrary set of possible labels. A *subgraph* of G is a tuple $\langle V', E', L\rangle$ where $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. If S is a subgraph of G, we say G *contains* S. We consider undirected graphs for ease of exposition, but our techniques also apply to directed graphs. A *pattern* P is an arbitrary graph. Given a *data graph* G and a pattern P, if a subgraph S of G can be mapped one-to-one to P such that every edge in P is also present in S, and S and P have the same labels, then we say *S matches P* and the subgraph S is a *match for P*. We refer to vertices in the input data graph as *data vertices* and those in pattern graphs as *pattern vertices*.

2.2 Mining with Containment Constraints

Containment Constraints specify which matches are permissible based on other matches. A *containment constraint* is represented as a pair of patterns $\langle P^M, P^+\rangle$ that constrains matches for P^M. The constraint is specified over two cases depending on whether P^M is larger or smaller than P^+:
(a) If P^+ contains P^M, then constraint $\langle P^M, P^+\rangle$ specifies that a match m_1 for P^M is permitted iff there is no match m_2 for P^+ such that m_1 is a subgraph of m_2.
(b) If P^+ is contained within P^M, then constraint $\langle P^M, P^+\rangle$ specifies that a match m_1 for P^M is permitted iff there is no match m_2 for P^+ such that m_2 is a subgraph of m_1.
Given a data graph G, a pattern P^M and containment constraint $\langle P^M, P^+\rangle$, a subgraph s of G that matches P^M is considered valid iff it also satisfies the containment constraint $\langle P^M, P^+\rangle$.

Several graph mining applications have containment constraints, usually in the form of maximality or minimality of matches. We discuss the representative applications below that cover the different types of containment constraints.

The Maximal Quasi-Cliques (MQC) application [31] mines γ-quasi-cliques, *i.e.*, dense subgraphs of size k where each vertex has degree at least $\gamma(k-1)$. The maximality constraint mandates that the γ-quasi-cliques are not contained in any other γ-quasi-clique. There can be multiple quasi-cliques of a given size; hence, MQC has a collection of containment constraints $\langle P_1^M, P_1^+\rangle, \langle P_2^M, P_2^+\rangle, \ldots$ where each P_i^M is a quasi-clique of size k and P_i^+ is a quasi-clique of size $k' \geq k$. In Figure 1, `a-c-d-e` is a quasi-clique of size 4 (matches P^M), but it is invalid because `a-b-c-d-e-i` is a quasi-clique of size 6 (matches P^+) and the former is a subgraph of the latter. The Maximal Cliques application is a special case of MQC where both P^M and P^+ are cliques (fully connected patterns).

The Keyword Search (KWS) application [4] mines connected subgraphs up to a certain size k whose vertices cover a fixed set of labels W called keywords. Here, the matching subgraphs must be minimal: every vertex must either have a

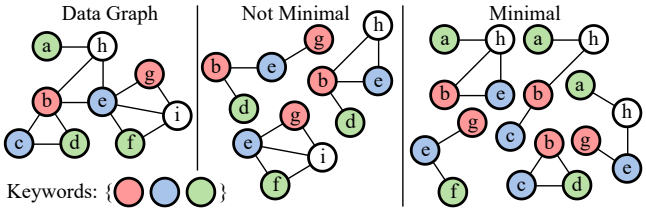


Figure 3. Example for Keyword Search on labeled data graph with matches containing three keywords (colored in red, blue and green). The matches on the right are minimal, while matches on the left are not minimal since they contain connected subgraphs that cover all three keywords.

label from W, or if the vertex is removed from the subgraph, it is no longer connected. Hence, each minimality constraint has P^M as a size-k pattern covering all labels in W, and P^+ being its subgraph that also covers all labels in W. Figure 3 shows an example data graph and all the minimal matches covering labels red, green and blue. Observe that although match `a-b-e-h` contains vertex h that does not cover any of the three labels, it is still minimal because the subgraph matching only `a-b-e` is disconnected.

Finally, Nested Subgraph Queries (NSQ) are queries where matches are constrained by being present or absent inside other specific subgraphs. For instance, an NSQ finding triangles that are not contained inside a size-5 house graph [1] has a single containment constraint $\langle P^M, P^+ \rangle$ where P^M is the triangle pattern and P^+ is the house graph pattern. An Anti-Vertex query [26] can also be modeled as a query with a single containment constraint $\langle P^M, P^+ \rangle$ where P^M is the pattern without the anti-vertex and P^+ is the pattern with an additional regular vertex in place of the anti-vertex.

2.3 Exploration Tasks in Graph Mining Systems

Graph mining systems efficiently explore subgraphs in a data graph G that match a pattern P. In order to discuss how to efficiently support containment constraints in graph mining systems, we first provide necessary background about their pattern matching process by modeling their exploration strategies as parallel exploration tasks.

Execution in graph mining systems can be logically separated into two phases: the *pattern matching phase* and the *match processing phase*. During the pattern matching phase, subgraphs of the data graph G that match the given pattern P are explored. Each subgraph is then processed using builtin graph mining algorithms (*e.g.*, counting) or user-defined functions (*e.g.*, filter, map, and reduce) in the match processing phase. Graph mining systems develop *exploration plans* that guide the pattern matching phase. The exploration plans mainly consist of a *matching order* or schedule (the order in which pattern vertices are matched with data vertices) and

[1] A house graph is a triangle (roof) combined with a 4-cycle (body).

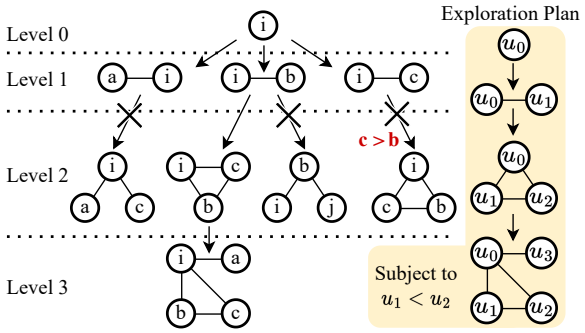


Figure 4. Exploration plan & search tree for tailed-triangle (*i.e.*, triangle with a dangling edge, shown at level 3). RL-Paths reaching level 3 match, whereas those terminating at lower levels do not.

symmetry-breaking restrictions (constraints on vertex ids of G to skip duplicate subgraphs).

Exploration Tasks (*ETasks*). Subgraph exploration in the pattern matching phase is decomposed into static, independent *exploration tasks* (or ETasks) that traverse G in parallel to generate subgraphs, one subgraph per thread at a time. ETasks are identified by the tuple $\langle P, S, C \rangle$, consisting of the pattern to match P, currently matched subgraph S of the data graph, and a local cache C with an entry for each vertex in P. Each ETask proceeds in depth-first fashion to match P, with S initialized to a single vertex in G (called its *root*), and C empty. S is then extended step-by-step with new vertices from G following the matching order, such that S always matches a subgraph of P. We mark the vertices of P as $u_0, \ldots, u_{|P|-1}$, where u_i is the i-th vertex in the matching order.

Initially, S is a match for only u_0. For each subsequent vertex u_i, the ETask: (a) computes a set of vertices V using set operations on the adjacency lists of vertices in S as well as cached values in C; (b) sets V as the cache entry for u_i; and (c) extends S using a vertex $v \in V$, if v has the correct label and satisfies the symmetry breaking restrictions on P; before (d) descending in the depth-first traversal to extend S to match u_{i+1}. Once S matches the entire P, it proceeds to the match processing phase. If there are no unused vertices from V to extend with, or if u_i was the final vertex in P, then the ETask backtracks to the previous pattern vertex u_{i-1}. The ETask completes when it must backtrack from u_1, *i.e.*, when it must match a new vertex to u_0.

When several patterns have identical structure (*i.e.*, same edge and vertex set) but different labels, the labels are merged so that they are all explored by a single ETask. In this case, the ETask ignores vertex labels at intermediate steps of the exploration, and for each found match it computes the final pattern using an isomorphism check [17, 35]. This enables greater reuse of cache C and reduces per-task overheads recurring across many patterns with the same structure.

