Subgraph Matching: on Compression and Computation

Miao Qiao
Massey University
New Zealand
m.qiao@massey.ac.nz

Hao Zhang Hong Cheng
The Chinese University of Hong Kong
Hong Kong
{hzhang,hcheng}@se.cuhk.edu.hk

ABSTRACT

Subgraph matching finds a set $\mathcal{I}$ of all occurrences of a pattern graph in a target graph. It has a wide range of applications while suffers an expensive computation. This efficiency issue has been studied extensively. All existing approaches, however, turn a blind eye to the *output crisis*, that is, when the system has to materialize $\mathcal{I}$ as a preprocessing/intermediate/final result or an index, the cost of the export of $\mathcal{I}$ dominates the overall cost, which could be prohibitive even for a small pattern graph.

This paper studies subgraph matching via two problems. 1) Is there an ideal compression of $\mathcal{I}$? 2) Will the compression of $\mathcal{I}$ reversely boost the computation of $\mathcal{I}$? For the problem 1), we propose a technique called VCBC to compress $\mathcal{I}$ to $\mathsf{code}(\mathcal{I})$ which serves effectively the same as $\mathcal{I}$. For problem 2), we propose a subgraph matching computation framework CBF which computes $\mathsf{code}(\mathcal{I})$ instead of $\mathcal{I}$ to bring down the output cost. CBF further reduces the overall cost by reducing the intermediate results. Extensive experiments show that the compression ratio of VCBC can be up to 10^5 which also significantly lowers the output cost of CBF. Extensive experiments show the superior performance of CBF over existing approaches.

PVLDB Reference Format:
Miao Qiao, Hao Zhang, Hong Cheng. Subgraph Matching: on Compression and Computation. *PVLDB*, 11(2): 176-188, 2017.
DOI: 10.14778/3149193.3149198

1. INTRODUCTION

The subgraph matching of a pattern graph p on a target graph d reports the set $\mathcal{I}_p$ of all the subgraphs of d that are isomorphic to p. This problem underpins various analytical applications based on the significant role graphs play in modelling the interconnectivity of objects in areas such as biology, chemistry, communication, transportation and social science. For example, by letting pattern graphs have semantic/statistical meanings, subgraph matching is used to monitor terrorist cells in activity networks [10], identify properties of recommendation/social networks [18, 23], and decode functions of biological networks [5]. Subgraph matching naturally becomes a fundamental construct of the query language of graph databases such as Neo4j, AgensGraph and SAP HANA.

Unfortunately, the computation of subgraph matching is NP-complete [11]. The basic approach is a brute-force search over all the subgraphs of d. Ullman's backtracking algorithm [30] has sparked studies on different searching orders, pruning rules and neighborhood indexes (see [22] as an entrance). However, these techniques assume that the target graph fits into the memory of a machine, which does not hold on many real graphs nowadays[1]. This fact has motivated the research on two approaches: using external memory and using a cluster of machines. A common issue to both approaches is how to arrange the materialization caused by the memory limit.

The first approach [9, 16, 17, 25, 26] is investigated under external memory (EM) model [3] where cost is defined as the total number of I/Os performed. An I/O transfers a block of B words between the main memory and the disk. Subgraph matching has two settings in EM model, *subgraph listing* [9] and *subgraph enumeration* [26]. Subgraph listing requires the system to materialize $\mathcal{I}_p$ whereas subgraph enumeration does not. Such a distinction separates the *output cost*—the $\Theta(\frac{|\mathcal{I}_p|}{B})$ I/Os of exporting $\mathcal{I}_p$ to the disk—from the *enumeration cost*—the cost of subgraph enumeration [16, 26].

The second approach is to study subgraph matching [1, 2, 19, 20, 21, 27, 29] on parallel computing platforms such as MapReduce. Brute-force search algorithms for subgraph matching are parallelized in two styles, BFS and DFS, differ on whether intermediate results are materialized or not.

