Exploiting Cross-Task Dependencies in Graph Mining with Containment Constraints

by

Joanna Che

B.Sc., Simon Fraser University, 2019
B.Eng., Zhejiang University, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Joanna Che 2023
SIMON FRASER UNIVERSITY
Fall 2023

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Joanna Che**
Degree:	**Master of Science**
Thesis title:	**Exploiting Cross-Task Dependencies in Graph Mining with Containment Constraints**
Committee:	**Chair:** Saba Alimadadi Assistant Professor, Computing Science
	Keval Vora Supervisor Associate Professor, Computing Science
	William (Nick) Sumner Committee Member Associate Professor, Computing Science
	Anders Miltner Examiner Assistant Professor, Computing Science

Abstract

Graph mining workloads search for subgraphs of interest in large graphs. This often involves finding subgraphs with containment constraints such that the subgraph does not contain a smaller subgraph (minimality) or the subgraph is not part of a larger subgraph (maximality). Existing graph mining systems employ efficient task-parallel strategies to quickly explore subgraphs of interest, however they remain oblivious to containment constraints. Hence, graph mining systems require expensive constraint checking on every explored match as well as redundant explorations that limit their scalability.

In this work, we develop a novel exploration model for mining subgraphs with containment constraints. First, we identify the impact of constraints using different types of dependencies (successor, predecessor, and lateral) between the subgraphs of interest. Then, we develop four key task management strategies to exploit these dependencies: (a) *task fusion* to merge correlated tasks for increasing cache reuse; (b) *task promotion* to allow continuous explorations from available subgraphs and skip re-exploring subgraphs from scratch; (c) *task cancellations* to avoid unnecessary constraint checking and prioritizes faster constraint validations; and (d) *task skipping* to safely skip certain exploration and validation tasks. Finally, we extensively evaluate our model on real-world graphs and applications. Our task management strategies efficiently compute graph mining queries with containment constraints and our exploration model scales to very large graph mining workloads that could not be handled with prior approaches.

Keywords: Graph Mining; Pattern Matching; Subgraph Exploration; Maximal Quasi-Cliques; Graph Keyword Search; Nested Subgraph Queries

Acknowledgements

I am immensely grateful for the invaluable guidance and unwavering support extended to me by my advisor, Prof. Keval Vora, whose mentorship has played a pivotal role in shaping this thesis. Without his expertise and unwavering encouragement, this work would not have reached fruition.

My heartfelt appreciation extends to all my lab mates: Mobin Dariush, Pourya Vaziri, Lynus Vaz, and Rakesh Mahadasa, whose contributions have been invaluable throughout this research endeavor. I am especially indebted to Kasra Jamshidi and Mugilan Mariappan for their exceptional kindness, unwavering support, and invaluable assistance that have been crucial in shaping the trajectory of this work.

I want to express my deepest gratitude to my family and friends for their unwavering encouragement and continuous support, which have served as a constant source of strength and inspiration throughout this transformative journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graph mining is important in many domains, such as bioinformatics [4, 23], social network analysis [22, 34], cybersecurity [49, 45]. These domains have applications where they require mining subgraphs of interest in large graphs. However, graph mining is a challenging task because it involves subgraph matching which is NP-complete. Also, the amount of results can grow exponentially and holding or storing all the results may not be feasible. To simplify such graph mining tasks, various graph mining systems like Peregrine [26] and others [46, 5, 6, 7, 44, 39, 43, 18] have been developed which explore subgraphs of interest in large graphs.

These systems decompose the subgraph exploration into static, independent *exploration tasks* that traverse through the graph in parallel, finding one subgraph *match* per task at a time. While these systems support applications like counting motifs or finding frequently occurring subgraphs, several applications like Maximal Quasi-Cliques [35] and Keyword Search [16] are difficult to run on these systems as they stipulate additional constraints like *maximality* or *minimality*.

In this thesis, we define these additional constraints as *Containment Constraints* as they are in the form of a match being present or absent inside another match. To efficiently support graph mining applications with containment constraints on modern pattern-based graph mining systems, we develop Contigra, a general execution model for finding subgraphs with containment constraints.

1.1 Motivation

Containment constraints are common in many applications. For example in social network analysis, a common use case is community detection for recommending friends. Vertices would represent people and edges would represent connections between them. A densely connected subgraph would mean that users in that subgraph would likely know the other users in the group even if there is no edge connecting them, allowing for friend recommendations [36]. A containment constraint, such as maximality, can be used to improve results by only returning

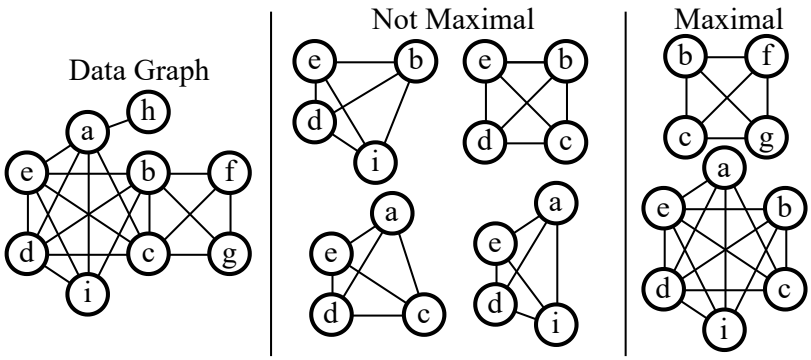


Figure 1.1: Example data graph and its quasi-cliques with $\gamma = 0.8$. There are 5 γ-quasi-cliques of size 4, but the ones shown on the left are not maximal due to the size 6 γ-quasi-clique `a-b-c-d-e-i` (shown on the right).

cliques or quasi-cliques (*i.e.*, dense subgraphs that may have missing edges) that are not contained inside larger cliques or quasi-cliques.

In addition to this, containment constrained graph mining applications can also be useful for spam emails and can be used to reveal information about these emails and help investigators with forensic analysis. For example, if a spammer owns multiple domain names and sends spam emails, we can represent vertices as domains and edges as the subjects from the emails. The analysis can find relationships between the emails through clustering methods as it is likely for a spammer to send emails with similar subjects from the domains they own [49]. As cliques or quasi-cliques are dense subgraphs, they can be used to find these clusters and identify the domains a spammer may own.

There are also other applications with containment constraints. For example, the Keyword Search application searches for the minimal connected subgraph that contains all the valid keywords. This can be useful in many situations such as recommending location-based points of interest to a user. For example, if a tourist is exploring a city, they may be interested in terms such as "hotel", "tourist attraction", and "restaurant". The data graph would have vertices representing locations with labels and edges as roads connecting them. In this situation, a good recommendation would return locations that contain these terms and are close to each other [51]. There are also queries that specify neighborhood constraints with Anti-Vertices [27] that produces subgraphs which are not contained in larger subgraphs using anti-vertices. These can be further generalized into Nested Subgraph Queries that use nested MATCH clauses in Cypher/GQL [17, 14].

Figure 1.1 shows an example data graph as well as all the quasi-cliques it contains. A quasi-clique is a dense graph structure similar to a clique (a fully connected graph), but there may be a few edges missing. Note that there can be multiple different ways to define a quasi-clique and they all differ depending on their definition of density. Regardless of whichever density definition is used, the following issues in finding maximal quasi-cliques remain the same.

In our example, despite there being many quasi-cliques, only two of the ones shown in Figure 1.1 are maximal. In the figure `a-c-d-e` is a valid quasi-clique however it is not maximal because it is contained inside `a-b-c-d-e-i`. As a result, the Maximal Quasi-Clique problem must return `a-b-c-d-e-i` in its result set and any quasi-cliques contained within it, such as `a-c-d-e`, must be excluded.

Finding such maximal quasi-cliques on pattern-based graph mining systems is challenging for several reasons. First, the maximality constraint is not specified on the structure of the subgraph alone, meaning a user cannot only specify the patterns the system needs to find. In the previous example, all subgraphs shown are valid quasi-cliques but only a subset of the results are valid for the Maximal Quasi-Clique application. This is visible in Figure 1.1 where `a-c-d-e` and `f-g-c-b` have the same structure, but only the latter is maximal. In order for the maximal constraint to be applied, the system needs to be able to check the relationship between matches. However, exploration tasks in graph mining systems follow static loop schedules (or matching orders), the task exploring `a-c-d-e` can remain independent of the task exploring `a-b-c-d-e-i`, making it difficult to enforce the maximality constraint by checking these two matches. The only way to ensure maximality is to examine every individual match returned by the exploration 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).