Thus, the depth-first exploration beginning at the initial task state and following the matching order induces a *search tree* containing the different subgraphs of G that

arise. The search tree is organized into *levels*, with level 0 being the root consisting of a single vertex v and level $k-1$ being the subgraphs with k vertices that are explored when starting from v. Every point in the exploration where the ETask must backtrack corresponds to a root-to-leaf path in this tree. We call each such path an RL-Path of the ETask. An RL-Path *matches* if the leaf subgraph corresponds to a match for P. Figure 4 shows the exploration plan and search tree for a tailed-triangle pattern (a triangle with a dangling edge). The RL-Paths ending at level 3 match, *i.e.*, they result in a tailed-triangle. Other RL-Paths end at lower levels because they could not be extended by following the exploration plan, and hence they do not match.

The above model captures the execution of state-of-the-art pattern-based graph mining systems, including compilation-based systems [35, 39], pattern-aware systems [17, 25], and decomposition-based systems [5]. They all follow matching orders in depth-first fashion, computing and caching vertex sets from G using set operations at each step, and mapping them with pattern vertices until a match is found.

3 Overview of Contigra

Contigra is an execution model for graph mining with containment constraints that actively stops exploration of unnecessary subgraphs (*i.e.*, ones that are expected to violate constraints). By doing so, matches are checked against the required constraints naturally during exploration.

To enable Contigra, we model dependencies across concurrent tasks that, when enforced during exploration, ensure the constraints are correctly satisfied. The dependencies are categorized into three types (Section 4) based on how tasks traversing at different levels in search trees relate to each other according to various kinds of containment constraints.

Successor dependencies capture containment constraints like maximality by constraining ETasks for target subgraphs based on other subgraphs that are deeper in search trees. Contigra actively checks successor dependencies using *validation tasks* that spawn and continue from the ETasks to find subgraphs at deeper levels. To maximize reuse of the explored subgraphs and the associated local caches, the validation tasks are *fused* together with the ETasks that spawned them (Section 5.2). Since exploration plans for different subgraphs can be incompatible with each other, Contigra achieves task fusion by carefully aligning the exploration plans and bridging gaps in search trees, while simultaneously ensuring efficient RL-Paths are prioritized during validation. Furthermore, Contigra reuses the results computed by validation tasks for subsequent exploration by directly promoting validation tasks to ETasks and canceling the original ETasks that would have otherwise started from scratch.

Lateral dependencies capture constraints across tasks that traverse to the same depth in the search trees. Contigra automatically infers these dependencies on top of the available successor dependencies to cancel tasks that would result in redundant computation whenever certain constraints get violated (Section 6).

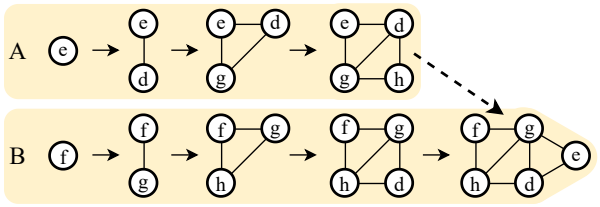


Figure 5. ETask A has a successor dependency with ETask B since A matches `e-d-h-g` which is contained inside match `e-d-h-g-f` which is matched by B.

Finally, *predecessor dependencies* capture containment constraints like minimality by constraining ETasks based on other subgraphs that are at a shallower depth in search trees. Due to the local nature of these dependencies, Contigra does not employ separate validation tasks, but instead statically analyzes the exploration state space to either skip ETasks from scheduled (when they are expected to result in dependency violation), or cancel them dynamically when dependency violations occur during exploration (Section 7).

4 Cross-Task Dependencies

We model the impact of containment constraints in terms of dynamic dependencies across ETasks. Containment constrained applications must satisfy these dependencies to ensure correctness or improve efficiency. Dependencies manifest among ETasks in three ways; in all three cases, each matching RL-Path in the dependent task depends on the result of a different RL-Path in order to determine how to process its subgraph. Hence, the notion of cross-task dependencies naturally extends to dependencies between matching RL-Paths in different tasks.

Successor Dependency. A containment constraint $\langle P^M, P^+ \rangle$, where P^+ is larger than P^M, constrains subgraphs matching P^M depending on the subgraphs that are explored deeper in search trees. When subgraphs explored by an ETask A depend on subgraphs explored by another ETask B which traverses *deeper in a search tree*, we say A has a successor dependency on B.

Consider the RL-Paths for ETask A and ETask B in Figure 5 for maximal quasi-cliques. A explores a size-4 quasi-clique, while B explores a size-5 quasi-clique. While A and B explore vertices in different order, by the time B reaches the final step of the RL-Path it has matched all the vertices in the quasi-clique Q found by A. Hence, Q is not maximal if B finds a quasi-clique. Since A depends on B, which explores deeper than A, we say A has a successor dependency on B.

Successor dependencies are validated by efficiently conducting explorations deeper in the search tree; details will be presented in Section 5.

Predecessor Dependency. A containment constraint $\langle P^M, P^+ \rangle$, where P^+ is smaller than P^M, constrains subgraphs matching P^M depending on subgraphs that are

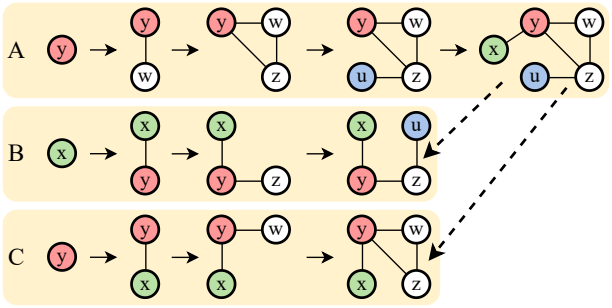


Figure 6. ETask A has predecessor dependencies with ETasks B and C since A matches `x-y-w-z-u` which contains `x-y-z-u` and `x-y-w-z` which matched by B and C respectively.

explored at a shallower depth in search trees. When an ETask A depends on an ETask B which traverses to a *shallower* depth in the search tree, we say A has a predecessor dependency on B.

For example, Figure 6 shows RL-Paths from 3 different tasks performing minimal keyword search. The ETask for RL-Path A explores a size-5 subgraph s containing all keywords, and depends on the ETasks exploring RL-Paths B and C that explore size-4 subgraphs. B and C both explore subgraphs that are contained in s, so if either of those matches contain the correct keywords then s is not minimal.

Note that in the above example the smaller subgraphs containing all the keywords are never explored in the RL-Path that matches s. Despite seeming like a local property, constraints like minimality induce predecessor dependencies across different tasks. In Section 7, we present how to efficiently validate such predecessor dependencies.

Lateral Dependency. When a task A depends on a task B which traverses to *the same depth in the search tree*, we say that A has a lateral dependency on B. Lateral dependencies may not be explicitly specified by containment constrained applications, since different matching RL-Paths at the same level explore different subgraphs and hence cannot contain each other. However, they can be automatically inferred and enforced by the system in order to improve efficiency by preemptively canceling certain tasks when a different task has already performed the required computation. These details will be explained in Section 6.

5 VTasks for Successor Dependencies

Successor dependencies $\langle P^M, P^+ \rangle$ arise when a subgraph matching P^M is constrained by matches for a larger pattern P^+, such as in applications searching for maximal subgraphs (*e.g.*, maximal quasi-cliques). Because the successor dependencies of an ETask involve subgraphs deeper in the search tree, they must be validated dynamically during exploration when ETasks explore matching subgraphs at the end of each RL-Path. In this section, we introduce a new kind of task called *validation tasks* (or VTasks) that are launched as subtasks of ETasks for validating successor dependencies.

Algorithm 1 Exploration Task

```
1:  function ETask(P : pattern, S : subgraph, C : cache)
2:    if |P| = |S| then                                  ▷ Leaf node of search tree
3:      status ← Match
4:      for P+ ∈ validationPatterns(P) do
5:        if VTask(P+, S, S, C) = VTask-Matched then
6:          status ← No-Match
7:          break                                        ▷ Cancel remaining VTasks
8:        else
9:          cancel ⟨P+, S, C⟩                          ▷ Cancel ETask
10:       end if
11:     end for
12:     if status = Match then
13:       ProcessMatch(S)                              ▷ Pass to Match Processing module
14:     end if
15:     for P+ ∈ nextExplorationPatterns(P, S) do
16:       if ⟨P+, S, C⟩ was not canceled then
17:         ETask(P+, S, C)                            ▷ Promote to new ETask
18:       end if
19:     end for
20:   else                                               ▷ Intermediate exploration step
21:     V ← computeCandidates(P, S, C)
22:     for v ∈ V do
23:       ETask(P, S ∪ {v}, C)
24:     end for
25:   end if
26: end function
```

During validation, VTasks are often required to explore portions of the search tree that are incompatible with their parent ETasks. Naïvely performing such explorations could result in redundant computations. We remedy this with *Task Fusion* that enables dynamic cache-sharing between ETasks and VTasks despite incompatible matching orders. On the other hand, a VTask may target the same pattern as another ETask. To avoid redundant exploration in such cases, we develop *Task Promotion* that directly converts the state of a VTask into that of an ETask to the same pattern.