BFS-style algorithms [20, 21, 29] are iterative. In its final iteration, $\mathcal{I}_p$ is computed from an intermediate result $\mathcal{I}_{p'}$ of the previous iteration—the instance set of another pattern graph p'. p' is normally smaller than p by a node or an edge. Such a process applies unless p has only one node/edge. The system must materialize and shuffle $\mathcal{I}_{p'}$ to initiate the computation of $\mathcal{I}_p$. This is a severe burden: shuffle is the most expensive operation in a parallel system such as MapReduce.

DFS-style solutions [1, 2, 19, 27] do not materialize intermediate results. The target graph is partitioned, replicated and shuffled before the one-round parallel computation takes

Proceedings of the VLDB Endowment, Vol. 11, No. 2

DOI: 10.14778/3149193.3149198

[1]Consider Facebook as an example: with 10^9 daily active users http://newsroom.fb.com/company-info/ and an average of 190 friends per user http://arxiv.org/abs/1111.4503, the graph requires 1.6 petabytes of storage.

place. DFS-style solutions have some theoretical analysis [2], but their practical performances on real target graphs may not be appealing [20] compared to BFS-style solutions.

Though the instance set $\mathcal{I}_p$ of a subgraph matching may be massive in this big data era, its materialization could be demanded or even inevitable in practice. This is especially true when subgraph matching is the basic form of a query in a graph database system such as Neo4j. A traditional database materializes views for query optimization, which, in the context of a graph database, is to materialize the instance set of a subgraph query. This practice avoids repetitive computations of frequent queries and common sub-queries, saves system resources, shortens query delay and enhance concurrency. Besides, BFS-style parallelisms inevitably materialize $\mathcal{I}_p$. A persistent $\mathcal{I}_p$ is also demanded when subgraph matching serves as a preprocessing/intermediate step of a application [10, 18, 23, 5]; otherwise any unexpected error will trigger a re-computation of $\mathcal{I}_p$ — could be even more expensive than materializing $\mathcal{I}_p$.

When the system has to materialize the instance set $\mathcal{I}_p$ as a preprocessing result, intermediate result, index, or final result, etc., existing solutions turn a blind eye to the *output crisis* of subgraph matching: the $\Omega(\frac{|\mathcal{I}_p|}{B})$ I/Os on listing $\mathcal{I}_p$ to the disk becomes a *lower bound* of the overall cost no matter how deftly one computes $\mathcal{I}_p$. This observation has led us to investigate subgraph matching via two problems:

1. Is there an *ideal compression* on the instance set $\mathcal{I}_p$?
2. Will the compression of $\mathcal{I}_p$ reversely boost the computation of subgraph matching?

Our contributions. This is the first attempt, in the literature, on resolving the *output crisis* of subgraph matching using output compression. Output compression is vertical to input compression techniques [14] which focus on downsizing the size of the target graph in a subgraph matching.

This paper proposes the *vertex-cover based compression* (VCBC) technique to compress $\mathcal{I}$ to $\mathsf{code}(\mathcal{I})$. VCBC features an impressive compression ratio, that is, the size of $\mathsf{code}(\mathcal{I})$ is significantly smaller than that of $\mathcal{I}$. Moreover, $\mathsf{code}(\mathcal{I})$ serves effectively the same as a materialized $\mathcal{I}$, that is, the decompression process of VCBC restores $\mathcal{I}_p$ in a streamed manner from $\mathsf{code}(\mathcal{I})$ in $\Theta(\frac{|\mathcal{I}_p|}{B})$ I/Os. VCBC, together with general compression techniques, provides an effective *storage solution* for subgraph matching. Such a storage solution is desirable in three cases. 1) $\mathcal{I}_p$ is prohibitively large such that existing solutions cannot afford materializing $\mathcal{I}_p$. 2) The materialization of $\mathcal{I}_p$ constitutes the performance bottleneck of an algorithm. 3) The access of $\mathcal{I}_p$ is not efficient enough unless $\mathcal{I}_p$ is placed on a faster yet more expensive medium, for example, SSD or the main memory.