Secondly, checking whether a match satisfies maximality is not a simple task. Each quasi-clique match can potentially be a part of multiple larger quasi-cliques. For instance in Figure 1.1 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 is further complicated since maximality checks can span across multiple sizes. For example a size k quasi-clique might not be inside any size $k+1$ quasi-clique while it may still be part of a size $k+2$ quasi-clique. This is a result of quasi-cliques not adhering to the anti-monotonicity property, where if a property is true for a graph, then that property will be true for subgraphs contained within that graph. Or in other words, a subgraph of a quasi-clique may not be a valid quasiclique. As a result, the process of satisfying the maximality constraint is computationally intensive for each match as it becomes necessary for every match to go through multiple checks against matches from other tasks.

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 1.2 shows the performance for Peregrine [26] and GraphPi [43]. As we can see, the maximality checks often add over an order of magnitude performance penalty compared to executions without

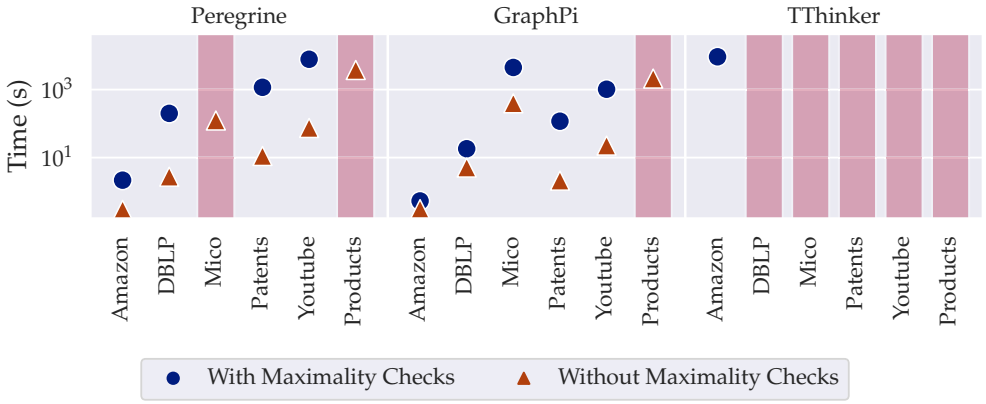


Figure 1.2: Performance of mining quasi-cliques with and without the maximality checks. Red bars indicate executions with maximality check that did not complete in 12 hours.

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.

To address this scalability bottleneck, solutions like TThinker [32] and others [19, 35, 42] develop custom algorithms for finding maximal quasi-cliques. These solutions reduce the number of matches to be checked by pruning the sparse regions of the graph that would never contain the dense quasi-cliques, hence reducing n in $\mathrm{O}(C(n,k))$ checks However, similar to the 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 1.2 and as we can see, it only successfully completes finding maximal quasi-cliques for one small graph and does not finish execution for the remaining five graphs.

The above scalability bottlenecks are also present when enforcing other containment constraints. Minimal Keyword Search, another example of containment constrained graph mining application, aims to find subgraphs with certain specific keywords such that they themselves do not contain smaller subgraphs with all those keywords. Again here, multiple minimality checks need to be performed for each match against other subgraph matches that get explored by other tasks, hence limiting scalability for large graphs. Similarly for Nested Subgraph Queries [17], multiple checks need to be done to ensure that the absence or presence of subgraphs within and around the matches are explored.

1.2 Our Solution

In this thesis, we develop efficient techniques to enable graph mining applications with containment constraints. Our goal is to enrich the pattern matching strategies in state-of-the-art pattern-based graph mining systems [39, 26, 43, 18] 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.

To enable Contigra we model the effect of containment constraints in terms of dynamic *dependencies* across concurrent exploration tasks (Chapter 3). We identify three key dependency types (successor dependencies, lateral dependencies, and predecessor dependencies) 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 (Chapter 4). While validation tasks are spawned dynamically from exploration tasks, we enable *task fusion* that fuses exploration and validation tasks together and allows us to maximize cache reuse through leveraging the associated caches from the explored subgraphs 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.

Contigra is also able 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 to avoid unnecessary constraint checking (Chapter 5). 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. Prior graph mining solutions have remained oblivious to dependencies between patterns, and hence are unable to take advantage of the benefits that exploiting these dependencies could provide.

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 we either safely *skip exploration*

tasks, *skip validation* of constraints, or perform *eager filtering* to actively check constraints (Chapter 6).

We demonstrate the effectiveness of techniques by incorporating Contigra in Peregrine, a state-of-the-art graph mining system and evaluate 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 state-of-the-art graph mining systems as well as a custom solution for maximal quasi-clique failed to process (Chapter 7).

Furthermore, the techniques developed in this thesis are widely applicable beyond applications with containment constraints. Our modeling of the effects of containment constraints using dependencies allows Contigra to be useful for any application that has relationships and constraints across different patterns and their matches. For example subgraph mining is often required in protein interaction graphs [55], where vertices represent proteins and edges represent interactions between proteins. The graphs that represent such data are often noisy due to errors in data collection. In this case, subgraphs that are similar to the given target patterns should be explored. Even though these subgraphs do not have any containment constraints, the similarity of subgraphs can be modeled as dependencies between them as well as the target pattern in order to avoid exploring subgraphs that may be less relevant due to presence of other subgraphs.

Chapter 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 $L : V \to \mathcal{L}$ 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 Graph Mining

2.3 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 $\langle P^M, P^+ \rangle$.

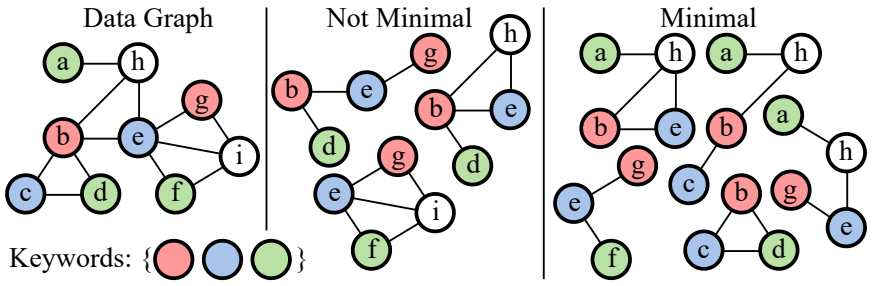


Figure 2.1: Example for Keyword Search on labeled data graph with matches containing three keywords (colored in red, blue and green). The matches on the right are minimal, while matches on the left are not minimal since they contain connected subgraphs that cover all three keywords.

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.

2.4 Maximal Quasi-Cliques

The Maximal Quasi-Cliques (MQC) application [32] 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 $\langle P_1^M, P_1^+\rangle$, $\langle P_2^M, P_2^+\rangle$, ... 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 1.1, `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).

2.5 Keyword Search

The Keyword Search (KWS) application [5] 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 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 2.1 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.

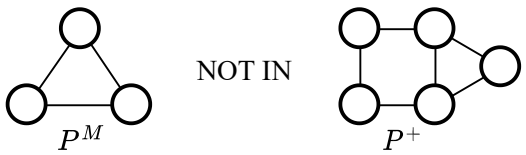


Figure 2.2: Example for NSQ for finding all triangles (P^M) not in a house graph (P^+).

2.6 Nested Subgraph Queries

Finally, certain Nested Subgraph Queries [17] (NSQ) can also be modeled as queries with containment constraints. For instance Figure 2.2 depicts a nested subgraph query for finding triangles that are not contained inside a size-5 house graph. This query 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 [27] 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.

2.7 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 that dictates the order in which pattern vertices are matched with data vertices. The exploration plans also account for *symmetry-breaking restrictions* which use constraints on vertex ids of G to skip duplicate subgraphs.

2.7.1 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

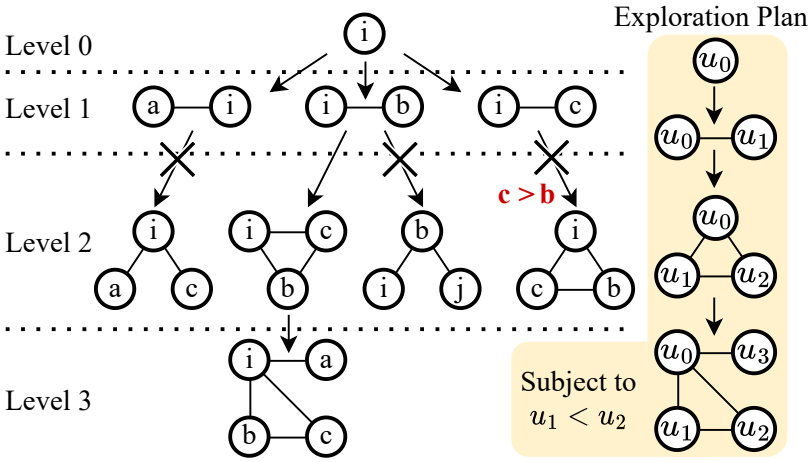


Figure 2.3: Exploration plan & search tree for tailed-triangle (*i.e.*, triangle with a dangling edge, shown at level 3). RL-Paths reaching level 3 match, whereas those terminating at lower levels do not.