5.1 VTask: Validation Task

While an ETask begins from a data vertex and traverses the search tree to generate all subgraphs matching a target pattern, a VTask is a special task represented as $\langle P, S^M, S, C \rangle$ which searches for a subgraph that both: (a) contains the subgraph S^M, and (b) matches the pattern P. Similar to ETask, VTasks also maintain a state consisting of subgraph S and cache C; however instead of exploring every RL-Path, VTasks terminate as soon as a single RL-Path containing S^M matches. An ETask $\langle P, S, C \rangle$ launches VTasks every time an RL-Path matches in order to check the successor dependencies for S. These VTasks take the form $\langle P^+, S, S, C \rangle$, where S^M is initialized to S and P^+ is a pattern larger than P.

For example in maximal quasi-cliques from Figure 1, an exploration task $\langle P, S, C \rangle$ with P being a 4-clique and S being `a-e-d-i` will spawn VTasks with P^+ being size-5 and size-6 quasi-clique patterns that contain a 4-clique. If a VTask matches, then S is contained in the larger pattern P^+, and hence S does not satisfy the containment constraint. This

Algorithm 2 Validation Task

```
27: function VTask(P : pattern, S^M : subgraph, S : subgraph, C : cache)
28:   P_S ← pattern(S)
29:   for ρ ∈ validPermutations(P_S) do
30:     S' ← permute S using ρ   ▷ Align S with exploration plan for P
31:     C' ← permute C to correspond to S'
32:     if |P| = |S'| − 1 then
33:       V ← computeCandidates(P, S', C')
34:       C ← permute C' back                    ▷ Reuse the VTask cache
35:       if V ≠ ∅ then        ▷ There are matches for P containing S^M
36:         return VTask-Matched
37:       end if
38:     else
39:       P^+ ← nextIntermediatePattern(P_S)   ▷ From Section 5.2.2
40:       V ← computeCandidates(P^+, S', C')
41:       for v ∈ V do
42:         if VTask(P, S^M, S' ∪ {v}, C') = VTask-Matched then
43:           C ← permute C' back                ▷ Reuse the VTask cache
44:           return VTask-Matched
45:         end if
46:       end for
47:     end if
48:   end for
49:   return No-VTask-Match
50: end function
```

is the case with a-e-d-i in our example as a VTask finds a-e-d-i-b, hence deeming a-e-d-i to be not maximal. The containment constraint is satisfied if none of the VTask RL-Paths matches. Algorithm 1 and Algorithm 2 summarize the core operations performed in ETasks and VTasks.

5.2 Task Fusion

Implementing VTasks directly using ETasks is not straightforward because it is unclear how to follow an exploration plan to generate a subgraph containing a specific subgraph S^M. Specifically, an ETask targeting pattern P can exhibit a successor dependency for another ETask targeting larger pattern P^+ which is rooted at a different vertex. Consider the example in Figure 7, where an ETask matches pattern P^M (a diamond pattern) with subgraph $S = S^M$ being e-d-g-h and resulting cache C. However, due to the matching order used by the ETask, no RL-Path of the same task will find the larger quasi-clique containing e-d-g-h. While it is possible to have multiple different exploration plans for each P^+, such an approach would be infeasible since VTasks would perform many redundant computations in order to re-explore S^M and then match P^+ from a different starting point. Moreover, sharing C between tasks statically (*i.e.*, prefix-sharing [35] or shared connected subpattern [17]) only works when tasks have compatible exploration plans, *i.e.*, results cannot be shared between dynamically spawned tasks.

For this, we develop *Task Fusion*, where VTasks are *fused* with the ETask that spawned them, copying their state to guarantee that only subgraphs containing S^M are found. Hence, the available S is reused along with the ETask's cache C to compute the remaining vertex sets that complete an RL-Path. However, tasks cannot be easily fused by simply

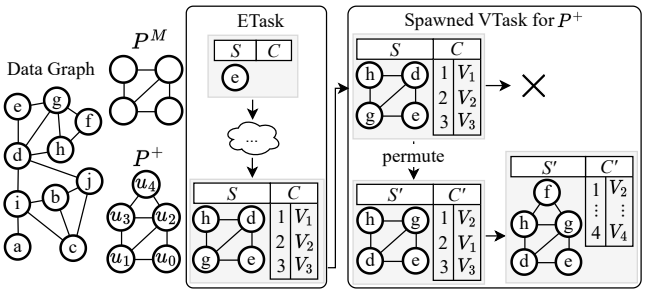


Figure 7. As ETask matches e-d-g-h for P^M, VTask for P^+ is spawned. Task alignment permutes e-d-g-h so that it can match e-d-g-h-f for P^+.

setting the VTask state to $\langle P^+, S, S, C\rangle$ (*i.e.*, reusing S and C for the larger pattern P^+) due to two main obstacles. First, the exploration plan for continuing to match P^+ using S would lead to incorrect execution of VTasks due to incompatibility with the exploration plan for P^M (*i.e.*, matching order and symmetry-breaking restrictions already applied on S). And second, the target pattern for an ETask and the target pattern for a VTask can differ by more than one level, leaving a non-trivial matching strategy for VTask to follow. We describe how these issues are addressed next.

5.2.1 Aligning Explorations in Fused Tasks. In our example in Figure 7, analyzing P^+ (a diamond-house pattern that matches e-d-g-h-f) results in a symmetry-breaking restriction $u_1 > u_3$, *i.e.*, the data vertex mapped to u_1 should have a larger id than the one mapped to u_3. Hence, e-d-g-h is an invalid intermediate state for P^+, since u_1 is mapped to g which is smaller than h mapped to u_3. While symmetry-breaking restrictions cannot be enforced when checking successor dependencies, forgoing them completely (*i.e.*, pattern-oblivious exploration) drastically reduces performance [25].

Moreover, the exploration plan for matching P^+ is incompatible with the plan followed by the RL-Path matching P^M. Following the exploration plan for P^+, candidate vertices for u_4 are drawn from the common neighbors of d and h (mapped to u_2 and u_3). But d and h have no common neighbors, so e-d-g-h-f is never found. This means the exploration plan for P^+ cannot be applied to check successor dependencies.

We address both issues using a combination of pattern-aware and pattern-oblivious approaches. The ETask uses the exploration plan with symmetry-breaking restrictions during exploration, and then it carefully adjusts the state to undo restrictions and reconcile exploration plans before executing the VTask. To safely account for all possible ways a match for P^+ could be encountered beginning from e-d-g-h, the exploration plan for P^+ is applied to those permutations of e-d-g-h that match the diamond pattern. One such permutation is shown in Figure 7, where the exploration plan draws candidates for u_4 from the shared neighborhood of g and h, yielding e-d-g-h-f.

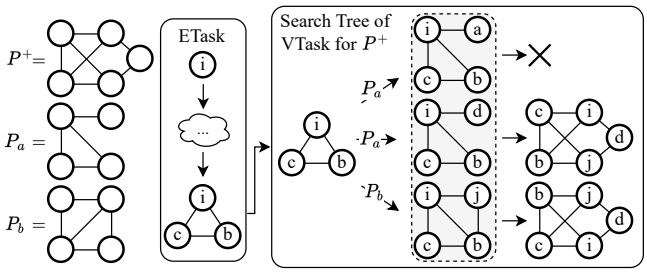


Figure 8. Bridging the gap between a triangle (size 3) and a size-5 match. There are two possible size-4 patterns P_a and P_b and the amount of work differs based on which one is explored first. If P_a is explored before P_b, the VTask takes more steps to find size-5 match since `c-b-i-a` does not explore further and the VTask needs to backtrack.