A perhaps more interesting contribution is the *Crystal-Based computation Framework* (CBF). CBF reduces the overall cost of subgraph matching by materializing $\mathsf{code}(\mathcal{I}_p)$ instead of $\mathcal{I}_p$. Such a reduction is significant especially when the output cost is the bottleneck of the subgraph matching computation. Moreover, in terms of enumeration — computing $\mathcal{I}_p$ without materializing the result, CBF outperforms the existing approaches by up to orders of magnitude. In particular, CBF excels in matching complex pattern graphs against dense target graphs where all existing solutions fail, as will be shown in our empirical studies.

Table 1: Notations

Symbol	Description
p, d	The pattern graph p and target graph d.
n_p, m_p	$n_p = \|V(p)\|, m_p = \|E(p)\|$.
$g(V')$	The induced subgraph of g on vertex set V'.
$\mathsf{code}(\cdot)$	The compressed code of a piece of data.
$\rho(\cdot)$	The compression ratio: Equation 1.
$\mathcal{I}_p$	The instance set of p — the set of subgraphs of d that are isomorphic to p.
f_g	The instance-bijection of instance $g \in \mathcal{I}_p$.
ord_p	The order on $V(p)$ for symmetry breaking.
$\mathcal{H}_{V_c}(g)$	The helve of instance g: $f_g(u)$ for all $u \in V_c$.
$\mathcal{H}(\mathcal{I}_p)$	The set of helves of instances in $\mathcal{I}_p$.
$\mathsf{Img}_p(u\|h)$	$\{f_g(u)\|g \in \mathcal{I}_p\|h\}$ of a node u.
$\mathcal{I}_p\|h$	The set of instances in $\mathcal{I}_p$ with helve h.
$\{V_c, \lambda, \mathcal{P}\}$	A core-crystal decomposition of p.
V_c	A vertex cover of p.
$\mathsf{core}(p)$	$p(V_c)$, the induced subgraph of p on V_c.
$\overline{V_c}$	The complement of V_c, that is, $V(p) \setminus V_c$.
$\mathcal{P}$	$p_1, p_2, \cdots, p_\lambda$, λ subgraphs of p, where p_i is a crystal $\mathcal{Q}_{x_i,y_i}$, for $i \in [1, \lambda]$.
$\mathcal{Q}_{x,y}$	A graph with y nodes fully connected to a $\mathcal{C}_x$.
$\mathcal{C}_x$	A clique of size x.
M	Size of the main memory.
B	Size of a disk block.
σ, η	Two constants defined in the assumption.

Organization. Section 2 formally defines subgraph matching and the two problems to be addressed in this paper. Sections 3 studies the compression problem while Section 4 investigates the computation problem. Section 5 surveys related work. Section 6 evaluates our techniques via extensive experimentation. Section 7 concludes the paper.

2. PRELIMINARIES

We now formally introduce all the definitions. Table 1 aggregates all the notations used in the paper.

2.1 Subgraph Matching

This paper focuses on the subgraph matching on unlabeled and undirected graphs. A **graph** g consists of a set $V(g)$ of vertexes and a set $E(g)$ of edges. A vertex is also called a node. An edge $e(u, v)$ connects two vertexes u and v in $V(g)$. $e(u, v)$ is *incident to* both u and v. The degree of a node v is the total number of edges incident to v. A graph g is a **clique** if for every pair u, v of nodes in $V(g)$, edge $(u, v) \in E(g)$. A clique of size k is denoted as $\mathcal{C}_k$.