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 [39, 18]. 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 2.3 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 [39, 43], pattern-aware systems [26, 18], and decomposition-based systems [6]. 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.

Chapter 3

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.

With containment constraints, dependencies manifest among ETasks in three ways: (a) an ETask can depend on another ETask which explores *deeper in the search tree* (*successor dependency*); (b) an ETask can depend on another ETask at a *shallower depth in the search tree* (*predecessor dependency*); and (c) an ETask can depend on another ETask at *the same depth in the search tree* (*lateral dependency*). 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.

3.1 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.

For example in maximal quasi-cliques, a matching RL-Path yielding a quasi-clique match Q has successor dependencies on all RL-Paths exploring larger quasi-cliques which contain Q. If such larger quasi-cliques exist, then Q is not maximal. Consider the RL-Paths for ETask A and ETask B in Figure 3.1. A explores a size-4 quasi-clique, while B explores a size-5 quasi-clique. Notice that 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, more details will be presented in Chapter 4.

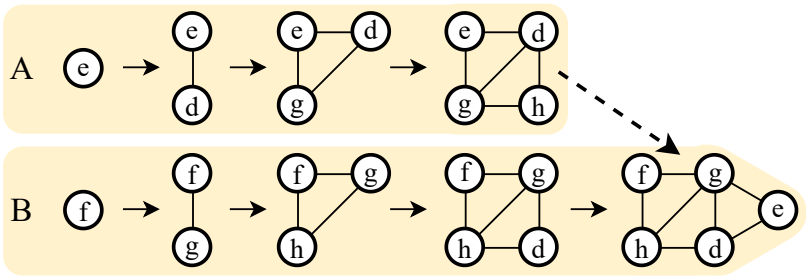


Figure 3.1: ETask A has a successor dependency with ETask B since A matches `e-d-h-g` which is contained inside match `e-d-h-g-f` which is matched by B.

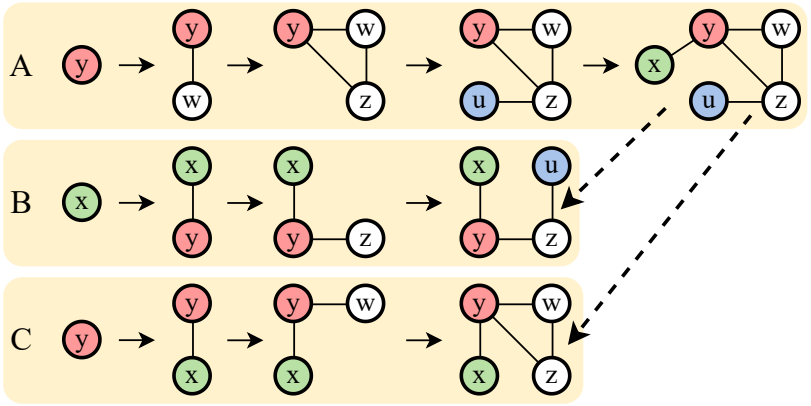


Figure 3.2: ETask A has predecessor dependencies with ETasks B and C since A matches `x-y-w-z-u` which contains `x-y-z-u` and `x-y-w-z` which matched by B and C respectively.

3.2 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 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 in minimal keyword search, a subgraph is considered valid based on the results of all tasks exploring smaller subgraphs with the same vertices. Figure 3.2 shows RL-Paths from 3 different tasks performing Keyword Search. The ETask for RL-Path A explores a size-5 subgraph s containing all keywords, and depends on the ETasks exploring size-4 RL-Paths B and C. B and C both match 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 Chapter 6, we present how to efficiently validate such predecessor dependencies.

3.3 Lateral Dependency

When an ETask A depends on an ETask B which traverses to *the same depth in the search tree*, we say that A has a lateral dependency on B. Lateral dependencies are not 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 Chapter 5.

Chapter 4

VTasks for Successor Dependencies

Successor dependencies must be checked as ETasks explore matching subgraphs at the end of each RL-Path. We introduce a new kind of task called *validation tasks* (or VTasks) that are responsible to validate successor dependencies. In this section, we will describe how VTasks are launched as subtasks of ETasks, and present *Task Fusion* and *Task Promotion* to reuse exploration states for efficiency.

4.1 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 1.1, 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 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.

4.2 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.

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
```

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 4.1, 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 [39] or shared connected subpattern [18]) 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 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,

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 4.2.2
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
```

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.

4.2.1 Aligning Explorations in Fused Tasks

In our example in Figure 4.1, 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 [26].

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

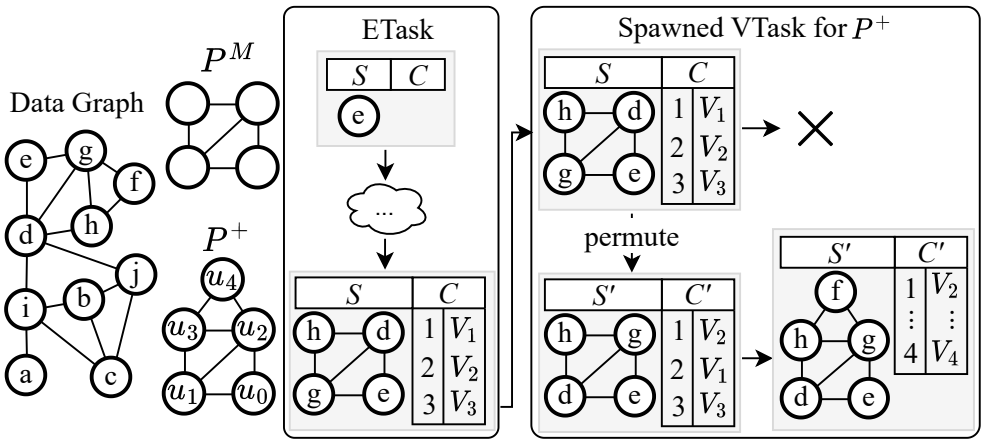


Figure 4.1: As ETask matches `e-d-g-h` for P^M, VTask for P^+ is spawned. Task alignment permutes `e-d-g-h` so that it can match `e-d-g-h-f` for P^+.

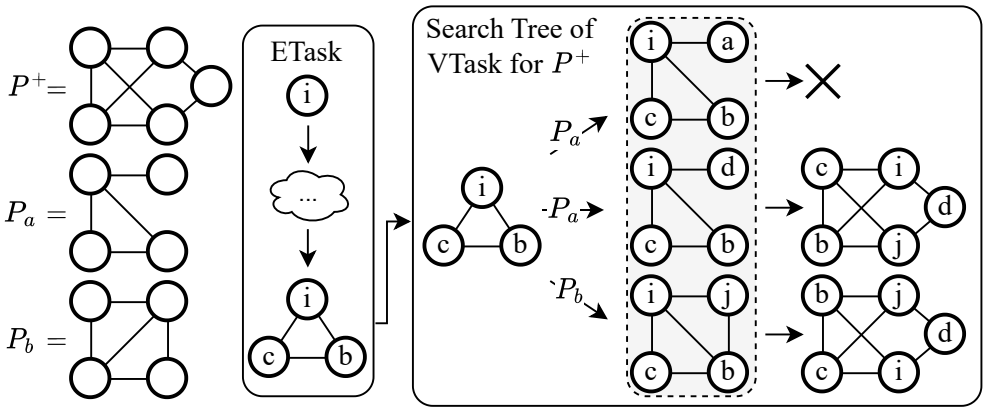


Figure 4.2: Bridging the gap between a triangle (size 3) and a size-5 match. There are two possible size-4 patterns P_a and P_b and the amount of work differs based on which one is explored first. If P_a is explored before P_b, the VTask takes more steps to find size-5 match since `c-b-i-a` does not explore further and the VTask needs to backtrack.