Lines 29–48 in Algorithm 2 show how state is adjusted in VTasks. For each valid permutation ρ, the subgraph S is permuted into a new subgraph S'. ρ is also applied to the cache C to obtain a new cache C' (Line 31) that is consistent with S'. The VTask proceeds to compute data vertices to match P^+ using S' and C' (Line 33). Then, C' is permuted back to C in order to allow other VTasks to correctly reuse the computations in the current VTask (Lines 34 and 43).

The necessary permutations are computed before exploration, and they are only applied when executing VTasks, hence ensuring ETasks explore each subgraph exactly once.

5.2.2 Bridging Gaps in VTask Search Trees. Successor dependencies between S^M and the target subgraph that is one level away from S^M in the RL-Path can be checked by operating on S' and C' (Line 33 in Algorithm 2). However, achieving this for subgraphs that are more than one level away from S^M requires more work. Figure 8 shows an instance of this case. The ETask explores P^M of size 3 (a triangle), but the VTask searches for P^+ with 5 vertices. Naïvely executing the VTask from P^M to P^+ would lead to similar issues arising from symmetry-breaking restrictions incorrectly pruning subgraphs when fusing VTasks with ETasks.

To correctly check successor dependencies in such cases, we bridge the gap between the ETask's target pattern and the VTask's target pattern. We chart a path through the search tree by choosing which pattern to explore at each intermediate step. As existing systems cannot provide exploration plans for continuing exploration from an arbitrary subgraph, we implement each intermediate step as a VTask invocation, so it can be fused with the previous step. This allows using the underlying system's exploration plans for the patterns at intermediate steps. Successor dependencies are therefore verified correctly as the plans for each separate VTask are aligned, while avoiding redundancy since caches are shared between steps. This idea is demonstrated in Algorithm 2, where on Line 42 the VTask recurses until the gap between the initial subgraph and the target pattern is closed, *i.e.*, a matching RL-Path is explored.

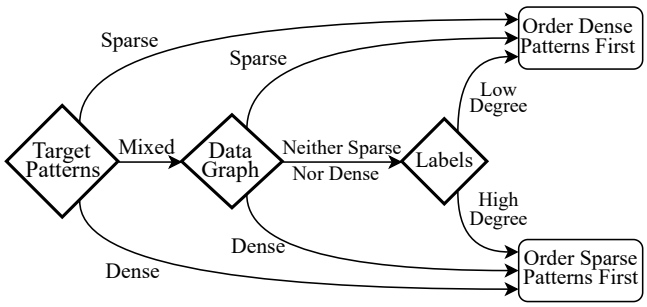


Figure 9. Decision tree for ordering RL-Paths.

***Efficient* RL-Paths.** The above procedure opens up multiple RL-Path options from S^M to the target pattern of the VTask. In our example, there are 3 RL-Paths starting at S^M that the VTask can take to match P^+. As each RL-Path results in different amounts of matching work, the performance of VTasks is sensitive to the order in which RL-Paths are taken to validate dependencies. For example, in Figure 8, the triangle `i-b-c` contains 3 vertices whereas P^+ contains 5 vertices. If the VTask first attempts to match the tailed triangle P_a, it would compute `i-b-c-a` only to find that it cannot be further extended to match P^+, and then backtrack to compute `i-b-c-d` which would eventually lead to `i-b-c-d-j` that matches P^+. On the other hand, the VTask going through P_b before P_a would compute `i-c-b-j` which would directly lead to `i-c-b-j-d`, hence matching P^+ in the first RL-Path.

To select an efficient ordering of RL-Paths, we develop heuristics that estimate the likelihood of matching the subgraphs at each intermediate step based on the relationship of the target patterns to each other and to the data graph. Our heuristics are based only on the density of the data graph and the possible patterns in order to compute the priority of different paths statically before exploration begins.

Figure 9 shows our heuristics as a decision tree. The majority of matches occur in dense regions of the data graph, where dense subgraphs are common and likely surrounded by other dense subgraphs [28]. If the target pattern is dense, the expected number of matching dense subgraphs at each intermediate step is higher, causing more work since each intermediate subgraph is further processed by the VTask. Hence when the target patterns are all dense (*e.g.*, high γ values in maximal quasi-cliques), the RL-Path exploring the sparsest patterns is chosen first in order to reduce intermediate matches. However, when target patterns are sparse, this trend reverses. Sparse subgraphs are present throughout the data graph and high-degree vertices in dense regions of the graph often reach into sparse regions, where the sparsest patterns have matches. Hence, when the target patterns are sparse, we prioritize RL-Paths targeting the densest patterns. Finally, if the target patterns include both sparse and dense patterns, we base our decision on the density of the data graph; for dense data graphs, sparse patterns are prioritized, and for sparse data graphs we prioritize dense patterns.

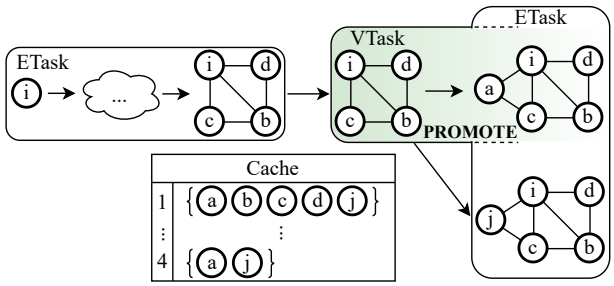


Figure 10. The VTask matches `i-b-c-d-a` for P^+, and is then promoted to ETask for subsequent exploration.

5.3 Promoting VTasks to ETasks

Applications like maximal quasi-cliques explore quasi-cliques of multiple sizes. In such cases, different ETasks explore patterns of different sizes, with certain VTasks checking dependencies against target patterns that are expected to be explored by other ETasks. Furthermore, constraints like maximality lead to exploring successors even when they get violated, *i.e.*, if a VTask matches a P^+ that contains P^M, then this P^+ becomes P^M in the subsequent ETask and a larger pattern that contains it would become P^+ in its VTask. This means the results computed by VTasks can be cached for future ETasks to avoid exploring the same subgraphs that were found by VTasks. We achieve this by promoting the VTask to an ETask, and canceling the original ETask which would have explored the subgraph from scratch. Thus, the resulting ETask directly uses the VTask cache C and the matched subgraph S for subsequent processing and validation.

Figure 10 shows an example following the patterns and data graph from Figure 7. The RL-Path belonging to the initial ETask finds the match `i-b-c-d` with cache C containing entries for the first 4 pattern vertices. Then a VTask is spawned which inherits C and finds a match `i-b-c-d-a` for P^+, caching candidates for the 5th pattern vertex at $C[4]$. Subsequently, this VTask is promoted to an ETask, and hence it immediately finds another match `i-b-c-d-j` without additional computation by reusing the candidates in $C[4]$.

VTasks are statically analyzed to identify the ones that target the same subgraphs as ETasks. Several VTasks originating from different ETasks can target the same pattern, and all VTasks that target the same P^+ are valid candidates to be promoted to an ETask exploring P^+. Since a single ETask is required to explore P^+, only one of the candidate VTasks is promoted to an ETask. While all candidate VTasks have a single matching RL-Path, when promoted to an ETask the remaining RL-Paths in the search tree also get explored. With each VTask having a different starting point, the size of the search tree and the number of RL-Paths that are traversed upon promotion differs across VTasks. Hence, the choice of which VTask is promoted to a given ETask has a direct impact in the amount of work performed by the promoted ETask. We determine which matching VTask gets promoted to a given ETask using the same heuristic from Section 5.2.2 that minimizes the number of RL-Paths the VTask will produce when it is treated as an ETask.

As shown in Algorithm 1, when an ETask finishes matching pattern P it runs VTasks on lines 4–11 to match P^+. If a VTask targeting P^+ does not match, then any ETask to P^+ from the same state cannot match, and hence the ETask is directly canceled on line 9. If all the VTasks match, since VTasks fuse their caches with that of the ETask which launched them (lines 34 and 43), the promoted ETask reuses the candidates already computed by the VTasks (line 21).