Let g_1 and g_2 be two graphs. The intersection $g_1 \cap g_2$ of g_1 and g_2 is a graph with vertex set $V(g_1) \cap V(g_2)$ and edge set $E(g_1) \cap E(g_2)$. If $g_1 \cap g_2 = g_1$, then g_1 is a **subgraph** of g_2. The **induced subgraph** $g(V')$ of a graph g on a vertex set V' is a graph with vertex set $V' \cap V(g)$ and edge set $E(g)|V'$ where $E(g)|V' = E(g) \cap (V' \times V')$.

Definition 1 (Graph Isomorphism [12]). *Given two graphs g_1 and g_2, an isomorphism from g_1 and g_2 is a bijection $f : V(g_1) \mapsto V(g_2)$ such that $(u, v) \in E(g_1)$ if and only if $(f(u), f(v)) \in E(g_2)$. If there is an isomorphism from g_1 to g_2, then we say g_1 is isomorphic to g_2.*

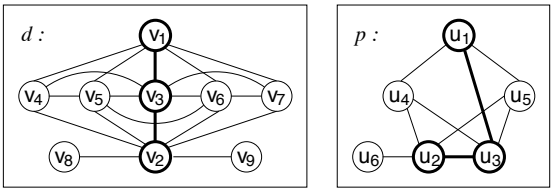


Figure 1: Target graph d and pattern graph p

Definition 2 (Graph Matching). *For a given target graph d and a given pattern graph p, subgraph matching reports the set $\mathcal{I}_p$ of all the subgraphs of d that are isomorphic to p. Denote $|V(p)|$ as n_p, $|E(p)|$ as m_p.*

A subgraph g of d is an **instance** of p if it is isomorphic to p. In other words, $g \in \mathcal{I}_p$ if and only if g is an instance of p. We thus call $\mathcal{I}_p$ the **instance set** of p.

Example 1. We use a *running example* of a subgraph matching on target graph d and pattern graph p in Figure 1.

Let $V' = \{v_1, v_2, \cdots, v_5\}$. $d(V')$ is the induced subgraph of d on set V'. Subgraph g with vertex set $V(g) = V' \cup \{v_6\}$ and edge set $E(g) = E(d(V')) \cup \{(v_2, v_6)\}$ is an instance of p with an isomorphism f that maps v_i to u_i, for $i \in [1, 6]$.

One instance g may have multiple isomorphisms to p. The standard technique of **symmetry breaking** (SimB) [15] validates exactly one isomorphism $f_g : V(p) \mapsto V(g)$ for each instance g. f_g is called the **instance-bijection** of g.

Specifically, SimB selects a set $\mathsf{ord}_p \subseteq V(p) \times V(p)$ of node pairs in the pattern graph. For each pair $\langle u, v \rangle$ in ord_p, a partial order $\prec$ is imposed such that $u \prec v$. Besides, SimB defines an arbitrary *total order* on target graph nodes $V(d)$. By default, for $u, v \in V(d)$, $u < v$ if the identifier of u is smaller than that of v. Given an instance $g \in \mathcal{I}_p$, an isomorphism f from p to g is valid if $f(u) < f(v)$, for any $u \prec v$. Each instance g has exactly one valid isomorphism f_g under ord_p. f_g is called the instance-bijection of g.

Example 2. In Figure 1, pattern graph p uses $\mathsf{ord}_p = \{\langle u_4, u_5 \rangle\}$ for symmetry breaking. In Example 1, instance g has an isomorphism f. g has another isomorphism f' which is the same as f except for $f'(v_4) = u_5$ and $f'(v_5) = u_4$. ord_p invalidates f'^{-1} since $f'^{-1}(u_4) > f'^{-1}(u_5)$ violates $u_4 \prec u_5$. The instance-bijection f_g of g under ord_p is

$$f_g(u_i) = v_i, \text{ for } \forall i \in [1, 6].$$

A mapping function maps a source to its image. For an instance g and its instance-bijection f_g, we call $f_g(u)$ the **image** of u under g. We call $\mathsf{Img}_p(u) = \{f_g(u) | g \in \mathcal{I}_p\}$ the **image set** of u under $\mathcal{I}_p$ where $\mathcal{I}_p$ is the instance set of p.