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 4.1, where the exploration plan draws candidates for u_4 from the shared neighborhood of `g` and `h`, yielding `e-d-g-h-f`.

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).

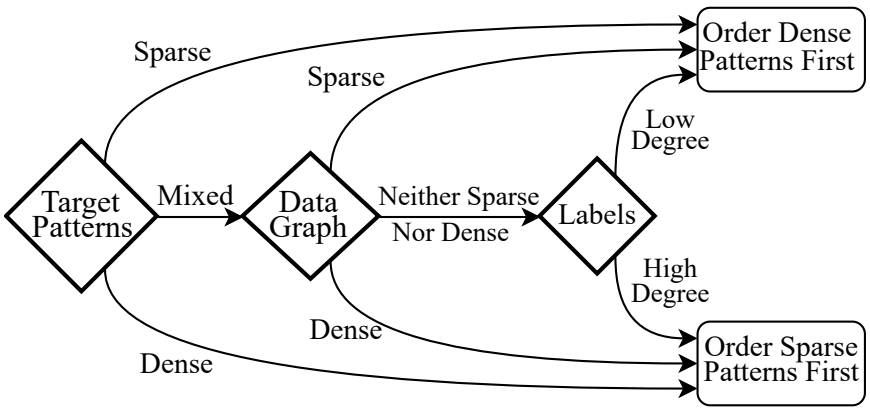


Figure 4.3: Decision tree for ordering RL-Paths.

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

4.2.2 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 4.2 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.

4.2.3 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 4.2, 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 4.3 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 [29]. 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.

4.3 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.

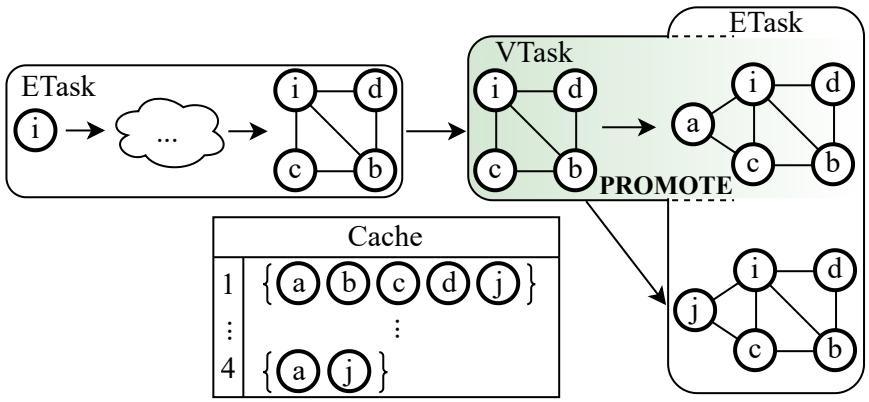


Figure 4.4: The VTask matches `i-b-c-d-a` for P^+, and is then promoted to ETask for subsequent exploration.

Figure 4.4 shows an example following the patterns and data graph from Figure 4.1. 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 4.2.2 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).

4.4 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.

Chapter 5

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. For example, in Figure 4.2 the ETask has a couple options for size 4 patterns, P_a and P_b. There is a lateral dependency between P_a and P_b, if the ETask creates VTasks to target P_a and P_b, if one of the VTasks matches, the other VTask does not need to be scheduled.

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 4.2.2) 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 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 cancellation 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.

5.1 Generality of Lateral Dependencies

In addition to lateral dependencies for VTasks, there are situations where lateral dependencies can be applied across ETasks.

For example subgraph mining is often required in protein interaction graphs [55], where vertices represent proteins and edges represent interactions between proteins. The graphs that represent such data are often noisy due to errors in data collection. In this case, subgraphs that are similar to the given target patterns should be explored. The similarity of subgraphs can be modeled as dependencies between the subgraphs in order to avoid exploring subgraphs that may be less relevant due to presence of other subgraphs. Specifically, this results in lateral dependencies across similar subgraphs of the same size. The lateral dependencies can be explored here to dynamically skip ETasks by choosing certain specific types of subgraphs (*e.g.*, those matching sparsest patterns). In this case, ETasks that match those specific types of patterns end up canceling other ETasks that are of different kind in order to avoid irrelevant subgraphs.

Chapter 6

Predecessor Dependencies for ETasks

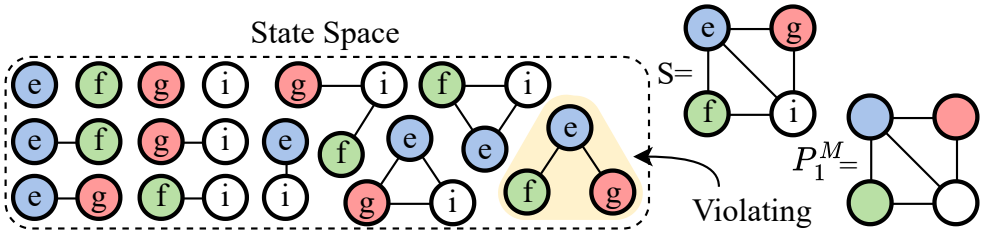
Unlike successor dependencies, predecessor dependencies are local to the explored subgraph S. Hence, we do not use special VTasks to validate predecessor dependencies, but instead perform validation on ETasks as they match RL-Paths.

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 3.2 in Chapter 3), and hence such an approach can miss containment constraints for one or more P^+.

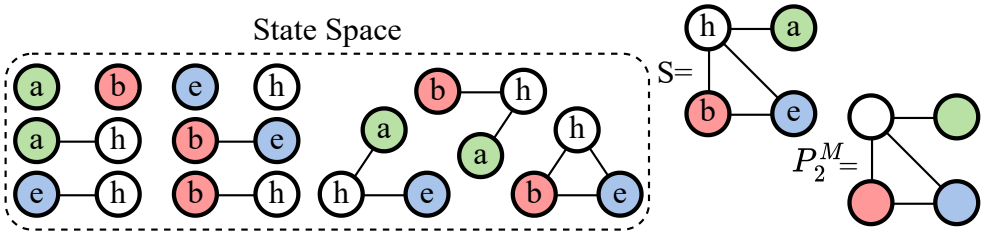
6.1 State Space & Virtual State Space

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 6.1a 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, comprising 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.



(a) Contains violating state.



(b) Does not contain violating state.

Figure 6.1: State spaces for RL-Paths that match P_1^M and P_2^M.

6.2 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 2.7), 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 6.1a. 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 6.1b 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).

Chapter 7

Evaluation

We evaluate the effectiveness of Contigra for containment constrained graph mining applications. We first evaluate the effectiveness of VTasks for successor dependencies on maximal quasi-cliques and nested subgraph queries. Then we evaluate the performance of our task management strategies such as task promotion, lateral dependencies, and RL-Path ordering. Then we evaluate the effectiveness of static analysis for predecessor dependencies on keyword search. Finally we evaluate generality of our techniques on applications without containment constraints by evaluating quasi-cliques.

7.1 Implementation Details

We develop Contigra on top of **Peregrine+**, a modified version of the state-of-the-art graph mining system Peregrine [26] which supports simultaneous exploration of multiple patterns proposed in recent works which are not open-source [18, 37]. 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 [18]), and ETasks to such patterns share their caches (*i.e.*, intra-pattern reuse [18]). 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

Data Graphs	Vertices	Edges	Labels	GCC
Amazon (AZ)	334.9K	925.9K	0	0.21
DBLP (DB)	317.1K	1.0M	0	0.31
Mico (MI)	96.6K	1.1M	28	0.41
Patents (PA)	2.7M	14.0M	36	0.07
Youtube (YT)	7.7M	50.7M	23	0.11
Products (PR)	2.4M	61.9M	46	0.13

Table 7.1: Real-world graph datasets used in evaluation.

GCC Value	Vertices	Edges	Final GCC
0.1	477.2K	3.8M	0.10
0.2	477.5K	3.7M	0.21
0.3	477.2K	3.7M	0.32
0.4	477.4K	3.7M	0.42
0.5	477.3K	3.7M	0.51

Table 7.2: Synthetic graph datasets used in evaluation.

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.

7.2 Applications, Datasets & Systems

7.2.1 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 7.1a, whereas the second query searches all tailed triangles not contained in size-6 patterns shown in Figure 7.1b. 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.

7.2.2 Datasets

Both real and synthetic graph datasets are used in our experiments. Table 7.1 and Table 7.2 show the details of the graph datasets used in our experiments. They list the number of vertices, edges, the number of unique labels, as well as the Global Clustering Coefficient (GCC). GCC measures the density of the datasets. GCC is calculated by the ratio of three times the number of triangles over the number of wedges.