5.4 Generality of Task Fusion & Promotion

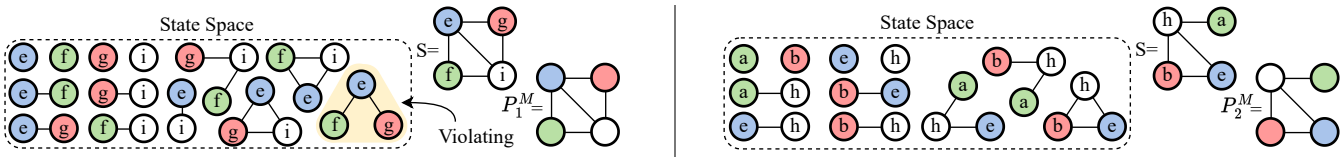
While task fusion and task promotion enable efficient execution while validating successor dependencies, they can also be applied to groups of ETasks in graph mining applications without successor dependencies (*i.e.*, beyond containment constrained applications). An ETask A targeting pattern P is fused with another ETask B targeting P' if P' is a subgraph of P, applying task alignment and bridging gaps as necessary. Then, if an RL-Path in B does not match P', A can be skipped to avoid exploring the same subgraphs. On the other hand, matching RL-Paths in B are promoted to executions of A to reuse caches and avoid redundant exploration from scratch.

6 Lateral Dependencies across VTasks

Applications often have multiple constraints $\langle P^M, P_1^+ \rangle$, $\langle P^M, P_2^+ \rangle$, ... on the same pattern P^M. For example, the maximality constraint in quasi-clique results into multiple constraints between a given quasi-clique of size k and different quasi-cliques of size $k+1$, each representing a different P_i^+. In such cases, when an ETask matches the subgraph S for P^M, multiple VTasks need to be scheduled, each targeting a different P_i^+. However, the subgraph S satisfies the application constraints only if it fulfills all of its dependencies, *i.e.*, S must satisfy all constraints on P^M that matches S. Hence, if one of the VTasks matches P^+, the other VTasks do not need to be executed as those dependency checks would not contribute to the final decision for S.

To avoid executing such unnecessary VTasks, we impose lateral dependencies between VTasks arising from the same ETask, which in turn enforces a serial execution of those VTasks. By doing so, VTasks can be easily canceled during execution; when any VTask in the serial execution matches, the remaining VTasks from that specific ETask are simply not executed (line 7 in Algorithm 2) and the ETask moves to matching the next RL-Path.

Since a single matching VTask cancels the remaining ones, it is important to order the VTasks such that the most likely to match are executed first. While the previous heuristics for ordering RL-Paths (Section 5.2.2) sought to reduce the chances of matching, here we want to identify VTasks that match as quickly as possible in order to end the validation process. Hence, we apply the same heuristics to estimate



(a) Contains violating state. **(b)** Does not contain violating state.

Figure 11. State spaces for RL-Paths that match P_1^M and P_2^M.

the relative likelihood of matching subgraphs but with the resulting decision inverted, *i.e.*, choose sparse patterns first if dense patterns first is prescribed, and vice versa.

Our lateral dependency-based execution enables VTask cancelation in a light-weight manner compared to any alternate solution that concurrently schedules all VTasks as it would require periodic synchronization to check whether the task is canceled. It is important to note that there is already sufficient parallelism in the form of ETasks, so serially executing VTasks does not affect scalability.

7 Predecessor Dependencies for ETasks

Predecessor dependencies $\langle P^M, P^+ \rangle$ where P^+ is smaller than P^M are typically seen in applications with minimality constraints (*e.g.*, keyword search). Unlike successor dependencies which involve previously unseen subgraphs, these constraints are local to an explored subgraph S and hence do not require special VTasks for validation. Instead, validation is performed on ETasks as they match RL-Paths.

A naïve approach to validate a predecessor dependency $\langle P^M, P^+ \rangle$ is to backtrack in the RL-Path for P^M to check if any of the intermediate subgraphs at previous steps matches P^+. However, RL-Paths do not necessarily explore all possible subgraphs of their target subgraph (as illustrated by Figure 6 in Section 4), and hence such an approach can miss containment constraints for one or more P^+. To address this, we classify all ETask states that can potentially lead to predecessor dependency violations. This enables Contigra to skip ETasks which are guaranteed to violate dependencies, perform *eager filtering* to prune ETasks which potentially violate them, and allow remaining ETasks which will never violate dependencies to execute unchanged.

State Space & Virtual State Space. Validating predecessor dependencies requires examining all possible subgraph states under all possible RL-Paths resulting from different exploration plans that target the same subgraph S. We call the set of these states the *state space* of an RL-Path. For example Figure 11a shows the state space of the RL-Path that explores labeled subgraph `e-g-f-i` that matches P_1^M for minimal keyword search. Note that the state space includes the smaller subgraph `f-e-g` containing all keywords.

For each matching RL-Path there are combinatorially many states explored at preceding levels, and therefore constructing and traversing all states in the space for every matching RL-Path is non-trivial. To alleviate this cost, we determine which states can violate the predecessor dependencies before exploration begins, using per-pattern *virtual state spaces*. For each target pattern P, we treat P as if it were the resulting state of a matching RL-Path, and construct its virtual state space, comprised of all connected subgraphs of P. This virtual state space corresponds one-to-one with the state space of all RL-Paths that match P, which allows us to statically determine before exploration begins whether any states violate the containment constraints.

***Skipping* ETasks.** An RL-Path R exploring state S that matches P^M violates the constraint $\langle P^M, P^+ \rangle$ for predecessor dependency when a state S' in its state space matches P^+. In this case, S is a *violating* state. For example in keyword search application which has minimality constraint, an RL-Path R has a violating state S if a smaller subgraph of S contains each of the necessary keywords.

We analyze the target patterns and group them in three cases based on whether the states in their virtual space violate the application constraints. For a given target pattern P^M, either: (a) there is some violating state, meaning every match for P^M violates its predecessor dependencies; or (b) none of the states are violating, meaning every match for P^M satisfies its predecessor dependencies. In addition, since an ETask can explore multiple patterns with the same structure but different labels (merged labels explained in Section 2.3), ETasks exploring multiple target patterns can fall in a third category (c) where they may explore some RL-Paths which violate predecessor dependencies, and others which do not.

In the first case, ETasks targeting P^M are unnecessary and can be safely skipped. In our example, the subgraphs in the virtual state space for P_1^M are identical in structure and labeling to those in the state space shown in Figure 11a. Hence the virtual state space also contains a violating subgraph, causing any RL-Path matching P_1^M to violate the minimality constraint. Thus, all ETasks targeting P_1^M are skipped. For the second case where every match for P^M satisfies its predecessor dependencies, the ETasks targeting P^M do not need to check containment constraints. Figure 11b shows the state space of an RL-Path matching P_2^M; the virtual state space of P_2^M has no violating states, and hence its ETasks do not perform any dependency checks.

Finally for the third case, ETasks perform *eager filtering* by checking constraints when exploring each RL-Path. ETasks maintain a set of violating states for each level in the search tree, corresponding to the violating states in the state space

of the merged patterns which are guaranteed to violate predecessor dependencies. Then at each level of exploration, if the current state matches a violating state for that level, the RL-Path is canceled and the ETask immediately backtracks.

This categorization drastically reduces the cost of satisfying predecessor dependencies by moving the computational burden from execution-time checks at each matching RL-Path to virtual state space analysis before execution. In KWS with 3 keywords and the exploration depth 4 (*i.e.*, patterns have at most 5 vertices), 273 of 287 patterns are guaranteed to violate predecessor dependencies, and the ETasks targeting these patterns are completely skipped (*i.e.*, a 95% reduction).

8 Evaluation

We evaluate the effectiveness of Contigra for containment constrained graph mining applications.

8.1 Implementation Details

We develop Contigra on top of **Peregrine+**, a modified version of the state-of-the-art graph mining system Peregrine [25] which supports simultaneous exploration of multiple patterns proposed in recent works which are not open-source [17, 34]. We implemented caches in Peregrine ETasks: set operation results are associated with each pattern vertex in a cache, and previous cache entries are reused to compute new operations. Patterns with identical pattern cores are explored simultaneously (*i.e.*, Shared Connected Subpattern [17]), and ETasks to such patterns share their caches (*i.e.*, intra-pattern reuse [17]). These modifications were implemented in ∼4300 lines of code.