Example 3. Example 2 shows the instance-bijection f_g of g. $f_g(u_1) = v_1$ so the image of u_1 is v_1, and thus $v_1 \in \mathsf{Img}_p(u_1)$.

2.2 Assumptions

This paper discusses subgraph matching in external memory (EM) model with two assumptions. In EM model, an I/O transfers a block of B words between the disk and the memory of a machine. The memory size is M words. The cost is defined as the total number of I/Os performed. We assume that the pattern graph has $O(1)$ nodes and the target graph has $O(M)$ nodes. Specifically, we assume:

A_1 $n_p = |V(p)| = O(1)$, that is, $n_p < \sigma$ for a constant σ.

A_2 $|V(d)| = O(M)$, that is, $|V(d)| < \frac{\eta}{\sigma} M$ for a constant $\eta < 1$ such that $V(d)$ fits in a memory of M/σ words.

2.3 D-Optimal Compression

A compression approach includes a compression algorithm and a decompression algorithm. Let D be a piece of data. The **code** of D, denote as $\mathsf{code}(D)$, is the compressed form of D. D can be restored from $\mathsf{code}(D)$ if the compression is lossless. The **compression ratio** on D is defined as

$$\rho(D) = \frac{|\mathsf{code}(D)|}{|D|}. \quad (1)$$

In EM model, any algorithm that lists D needs $\Omega(\frac{|D|}{B})$ I/Os, we thus define the notion of an "optimal" compression.

Definition 3 (D-Optimal Compression). *A compression approach is d-optimal if the decompression is output-sensitive—D can be restored from $\mathsf{code}(D)$ in $\Theta(\frac{|D|}{B})$ I/Os.*

In other words, a d-optimal compression guarantees that $\mathsf{code}(D)$ serves effectively the same as a materialized D.

2.4 Problems

For a subgraph matching on target graph d and pattern graph p, this paper focuses on two problems below.

Problem 1. *Given $\mathcal{I}_p$ of a pattern graph p, is there a d-optimal compression approach for $\mathcal{I}_p$ with a high $\rho(\mathcal{I}_p)$?*

Problem 2. *Given a target graph d and a pattern graph p, how to efficiently compute $\mathsf{code}(\mathcal{I}_p)$?*

Problem 2 is dependent on the solution of Problem 1: the cost for exporting $\mathsf{code}(\mathcal{I}_p)$ to the disk in Problem 2 is solely determined by the compression ratio $\rho(\mathcal{I}_p)$ in Problem 1. Thus, we partition the **overall cost** of Problem 2 into:

- **Output cost**: the cost on exporting the final results.
- **Enumeration cost**: the overall cost assuming that the export of the final results is for free.

3. VC BASED COMPRESSION

This section provides a *positive* answer to Problem 1 by devising a *vertex-cover based compression* (VCBC) technique.

VCBC is a compression of $\mathcal{I}_p$ based on a *vertex cover* of the pattern graph p. A **vertex cover** of p is a set V_c of nodes in $V(p)$ that jointly cover all the edges in $E(p)$ — a vertex v **covers** an edge e if e is incident to v. Formally, V_c is a vertex cover of p if for $\forall e(u, v) \in E(p)$, $V_c \cap \{u, v\} \neq \emptyset$.

To explain VCBC, we define the helve of an instance of p.

Definition 4 (Helve). *Let $V_c = \{u_1, u_2, \ldots, u_k\}$ be a vertex cover of p. Let g be an instance of p. The **helve** of g is the vectored images of V_c under the instance-bijection f_g:*

$$\mathcal{H}_{V_c}(g) = (f_g(u_1), f_g(u_2), \ldots, f_g(u_k)).$$

It is also denoted as $\mathcal{H}(g)$ if V_c is obvious in the context. Similarly, the helves of an instance set $\mathcal{I}$ is defined as

$$\mathcal{H}(\mathcal{I}) = \{\mathcal{H}(g) | g \in \mathcal{I}\}.$$

Table 2: $\mathsf{code}(\mathcal{I}_p|h)$ with $h = (v_1, v_2, v_3)$.