Table 7.1 shows the graph datasets used in our experiments, commonly used to evaluate graph mining solutions [39, 26, 18, 29, 6]. Amazon (AZ) [50] is a co-purchasing network. In this graph, vertices represent products sold and an edge between two vertices represent products that are frequently purchased together. For example, if a user buys product A and product A is often purchased with product B, there will be an edge in the graph between the vertices representing vertex A and vertex B. DBLP (DB) [50] is a co-authorship network. Vertices represent computer science researchers and two researchers are adjacent if they have co-authored a paper. A returned subgraph will contain authors who have collaborated together on a paper. The Patents (PA) [20] graph represents citations between patents. Vertices represent patents and edges represent citations which shows related patents. Youtube (YT) [12] is a network made from crawling the YouTube site. Each vertex in the graph represent a video and each edge in the graph represents a related video that was suggested for the original video. A returned subgraph will show which videos are related to each other. Finally Products (PR) [24] is a larger co-purchasing network on Amazon products. Similar to the smaller Amazon dataset, this dataset also represent vertices as products and edges represent products that are frequently purchased together.

Table 7.2 shows the synthetic graph datasets used in our experiments. We generate these datasets using the method described in [33]. We choose BTER (Block Two Level Erdos-Renyi) graphs for generating these datasets as it generates graphs that match a provided degree distribution and a given clustering coefficient. We specify a power-law distribution for the vertices as real-world graphs are known to follow a power-law distribution, with approximately 500K vertices and vary the GCC. The final number of generated vertices and edges are shown in the Table 7.2. The number of vertices and edges are similar with only the GCC differing between the datasets. We vary the GCC from 0.1-0.5 and we show the provided GCC as well as the actual GCC of the dataset in the table.

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 7.1) require higher time budget for experiments compared to traditional applications mainly due to the computational difficulty in checking containment constraints.

Application	Baseline	Speedup
Maximal Quasi-Cliques (§7.4.1)	TThinker	**12–41700**×
Nested Subgraph Queries (§7.4.2)	Peregrine+	**5.6–379**×
Keyword Search (§7.5)	Peregrine+	**21–16000**×
Quasi-Cliques (§7.6)	Peregrine+	**2.4–7.2**×

Table 7.3: Summary of Contigra's performance.

7.2.3 Systems

We compare the performance of our techniques with two state-of-the-art systems: **Peregrine+** and **TThinker** [32]. Peregrine+ is a general graph mining system that extends Peregrine [26] by batching the exploration plans together for efficiency (explained in Section 7.1). 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 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.

7.3 Performance Summary

Table 7.3 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.

7.4 VTask Performance

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

7.4.1 VTasks for Maximality

Table 7.4 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.

7.4.2 VTasks for Nested Subgraph Queries

We evaluate two nested subgraph queries shown in Figure 7.1. 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 7.1c 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.

7.4.3 Task Management Strategies

Task Promotion.

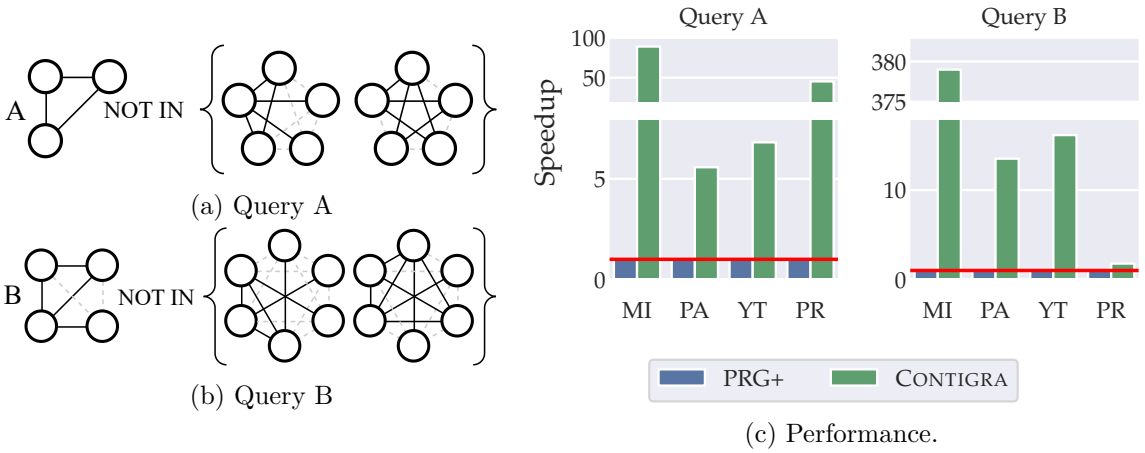
To evaluate the benefits of task promotion, we compare the hit rate of ETask caches with and without task promotion enabled in maximal quasi-cliques application. Figure 7.2 shows that cache hit rates can rise to 73% with task promotion from only 48% when it is disabled, as redundant operations are cached instead of recomputed.

Lateral Dependencies.

We quantify the effectiveness of VTask cancellation via lateral dependencies on the speedups over the baseline. Figure 7.3 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



(a) Query A

(b) Query B

(c) Performance.

Figure 7.1: Nested subgraph queries.

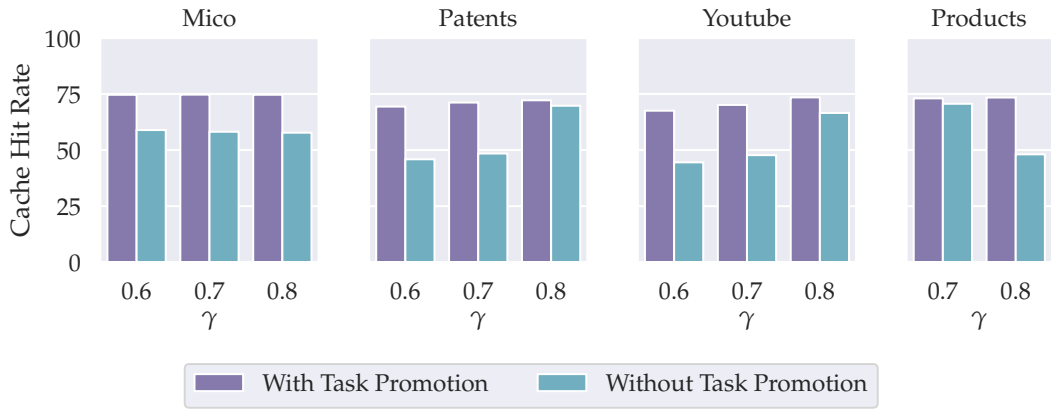


Figure 7.2: Cache hit rates with and without task promotion.

the effectiveness of our heuristics for prioritizing RL-Paths. Figure 7.4 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.

7.5 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 $k + 1$.

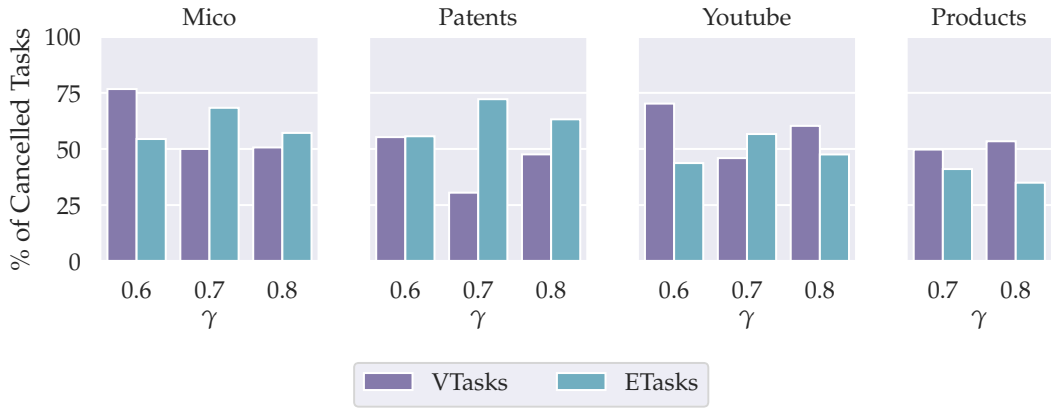


Figure 7.3: Task cancellations due to lateral dependencies.

Figure 7.5 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 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 7.6 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 7.7 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.