ETasks and VTasks are implemented as individual recursive matching steps like in Algorithm 1 and Algorithm 2. Hence, task fusion and task promotion can be applied transparently to combinations of ETasks and VTasks by calling the appropriate function with the current subgraph and cache. For task fusion, the task exploring deeper ensures the subgraph and cache are consistent with its target pattern by permuting them. For task promotion, a valid subgraph has already been explored and hence no permutation is required.

To avoid runtime overheads, many aspects of task management are computed before exploration begins. ETasks maintain lists of VTasks that must be executed, and VTasks maintain plans for bridging gaps to target patterns. All tasks also track which tasks they can promote to, sorted using our heuristics for ordering RL-Paths. Finally, the permutations for aligning exploration plans are stored in lookup tables indexed by pattern combinations. Since these computations occur at pattern-level and not per match, it took only 0.1s–2s across all our experiments, compared to pattern exploration times which are often in 10's-1000's of seconds.

8.2 Applications, Datasets & Systems

Applications. We evaluate three containment constrained applications to cover the different dependency types: Maximal Quasi-Cliques (MQC) and Nested Subgraph Queries (NSQ) for successor dependencies, and Keyword Search (KWS) for predecessor dependencies. Lateral dependencies are automatically imposed across VTasks during execution. The maximal quasi-cliques finds γ-quasi-cliques up to size 6 that are maximal with γ between 0.8 and 0.6, which results in exploring 7–26 different quasi-clique patterns. We use two different nested subgraph queries: the first query searches all triangles not contained in two size-5 patterns shown in Figure 12a, whereas the second query searches all tailed triangles not contained in size-6 patterns shown in Figure 12b. In keyword search, we explore minimal subgraphs with up to five vertices that contain two different sets of 3 keyword labels: first set containing most frequent labels occurring in the data graph, and other set containing less frequent labels. Each label set results in matching upto 287 different patterns.

Data Graphs	Vertices	Edges	Labels
Amazon (AZ)	334.9K	925.9K	0
DBLP (DB)	317.1K	1.0M	0
Mico (MI)	96.6K	1.1M	28
Patents (PA)	2.7M	14.0M	36
Youtube (YT)	7.7M	50.7M	23
Products (PR)	2.4M	61.9M	46

Table 1. Real-world graph datasets used in evaluation.

Datasets. Table 1 shows the graph datasets used in our experiments, commonly used to evaluate graph mining solutions [5, 17, 25, 28, 35]. Amazon (AZ) [46] is a co-purchasing network where vertices are Amazon products and edges represent two products which are frequently purchased together. DBLP (DB) [46] is a co-authorship network where vertices are computer science researchers, and two researchers are adjacent if they have co-authored a paper. In the Patents (PA) [19] graph, vertices represent patents and edges represent citations between patents. Youtube (YT) [11] is a network of related videos, while Products (PR) [23] is a larger co-purchasing network on Amazon products.

Containment constrained applications are computationally expensive compared to traditional mining applications, and existing state-of-the-art cannot compute results for several of these graph datasets. Larger graphs (beyond the ones listed in Table 1) require higher time budget for experiments compared to traditional applications mainly due to the computational difficulty in checking containment constraints.

Systems. We compare the performance of our techniques with two state-of-the-art systems: **Peregrine+** and **TThinker** [31]. Peregrine+ is a general graph mining system that extends Peregrine [25] by batching the exploration plans together for efficiency (explained in Section 8.1). For nested subgraph queries and keyword search applications, we wrote the containment constraint checking code in the user callback of Peregrine+ (∼600 lines of code). TThinker on

Application	Baseline	Speedup
Maximal Quasi-Cliques (§8.4.1)	TThinker	**12–41700×**
Nested Subgraph Queries (§8.4.2)	Peregrine+	**5.6–379×**
Keyword Search (§8.5)	Peregrine+	**21–16000×**
Quasi-Cliques (§8.6)	Peregrine+	**2.4–7.2×**

Table 2. Summary of Contigra's performance.

the other hand develops a custom solution for mining maximal quasi-cliques using strategies to prune sparse regions of the graph that would never contain dense quasi-cliques.

All experiments we conducted on a 3.10GHz Intel(R) Xeon(R) Gold 6242R CPU with 64GB RAM and 40 physical cores, allowing 80 threads with hyperthreading.

8.3 Performance Summary

Table 2 summarizes the performance of Contigra compared to state-of-the-art baselines. Contigra enables efficient execution of containment constrained graph mining applications compared to the Peregrine+ graph mining system as well as the custom TThinker for maximal quasi-cliques. Moreover, Contigra scales to larger graphs that these baselines could not handle mainly because Contigra verifies dependencies actively during exploration whereas these baselines examine matches after they are explored, hence often running out of time or requiring massive memory/storage capacities to hold the explored matches for constraint checking. Finally, Contigra's task fusion and task promotion techniques also speed up graph mining execution in unconstrained applications like quasi-cliques.

8.4 VTask Performance

We study the performance of VTask and associated techniques for validating successor dependencies.

8.4.1 VTasks for Maximality. Table 3 compares the performance of maximal γ-quasi-cliques for Contigra and the state-of-the-art TThinker. As shown, Contigra is 12–41,000× faster than TThinker. Contigra delivers high performance due to VTasks and their associated techniques; we observed that up to 76.7% of VTasks and up to 72% of ETasks get canceled as VTasks check constraints, and task promotion increases the cache utilization to 75%. TThinker is only able to complete executions on the small Amazon and DBLP graphs, where its execution is dominated primarily by the phase that checks maximality of the explored subgraphs, failing on the larger data graphs due to its massive space requirements. On the MiCo graph TThinker runs out of storage after using 208GB of buffer space to store its exploration tasks, while only producing 280MB of potentially maximal quasi-cliques for future post-processing. On Patents, YouTube, and Products, TThinker exhausts the system's 64GB of memory, despite all three graphs taking less than 1GB space. Hence, the speedups reported for these large graphs are only a lower bound.

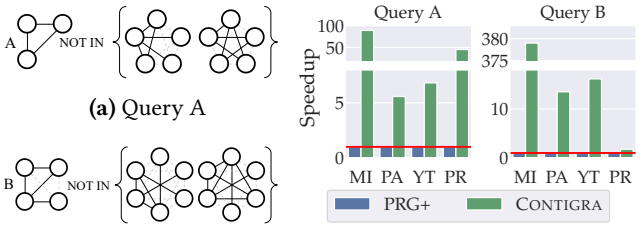
	$\gamma = 0.6$		
	Contigra	**TThinker**	**Speedup**
Amazon	0.92	9224.17	1.00e+4×
DBLP	13.76	TLE	6.28e+3×
Mico	4266.89	OOS	20.3×
Patents	199.28	OOM	434×
Youtube	1156.75	OOM	74.7×
	$\gamma = 0.7$		
	Contigra	**TThinker**	**Speedup**
Amazon	0.11	1263.08	1.20e+4×
DBLP	6.59	13435.26	2.04e+3×
Mico	887.67	OOS	97.3×
Patents	2.07	OOM	4.17e+4×
Youtube	18.59	OOM	4.65e+3×
Products	5867.63	OOM	14.7×
	$\gamma = 0.8$		
	Contigra	**TThinker**	**Speedup**
Amazon	0.12	785.24	6.53e+3×
DBLP	6.64	595.64	89.7×
Mico	1083.62	OOS	79.7×
Patents	2.92	OOM	2.96e+4×
Youtube	25.65	OOM	3.37e+3×
Products	7181.76	OOM	12×

Table 3. Execution times (in seconds) of Contigra and TThinker for maximal quasi-cliques. TLE indicates TThinker executions that did not complete in 24 hours. OOS indicates TThinker executions that ran out of storage and OOM indicates TThinker executions that ran out of memory.

8.4.2 VTasks for Nested Subgraph Queries. We evaluate two nested subgraph queries shown in Figure 12. The baseline in Peregrine+ extends each matched subgraph in the user-defined function to ensure it is not contained in any match for the larger patterns. Figure 12c compares the performance of Contigra with the Peregrine+ baseline. As shown, Contigra is 5.6–379× faster than the Peregrine+ baseline. This is mainly due to task fusion that enables cache reuse between VTasks, whereas the user-defined function in Peregrine+ has no access to the ETask caches. We again observe that Peregrine+ executions for several inputs do not complete in 24 hours, hence we plot conservative speedups.