$u \in V(p)$	u_1	u_2	u_3	u_4	u_5	u_6
$\mathsf{Img}_p(u\|h)$	v_1	v_2	v_3	v_4, v_5, v_6	v_5, v_6, v_7	$v_4, v_5, \ldots, v_9$

Example 4. In Figure 1, the pattern graph p has a vertex cover $V_c = \{u_1, u_2, u_3\}$. In Example 2, the instance-bijection f_g maps, for a instance g with $V(g) = \{v_i | i \in [1, 6]\}$, $u_i \in V(p)$ to v_i. The helve of g is therefore the images of V_c under g, $\mathcal{H}(g) = (g(u_1), g(u_2), g(u_3)) = (v_1, v_2, v_3)$.

3.1 Compression

Recall that Definition 4 defines $\mathcal{H}(\mathcal{I}_p) = \{\mathcal{H}(g) | g \in \mathcal{I}_p\}$ for instance set $\mathcal{I}_p$ under a vertex cover V_c. Let $h_1, h_2, \ldots, h_l$ be the $l = |\mathcal{H}(\mathcal{I}_p)|$ helves in $\mathcal{H}(\mathcal{I}_p)$. For each helve h_i, $i \in [1, l]$, VCBC compresses $\mathcal{I}_p|h_i$ to $\mathsf{code}(\mathcal{I}_p|h_i)$ in 3 steps:

C_1 Group the instances in $\mathcal{I}_p$ by their helves. Define the **conditional instance set** $\mathcal{I}_p|h_i$ of h_i as
$$\mathcal{I}_p|h_i = \{g | \mathcal{H}(g) = h_i\}.$$

C_2 Identify, for conditional instance set $\mathcal{I}_p|h_i$, the **conditional image set** $\mathsf{Img}_p(u|h_i)$ for each node $u \in V(p)$:
$$\mathsf{Img}_p(u|h_i) = \{f_g(u) | g \in (\mathcal{I}_p|h_i)\}.$$

C_3 Compress $\mathcal{I}_p|h_i$ with the concatenation of the conditional images $\mathsf{Img}_p(u|h_i)$ over all nodes u in p:
$$\mathsf{code}(\mathcal{I}_p|h_i) = \{\mathsf{Img}_p(u|h_i)| \text{ for } u \in V(p)\}.$$

Finally, VCBC compress $\mathsf{code}(\mathcal{I}_p)$ by concatenation:
$$\mathsf{code}(\mathcal{I}_p) = \{\mathsf{code}(\mathcal{I}_p|h_i) | i \in [1, l]\}.$$

Example 5. In Figure 1, $V_c = \{u_1, u_2, u_3\}$ is a vertex cover of p. Let $h = (v_1, v_2, v_3)$ be a helve. Table 2 shows the conditional image sets of nodes under h. Step C_3 concatenates $\mathsf{code}(\mathcal{I}_p|h) = \{v_1\}\{v_2\}\{v_3\}\{v_4, v_5, v_6\}\{v_5, v_6, v_7\}\{v_4, v_5, \cdots, v_9\}$. The instance g which maps u_i to v_i, $i \in [1, 6]$, is coded.

The compression ratio can be calculated via Equation 1.

Example 6. For Figure 1, the conditional instance set $\mathcal{I}_p|h$ of $h = (v_1, v_2, v_3)$ has 24 instances under ord_p and is stored with $6 \times 24 = 144$ integers. $\mathsf{code}(\mathcal{I}_p|h)$ consists of 15 integers. The compression ratio $\rho(\mathcal{I}_p|h)$ is $144 \div 15 = 9.6$.