7.6 Generality of Task Fusion & Promotion

We further evaluate the generality of task fusion and task promotion for applications without containment constraints. We run quasi-cliques without maximality constraints but with task fusion and task promotion between ETasks enabled and compare the performance to the

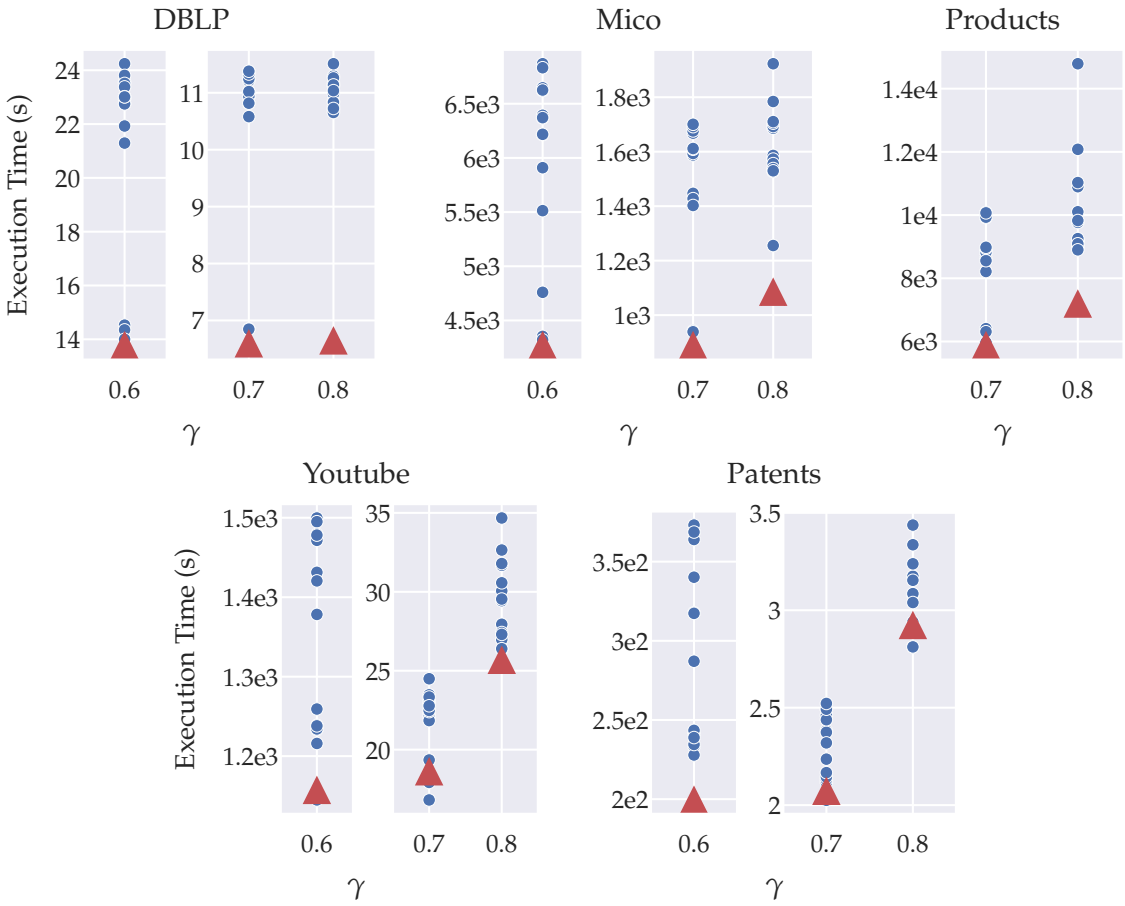


Figure 7.4: Executions with different RL-Path orderings. The ordering picked by our heuristic is marked with a red triangle.

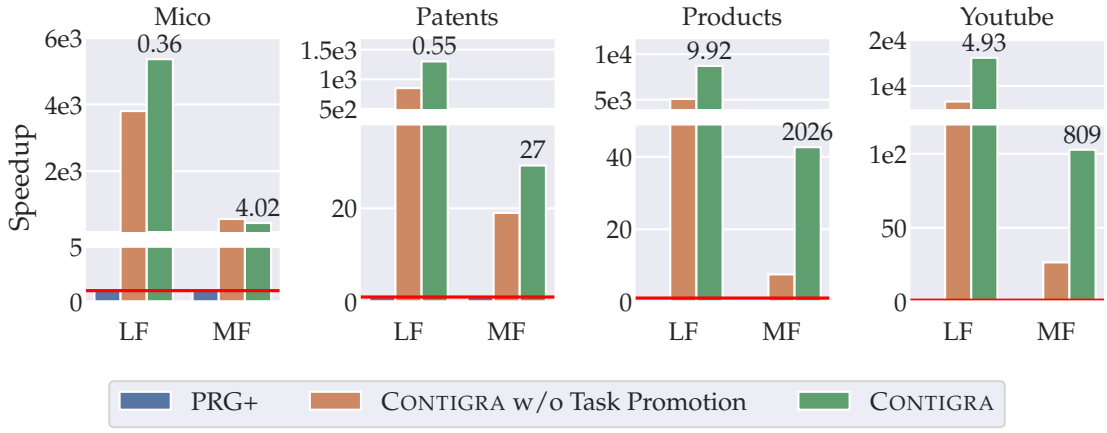


Figure 7.5: Performance of CONTIGRA and Peregrine+ for keyword search with most frequent (MF) and less frequent labels (LF). Numbers on top of bars indicate CONTIGRA execution times (sec).

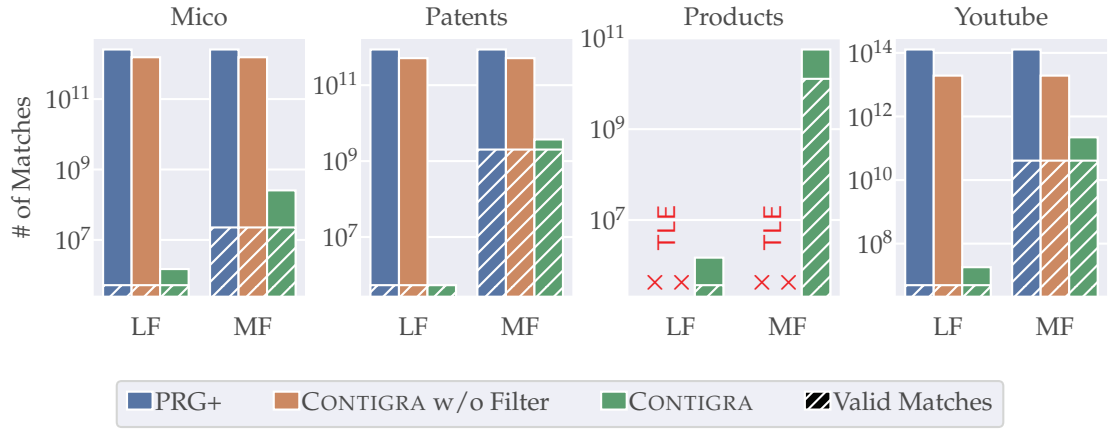


Figure 7.6: Number of matches checked for constraints in keyword search. TLE indicates executions that did not complete in 24 hours.

Peregrine+ baseline without these techniques. Figure 7.8 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.

To further drill down on the impacts of our strategies, we also evaluate the benefits in task fusion provided by bridging gaps. Figure 7.9 shows the execution of CONTIGRA with and without bridging gaps. Bridging gaps has up to 5.88× speedup due to the improvements in cache reuse.

7.6.1 Generality of RL-Path Ordering

The generality of RL-Path ordering is evaluated in Figure 7.10. We use the synthetic datasets for evaluation. Figure 7.10 shows the execution time for two opposing strategies for prioritizing RL-Paths and how dataset density affects our heuristics. We observe that when the target patterns are very dense (ie. $\gamma = 0.8$) the density of the dataset does not matter and purely focusing on the target patterns is the correct choice. As less dense target patterns are added for $\gamma = 0.7$, the density of the dataset begins to affect the execution time. With $\gamma = 0.6$, there is a mix of dense and sparse target patterns and the density of the dataset has a greater effect on execution time. These experiments further reinforce the need for our heuristics to check both the density of the target patterns and the density of the datasets if necessary.