8.4.3 Task Management Strategies.

Task Promotion. To evaluate the benefits of task promotion, we compare the hit rate of ETask caches with and without task promotion enabled in maximal quasi-cliques application. Figure 13 shows that cache hit rates can rise to 73% with task promotion from only 48% when it is disabled, as redundant operations are cached instead of recomputed.



(a) Query A **(b)** Query B **(c)** Performance.

Figure 12. Nested subgraph queries.

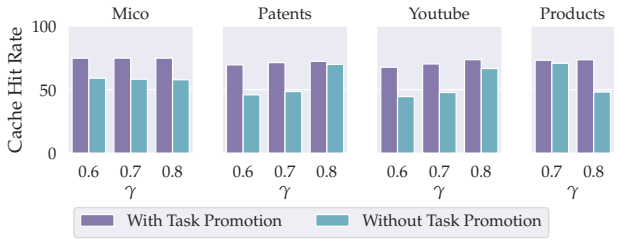


Figure 13. Cache hit rates with and without task promotion.

Lateral Dependencies. We quantify the effectiveness of VTask cancellation via lateral dependencies on the speedups over the baseline. Figure 14 shows executions of maximal quasi-cliques measuring the percentage of VTasks that were canceled. As shown, up to 77% of VTasks get canceled, motivating the importance of lateral dependencies.

RL-Path *Ordering*. Choosing which RL-Paths to explore first can affect performance when scheduling VTasks and bridging gaps between ETasks and VTasks. We study this performance impact and evaluate the effectiveness of our heuristics for prioritizing RL-Paths. Figure 16 shows executions of maximal quasi-cliques on various data graphs with different orderings of RL-Paths. As we can see, the performance difference between the fastest and the slowest executions is up to 2×. Our heuristics select the fastest executions in most of the cases. For Youtube with $\gamma = 0.7$ our choice is within 1.8 seconds of the fastest execution and for Patents with $\gamma = 0.7$ and $\gamma = 0.8$ our choice is within 0.1 seconds of the fastest execution.

8.5 Predecessor Dependencies

We evaluate keyword search with minimality constraint that results into predecessor dependencies. It searches for up to size-5 subgraphs containing 3 keyword labels that are most frequent in the data graph and other 3 keyword labels that are less frequent. Task promotion is also enabled in keyword search; since subgraphs of multiple sizes are explored, when an RL-Path to level k matches, its ETask gets promoted to patterns in level $k + 1$.

Figure 15 compares the results for CONTIGRA and Peregrine+ baseline. CONTIGRA performs 21–16138× faster than the baseline. This is due to the reduction in ETasks that are executed thanks to the combination of virtual state space analysis and eager filtering, as well as task promotion strategy. We observe that in comparison to Peregrine+ we explore

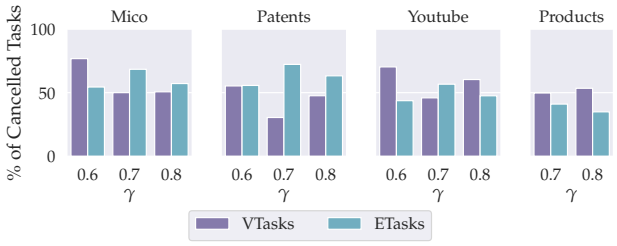


Figure 14. Task cancellations due to lateral dependencies.

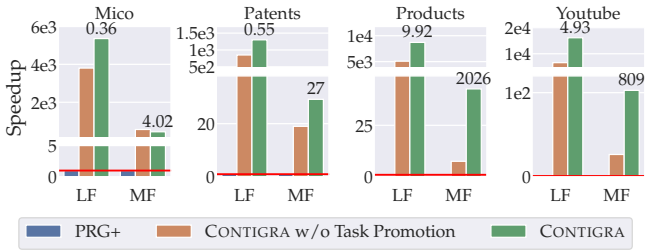


Figure 15. Performance of CONTIGRA and Peregrine+ for keyword search with most frequent (MF) and less frequent labels (LF). Numbers on top of bars indicate CONTIGRA execution times (sec).

only 0.6–2.5% of all possible ETasks. To break down which techniques lead to such aggressive reduction, we disabled task promotion in CONTIGRA and compared the performance when task promotion is enabled. We observed that with task promotion CONTIGRA explores only 19–47% of the ETasks in all cases except on YouTube graph, where up to 80% of the ETasks are explored mainly because many subgraphs end up having valid labels. Finally, eager filtering and task cancellation lead to far fewer RL-Paths being explored; and hence fewer dependency checks were performed; Figure 17 shows task elimination explores 40-85% fewer matches while eager filtering explores ~0.01% matches.

RL-Path *Ordering*. RL-Path ordering can significantly impact performance when promoting ETasks. Figure 18 shows the execution time using two opposing strategies for prioritizing RL-Paths. The dense strategy prioritizes RL-Paths targeting dense patterns first, while the sparse strategy prioritizes those targeting sparse patterns first. Our heuristics lead to the faster choice, giving up to 4.4× speedup. For Mico and Patents the performance difference is only ~0.6 seconds.

8.6 Generality of Task Fusion & Promotion

We further evaluate the generality of task fusion and task promotion for applications without successor dependencies. We run quasi-cliques without maximality constraints and with task fusion and task promotion between ETasks enabled, comparing the performance to Peregrine+ baseline without these techniques. Figure 19 shows the speedups of CONTIGRA over the baseline when matching γ-quasi-cliques without checking maximality. Task fusion and task promotion for ETasks lead to 2.4–7.2× faster execution.

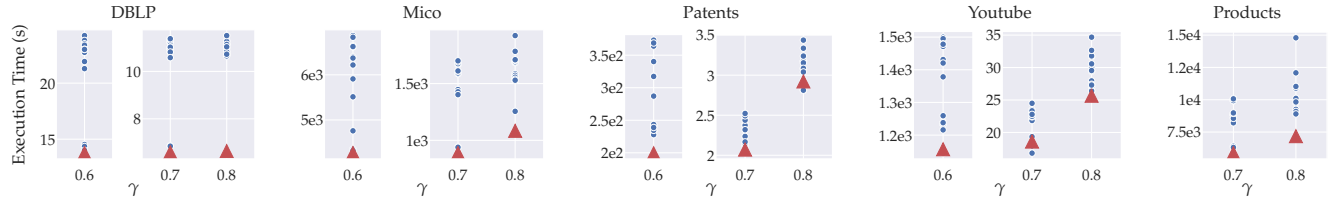


Figure 16. Executions with different RL-Path orderings. The ordering picked by our heuristic is marked with a red triangle.

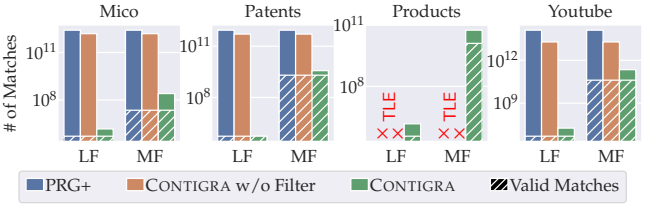


Figure 17. Number of matches checked for constraints in keyword search. TLE indicates executions did not complete.

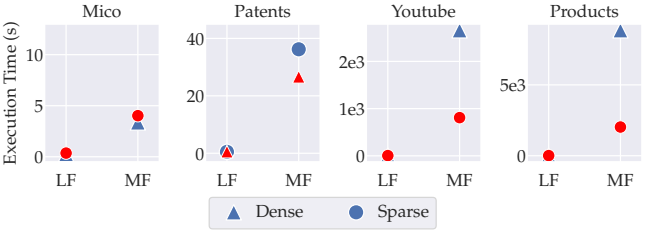


Figure 18. Executions with different RL-Path orderings in keyword search. The ordering picked by our heuristic is marked in red.

9 Related Work

To the best of our knowledge, this paper provides the first treatment for directly mining subgraphs that satisfy containment constraints, avoiding the need to examine subgraphs after exploration.