Remarks. Given an instance set $\mathcal{I}_p$, the compression can be done in a sorting time of $\mathcal{I}_p$, that is, in $\widetilde{O}(\frac{|\mathcal{I}_p|}{B})$ I/Os.

3.2 Decompression

As a reverse process of compression, decompression restores $\mathcal{I}_p$ from $\mathsf{code}(\mathcal{I}_p)$ by restoring, for each helve h_i, $i \in [1, l]$, in $\mathcal{H}(\mathcal{I}_p) = \{h_1, h_2, \ldots, h_l\}$, the conditional instance set $\mathcal{I}_p|h_i$ from $\mathsf{code}(\mathcal{I}_p|h_i)$, respectively, in 3 steps.

D_1 Load $\mathsf{code}(\mathcal{I}_p|h_i) = \{\mathsf{Img}_p(u|h_i) | u \in V(p)\}$ in memory.

D_2 Let S be the *Cartesian product* over the n_p image sets
$$S = \Pi_{u \in V(p)} \mathsf{Img}_p(u|h_i).$$

D_3 Let $\mathcal{I}'_p|h_i$ be the set of tuples in S without duplicated vertexes that are validated by ord_p.

Finally, report $\mathcal{I}'_p = \bigcup_{i \in [1,l]} (\mathcal{I}'_p|h_i)$.

THEOREM 1 (D-OPTIMAL). *The vertex-cover based compression is d-optimal. In other words, the decompression restores $\mathcal{I}_p$ in a streamed manner in $O(\frac{|\mathcal{I}_p|}{B})$ I/Os.*

PROOF. In step D_1, $\mathsf{code}(\mathcal{I}_p|h_i)$ consists of n_p conditional images sets. For each u in $V(p)$, image set $\mathsf{Img}_p(u|h_i) \subseteq V(d)$, thus $|\mathsf{Img}_p(u|h_i)| \leq |V(d)|$. Therefore, $\mathsf{code}(\mathcal{I}_p|h_i)$ does not exceed $\frac{M}{\sigma} \times \sigma = M$ words — fits into the memory. Besides, step D_2 and D_3 can be pipelined, that is, one can generate a tuple t of S then immediately test t via Step D_3. If t passes, stream t out right away. □

THEOREM 2 (LOSSLESS). *The vertex-cover based compression is lossless, that is, for a given V_c, $\mathcal{I}'_p = \mathcal{I}_p$.*

PROOF. We prove $\mathcal{I}_p = \mathcal{I}'_p$ in two directions.

1. $\mathcal{I}_p \subseteq \mathcal{I}'_p$. For any instance $g \in \mathcal{I}_p$, g will be recovered in the Cartesian product of S in step D_2 and pass the validation of ord_p in step D_3, and thus, $g \in \mathcal{I}'_p$.
2. $\mathcal{I}'_p \subseteq \mathcal{I}_p$: $\mathcal{I}'_p|h \subseteq \mathcal{I}_p|h$ for all helve h. Let $t = \{v_1, v_2, \ldots, v_{n_p}\}$ be a tuple in $\mathcal{I}'_p|h$. To prove $t \in \mathcal{I}_p|h$, it suffices to show that for any edge $(u_i, u_j) \in E(p)$, $(v_i, v_j) \in E(d)$ as t survived through step D_3. From the origin of t (D_2), there must be an instance $g_0 \in \mathcal{I}_p$ with helve h, and for each $u_i \notin V_c$, there must be an instance $g_i \in \mathcal{I}_p|h$ with $f_{g_i}(u_i) = v_i$. There is no edge between two nodes in $\overline{V_c}$. If u_i and u_i are both in V_c then $(v_i, v_j) \in E(g_0) \subseteq E(d)$; if u_i is in $\overline{V_c}$ and u_j is in V_c then $(v_i, v_j) \in E(g_i) \subseteq E(d)$. Thus, $\mathcal{I}'_p|h \subseteq \mathcal{I}_p|h$. □

Remarks. Theorems 1 and 2 provide an insight in the instance set $\mathcal{I}_p$, that is, when the images of a vertex cover V_c of the pattern graph p is fixed, all corresponding instances can be represented as a Cartesian product of the image sets of nodes in $V(p) \setminus V_c$. This insight guarantees that VCBC is a *d-optimal* compression for the instance set $\mathcal{I}_p$.