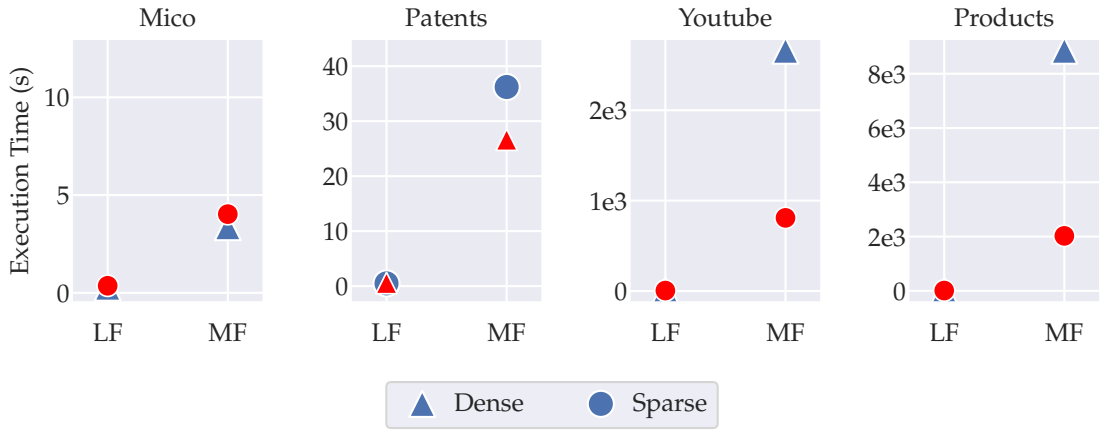


Figure 7.7: Executions with different RL-Path orderings in keyword search. The ordering picked by our heuristic is marked in red.

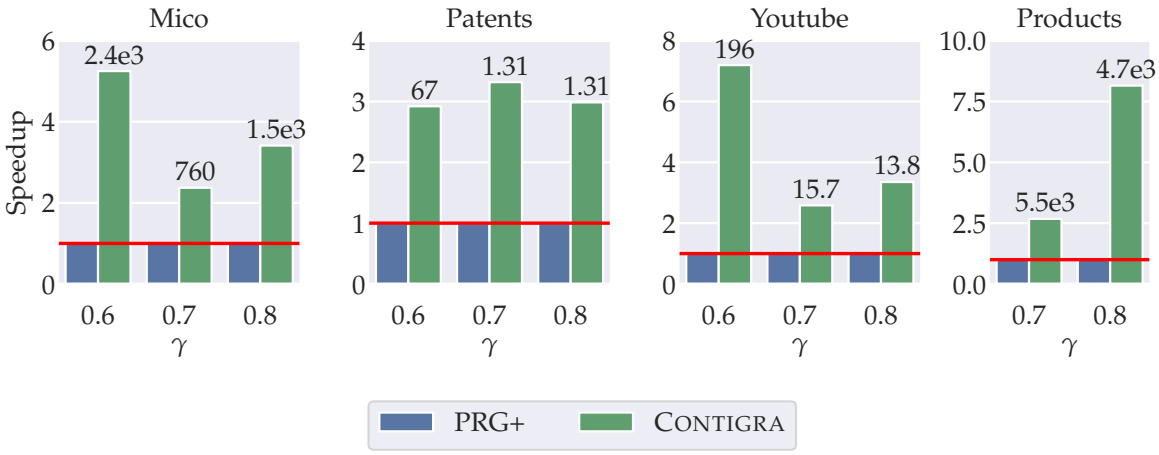


Figure 7.8: Performance of CONTIGRA and SumPA for quasi-cliques without maximality constraint. Numbers on top of bars indicate execution times (in seconds) for CONTIGRA.

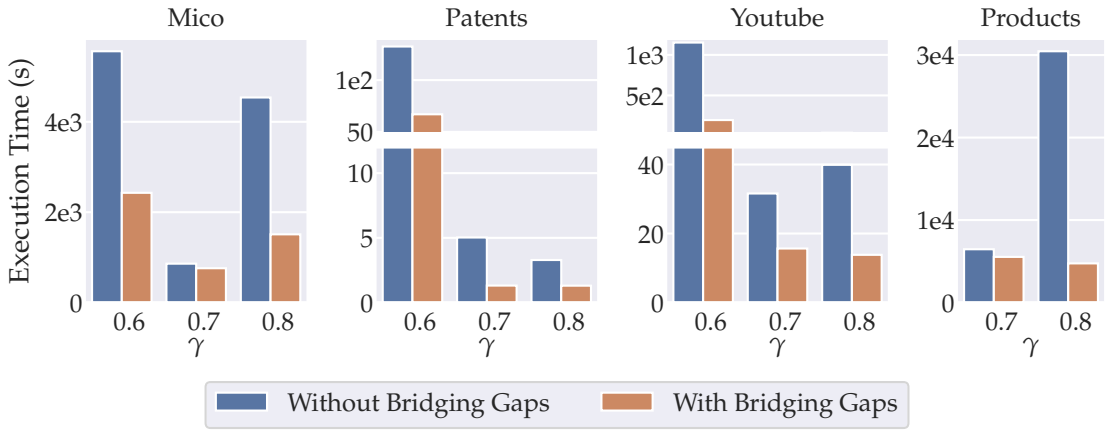


Figure 7.9: Executions showing the effect of including and not including the bridging gaps technique for γ-quasi-cliques.

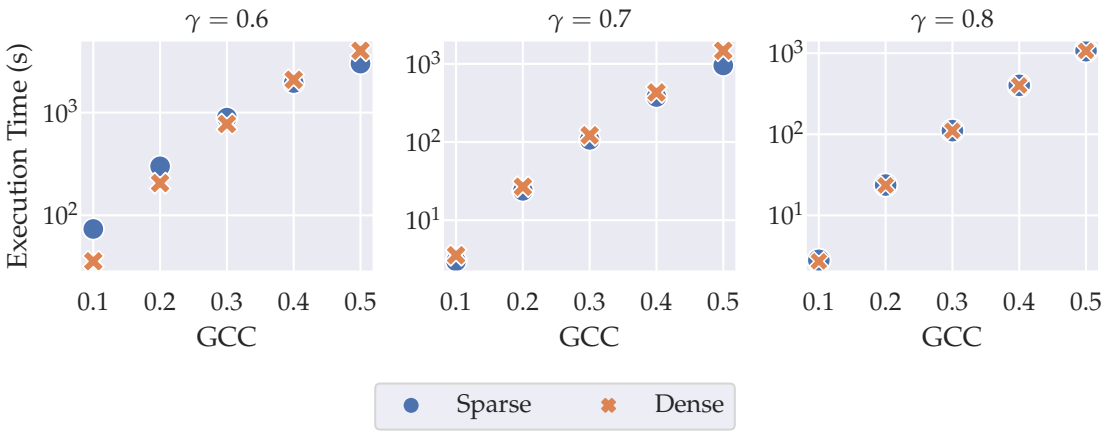


Figure 7.10: Executions evaluating different RL-Path orderings in γ-quasi-cliques on BTER datasets of varying density.

	$\gamma = 0.6$		
	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 7.4: 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.

Chapter 8

Related Work

There are many general-purpose graph mining systems and custom algorithms developed in the literature. However to the best of our knowledge, CONTIGRA is the first general execution model for containment constrained applications.

8.1 Graph Mining & Pattern Matching.

General-purpose graph mining systems [46, 15, 48, 26, 39, 56, 10, 5, 18, 6, 9] combine efficient pattern matching strategies with a programmable match processing module to support multiple different graph mining applications.

These systems can be divided into two main types, pattern-oblivious and pattern-aware. Pattern-oblivious systems [46, 15, 48, 56, 10, 5] explore subgraphs through iterative extensions by edges or vertices without taking pattern structure into account. Arabesque [46] was the first general graph mining system and it uses a "filter-process" model that filters out subgraphs during exploration based on specific properties before passing it to a user-defined process. However, Arabesque encounters significant bottlenecks due to memory limitations as a result of its breadth-first search exploration model. Subsequent works such as Fractal [15], Pangolin [10], Tesseract [5], and other pattern-aware systems overcome the memory bottleneck by changing to a depth-first search exploration model. Some pattern-oblivious systems such as Arabesque [5] and Tesseract [5] also require applications to be anti-monotonic. This limits the type of applications they can support because not all applications with containment constraints are anti-monotonic, such as maximal quasi-cliques.