Graph Mining & Pattern Matching. General-purpose graph mining systems [4, 5, 8, 14, 17, 25, 35, 42, 44, 50] combine efficient pattern matching strategies with a programmable match processing module to support various graph mining applications. Pattern-oblivious systems [4, 9, 14, 42, 44, 50] explore subgraphs through iterative extensions by edges or vertices without taking pattern structure into account. Pattern-aware systems [5, 8, 25] exploit structural properties of target patterns in the pattern matching module, enabling powerful optimizations [6, 27, 28]. Pattern-based graph mining also benefits from research in pattern matching systems [1, 7, 10, 24, 29, 34, 37, 39, 40, 48] that enable techniques from architecture research [7, 10, 40], dataflow systems [1, 37], and compilers [34, 39]. None of these solutions consider containment constraints across subgraphs, hence requiring users to implement the constraint checking logic in user-defined functions that are invoked after each match is explored. Pattern-oblivious systems are further limited by properties like anti-monotonicity which may not apply to containment constrained applications.

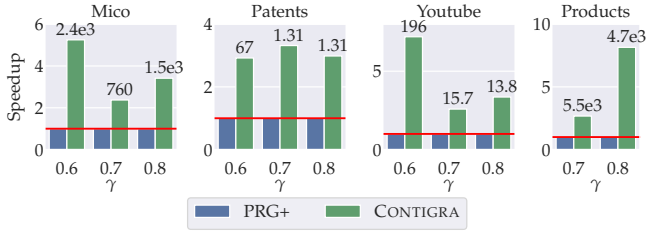


Figure 19. Performance of Contigra and Peregrine+ for quasi-cliques without maximality constraint. Numbers on top of bars indicate execution times (in sec) for Contigra.

Maximal Quasi-Cliques There are several specialized maximal quasi-clique solutions [12, 18, 31, 33, 36, 38, 49]. Most recent solutions are based on Quick [33], which is extended by the state-of-the-art system TThinker [31]. While the Quick algorithm reduces the search space by pruning out sparse regions of the graph, it relies on post-processing to eliminate matches that are not maximal, limiting its scalability for large graphs. On the other hand, Contigra efficiently executes maximal quasi-clique without post-processing, and can further support general constrained applications.

Graph Keyword Search There are many specialized keyword search algorithms [2, 15, 20, 30, 43, 47]. More recently, keyword search has been applied to RDF [15] and knowledge graphs [47]. These algorithms explore in pattern-oblivious manner and hence explore redundant subgraphs that cannot be minimal, whereas Contigra is able to skip such subgraphs using virtual state space analysis.

10 Conclusion

We developed Contigra for graph mining with *containment constraints*. We modeled containment constraints as cross-task *dependencies* and developed efficient *validation tasks* that merge explorations with validations, as well as techniques to actively avoid redundant explorations and constraint checks. Evaluation shows Contigra scales to massive workloads that could not be handled by existing solutions. This is the *first general treatment of containment constraints*, providing a framework for future systems to efficiently support containment constrained graph mining.

Acknowledgments

We would like to thank our shepherd Laurent Bindschaedler and the anonymous reviewers for their valuable and thorough feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, February 2018.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *18th International Conference on Data Engineering*, ICDE '02, pages 431–440, February 2002.

[3] Malay Bhattacharyya and Sanghamitra Bandyopadhyay. Mining the Largest Quasi-Clique in Human Protein Interactome. In *2009 International Conference on Adaptive and Intelligent Systems*, pages 194–199, 2009.

[4] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 458–473, 2021.

[5] Jingji Chen and Xuehai Qian. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '23, page 47–61, 2022.

[6] Jingji Chen and Xuehai Qian. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 413–426, 2023.

[7] Qihang Chen, Boyu Tian, and Mingyu Gao. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 43–55, 2022.

[8] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A Two-Level Framework for Efficient Graph Pattern Mining. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 378–391, 2021.

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment*, 13(10):1190–1205, April 2020.

[10] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA '21, pages 581–594, 2021.

[11] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and Social Network of YouTube Videos. In Hans van den Berg and Gunnar Karlsson, editors, *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238. IEEE, June 2008.

[12] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, Hongzhi Chen, and Guoren Wang. Fast Maximal Clique Enumeration on Uncertain Graphs: A Pivot-Based Approach. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2034–2047, 2022.

[13] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2246–2258, 2022.

[14] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1357–1374, 2019.

[15] Shady Elbassuoni and Roi Blanco. Keyword Search over RDF Graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 237–242, 2011.

[16] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, 2018.

[17] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. SumPA: Efficient Pattern-Centric Graph Mining with Pattern Abstraction. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 318–330, 2021.

[18] Guimu Guo, Da Yan, M. Tamer Özsu, Zhe Jiang, and Jalal Khalil. Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. *Proceedings of the VLDB Endowment*, 14(4):573–585, December 2020.

[19] Bronwyn Hall, Adam Jaffe, and Manuel Trajtenberg. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. *NBER Working Paper 8498*, 2001.

[20] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 305–316, 2007.

[21] John Hopcroft, Omar Khan, Brian Kulis, and Bart Selman. Tracking evolving communities in large linked networks. *Proceedings of the National Academy of Sciences*, 101(suppl_1):5249–5253, 2004.

[22] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. Mining Coherent Dense Subgraphs Across Massive Biological Networks for Functional Discovery. *Bioinformatics*, 21:i213–i221, June 2005.

[23] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[24] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 745–761, 2018.

[25] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages 1–16, 2020.

[26] Kasra Jamshidi, Mugilan Mariappan, and Keval Vora. Anti-Vertex for Neighborhood Constraints in Subgraph Queries. In *Proceedings of the ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '22, pages 1–9, 2022.

[27] Kasra Jamshidi and Keval Vora. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *SIGOPS Operating Systems Review*, 55(1):1–10, June 2021.

[28] Kasra Jamshidi, Harry Xu, and Keval Vora. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 162–181, 2023.

[29] Peng Jiang, Yihua Wei, Jiya Su, Rujia Wang, and Bo Wu. SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 185–197, 2023.

[30] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional Expansion for Keyword Search on Graph Databases. In *Proceedings of the 31st*

International Conference on Very Large Data Bases, VLDB '05, page 505–516, 2005.

[31] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. Parallel Mining of Large Maximal Quasi-Cliques. *The VLDB Journal*, 31(4):649–674, November 2021.

[32] Junqiu Li, Xingyuan Wang, and Yaozu Cui. Uncovering the overlapping community structure of complex networks by maximal cliques. *Physica A: Statistical Mechanics and its Applications*, 415:398–406, 2014.

[33] Guimei Liu and Limsoon Wong. Effective Pruning Techniques for Mining Quasi-Cliques. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 33–49, 2008.

[34] Daniel Mawhirter, Sam Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 289–303, 2021.

[35] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, 2019.

[36] Jian Pei, Daxin Jiang, and Aidong Zhang. On Mining Cross-Graph Quasi-Cliques. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, page 228–238, 2005.

[37] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 321–335, April 2021.

[38] Seyed-Vahid Sanei-Mehri, Apurba Das, Hooman Hashemi, and Srikanta Tirthapura. Mining Largest Maximal Quasi-Cliques. *ACM Transactions on Knowledge Discovery from Data*, 15(5), April 2021.

[39] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pages 1–14, 2020.

[40] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. NDMiner: Accelerating Graph Pattern Mining Using near Data Processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 146–159, 2022.

[41] Brian K. Tanner, Gary Warner, Henry Stern, and Scott Olechowski. Koobface: The evolution of the social botnet. In *2010 eCrime Researchers Summit*, pages 1–10, 2010.

[42] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, 2015.

[43] Haixun Wang and Charu Aggarwal. *A Survey of Algorithms for Keyword Search on Graph Data*, pages 249–273. Springer, February 2010.

[44] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 763–782, 2018.

[45] Chun Wei, Alan Sprague, Gary Warner, and Anthony Skjellum. Mining Spam Email to Identify Common Origins for Forensic Application. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, page 1433–1437, 2008.

[46] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[47] Yueji Yang, Divykant Agrawal, H. V. Jagadish, Anthony K. H. Tung, and Shuang Wu. An Efficient Parallel Keyword Search Engine on Knowledge Graphs. In *35th International Conference on Data Engineering*, ICDE '19, pages 338–349, 2019.

[48] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2049–2062, 2021.

[49] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Coherent Closed Quasi-Clique Discovery from Large Dense Graph Databases. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 797–802, 2006.

[50] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *36th International Conference on Data Engineering*, ICDE '20, pages 673–684, 2020.