Pattern-aware systems [26, 18, 6, 9] exploit structural properties of target patterns in the pattern matching module, enabling powerful optimizations [28, 29, 7]. Peregrine [26] introduced the concept of pattern-awareness through analysing structural properties of target patterns. Peregrine also introduced a pattern-based programming model that allows users to easily specify graph mining applications. Furthermore Peregrine also defined and natively supports AntiVertices [27] and can support simple nested subgraph queries. SumPA [18] extends Peregrine by using Peregrine's structure analysis and taking Peregrine's matching orders to merge multiple target patterns together into its execution plan. This allows SumPA to simultaneously explore multiple target patterns. GraphPi [43] is another pattern-aware

system that uses a cost model to determine the optimal schedule for execution. However these systems cannot handle containment constraints as it is not possible to specify a relationship between target patterns and applications with containment constraints will require additional processing after execution to ensure containment constraints are satisfied.

Pattern-based graph mining also benefits from research in pattern matching systems [2, 37, 43, 41, 44, 11, 38, 8, 53, 25, 30] that enable techniques from architecture research [44, 11, 8], dataflow systems [41, 2], sampling theory [25, 30], and compilers [43, 37]. None of these solutions consider containment constraints across subgraphs, hence requiring users to implement the constraint checking logic in UDF, leading to redundant operations and inefficiency.

8.2 Maximal Quasi-Cliques

There are several specialized maximal quasi-clique solutions [35, 40, 54, 42, 13, 32, 19]. Most recent solutions are based on Quick [35] which incorporates pruning techniques from prior algorithms like Crochet [40] and Cocain [54]. Newer algorithms such as KernelQC [42] and Quick+ from GThinker [19] add even more pruning strategies. All these algorithms reduce the search space by pruning sparse regions of the graph. However, they all rely on post-processing to eliminate matches that are not maximal as the pruning strategies are unable to remove all non-maximal results. These solutions are unsuitable for mining maximal quasi-cliques on large graphs as they are only able to handle specific sizes that allow them to heavily prune the original data graph. By removing the majority of the data graph, they limit their search space to a size manageable size. On the other hand, Contigra efficiently executes maximal quasi-clique without post-processing and can further support general constrained applications.

8.3 Graph Keyword Search

There are many specialized keyword search algorithms [16, 3, 31, 21, 52, 47]. BANKS [3] introduced the concept of keyword search on graph data by enabling search across relational tables connected by foreign keys. BANKS searches by starting at a keyword and working backwards to find an answer, Kacholia et al [31] work enhances BANKS with bi-directional search that speculatively searches forward from arbitrary vertices in order to find key words. BLINKS [21] then enhanced bi-directional graph keyword search [31] using a novel indexing scheme, allowing faster search as well as tunable time-space tradeoff. More recently, keyword search has been applied to RDF [16] and knowledge graphs [52]. Elbassuoni et al. [16] performs keyword search by using an inverted index from the keywords to the labels and checks each edge after it has been added to determine if the edge is valid or not. 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.

Chapter 9

Conclusions & Future Directions

9.1 Conclusion

This thesis presents Contigra for graph mining with *containment constraints*. Contigra models containment constraints as cross-task *dependencies* and exploits these dependencies to develop several techniques. Using successor dependencies, task fusion allows exploration and validation tasks to be merged together to maximize cache reuse. Task promotion prevents redundant work from occurring by converting validation tasks to exploration tasks eliminating the need for exploration to begin from scratch. Furthermore within task fusion, techniques like bridging gaps and efficient RL-Path ordering further refine execution by optimizing cache reuse and task cancellation. Lateral dependencies enable dynamic cancellation of unnecessary validation tasks, significantly reducing the number of constraint checks. Finally with predecessor dependencies, entire exploration tasks can be skipped through static analysis and eager filtering allows exploration tasks to stop prematurely if the constraints are determined to be unsatisfiable. These techniques collectively eliminate redundant work and maximizes cache reuse, enabling containment constraints to be checked efficiently during executing, thus removing the need for matches to be examined individually after exploration. For applications with containment constraints, Contigra is significantly faster than state-of-the-art systems. Moreover Contigra scales to massive workloads that could not be handled by existing systems. Finally, the techniques developed in this thesis are are also applicable to applications that do not have containment constraints, by modeling their execution requirements in terms of dependencies across subgraphs.

9.2 Future Directions

Contigra is the first to consider a general set of containment constrained problems. As such Contigra can provide the basis for future systems to efficiently support containment constraints by building on top of or extending Contigra.

A possible extension to Contigra is its interaction with modern techniques like Subgraph Morphing [29]. Given a query, subgraph morphing generates multiple equivalent queries and uses a cost-model to determine the most efficient execution plan. If we incorporate subgraph morphing into Contigra, there are some interesting challenges that occur. The cost model from subgraph morphing needs to change. Subgraph morphing does all of its analysis statically before choosing the optimal set of queries for execution. However, Contigra has dynamic tasks. As these tasks can be cancelled or spawned at runtime, the cost model will need to take into account the effect of task fusion and task promotion. This may require dynamically switching execution plans on the fly or fine-tuning the execution plan depending on the specific exploration task.

Another possible extension to Contigra is applying the strategies developed for graph pattern mining systems to graph databases. Graph databases like Neo4j [1] already support some types of containment queries such as Nested Subgraph Queries with their nested MATCH clauses. However, graph databases do not have any support for maximality and they use a completely different execution model compared to graph pattern mining systems.

Finally, we can also generalize containment constraints to a wider scope. Contigra supports matches that are fully contained or are fully absent from another match. Contigra does not consider constraints that are partially contained within another match, or overlapping matches. For example, we may want all house graphs whose triangle portion of the house is not part of any square with a diagonal. This increases the complexity of the relationships between the patterns as now there can be multiple target patterns for VTasks. Currently in Contigra, each VTasks only needs to be responsible for one target pattern. But when queries are overlapping, we will now need to consider how to incorporate these partial patterns into VTasks.

Bibliography

[1] Neo4j. https://neo4j.com/.

[2] 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.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings 18th International Conference on Data Engineering*, pages 431–440, Feb 2002.

[4] 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.

[5] 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.

[6] 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 2023, page 47–61, 2022.

[7] 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 2023, page 413–426. Association for Computing Machinery, 2023.

[8] 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, New York, NY, USA, 2022. Association for Computing Machinery.

[9] 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.

[10] 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.

[11] 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)*, pages 581–594, 2021.

[12] 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.

[13] 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.

[14] 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, New York, NY, USA, 2022. Association for Computing Machinery.

[15] 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.

[16] 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, New York, NY, USA, 2011. Association for Computing Machinery.

[17] 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.

[18] 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.

[19] Guimu Guo, Da Yan, M. Tamer Özsu, Zhe Jiang, and Jalal Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, dec 2020.

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

[21] 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, New York, NY, USA, 2007. Association for Computing Machinery.

[22] 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.

[23] 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, 06 2005.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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, page 162–181, 2023.

[30] 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, New York, NY, USA, 2023. Association for Computing Machinery.

[31] 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. VLDB Endowment, 2005.

[32] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. Parallel mining of large maximal quasi-cliques. *The VLDB Journal*, 31(4):649–674, nov 2021.

[33] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, 2014.

[34] 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.

[35] 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, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[36] Anjum Ibna Matin, Sawgath Jahan, and Mohammad Rezwanul Huq. Community recommendation in social network using strong friends and quasi-clique approach. In *8th International Conference on Electrical and Computer Engineering*, pages 453–456, 2014.

[37] 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.

[38] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Operating Systems Review*, 55(1):21–37, June 2021.

[39] 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.

[40] 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, New York, NY, USA, 2005. Association for Computing Machinery.

[41] 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. USENIX Association, April 2021.

[42] Seyed-Vahid Sanei-Mehri, Apurba Das, Hooman Hashemi, and Srikanta Tirthapura. Mining largest maximal quasi-cliques. *ACM Trans. Knowl. Discov. Data*, 15(5), apr 2021.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Haixun Wang and Charu Aggarwal. *A Survey of Algorithms for Keyword Search on Graph Data*, pages 249–273. Springer, 02 2010.

[48] 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.

[49] 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.

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

[51] Jianye Yang, Wu Yao, and Wenjie Zhang. Keyword search on large graphs: A survey. *Data Science and Engineering*, 6(2):142–162, March 2021.

[52] Yueji Yang, Divykant Agrawal, H. V. Jagadish, Anthony K. H. Tung, and Shuang Wu. An efficient parallel keyword search engine on knowledge graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 338–349, 2019.

[53] 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.

[54] 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, New York, NY, USA, 2006. Association for Computing Machinery.

[55] Shijie Zhang, Jiong Yang, and Wei Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *Proc. VLDB Endow.*, 3(1–2):1185–1194, sep 2010.

[56] 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 IEEE International Conference on Data Engineering*, ICDE '20, pages 673–684, 2020.