3.3 Compression Ratio

A Cartesian product over sets indicates a multiplication over set sizes. This reversely implies a high compression ratio. Below, we investigate the compression ratio of VCBC.

LEMMA 1. *The highest compression ratio of an instance set $\mathcal{I}_p$ of pattern p is given by a minimum vertex cover of p.*

PROOF. Let V_c and V'_c with $V_c \subseteq V'_c$ be two vertex covers of p. We show that the length of $\mathsf{code}(\mathcal{I}_p)$ under V_c is not longer than that under V'_c. Assume, without loss of generality, $V_c = \{v_i | i \in [1, x]\}$ and $V'_c = \{v_i | i \in [1, y]\}$ where $x \leq y$. Let h be a helve of V_c. $\mathcal{I}_p|h$ is a disjoint union of $\mathcal{I}_p|h'$ for $\forall h' \in pre(h)$. Here $pre(h)$ is the set of all the helves of V'_c with prefix equal to h. The Cartesian product (Step C_2) suggests that for each $u \in \overline{V_c}$ and $v \in V(d)$ with $v \in \mathsf{Img}_p(u|h)$ under V_c, there must be an $h' \in pre(h)$ such that $v \in \mathsf{Img}_p(u|h')$ under V'_c. Therefore, the length of $\mathsf{code}(\mathcal{I}_p|h)$ is no longer than the summation of the lengths of $\mathsf{code}(\mathcal{I}_p|h')$, for $\forall h' \in pre(h)$, which completes the proof. □

When the pattern graph p is a clique, any vertex cover of p has $\geq |V(p)| - 1$ vertexes. Therefore, we have Lemma 2.

LEMMA 2. *When the pattern graph is $\mathcal{C}_k$, the compression ratio of the vertex-cover based compression is $O(k)$.*

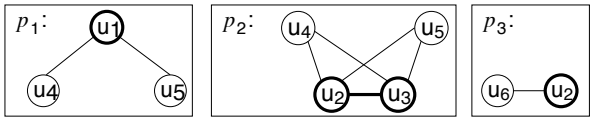


Figure 2: Three crystals. $p_1 : \mathcal{Q}_{1,2}$, $p_2 : \mathcal{Q}_{2,2}$, $p_3 : \mathcal{Q}_{1,1}$

Remarks. This subsection provides two findings on the compression ratio of VCBC, Lemma 1 and 2. However, it remains hard to quantify the compression ratio for general cases. Empirical results in Table 5 confirm that the Cartesian product of VCBC brings a significant compression ratio on real graphs. Moreover, the VCBC introduced in this section, together with general compression techniques such as *LZO*, *bzip2*, or *snappy*, provides an effective storage solution for subgraph matching, as shall be seen in Section 6.1.

4. CRYSTAL-BASED COMPUTATION

Based on VCBC, this section focuses on Problem 2. The aim is to find an approach to an efficient computation of $\mathsf{code}(\mathcal{I}_p)$ from the target graph d and the pattern graph p.

This section will introduce a *Crystal-Based computation Framework* (CBF). CBF computes $\mathsf{code}(\mathcal{I}_p)$ by computing $\mathsf{code}(\mathcal{I}_p|h_i)$, for each helve h_i in helves $\mathcal{H}(\mathcal{I}_p) = \{h_1, h_2, \cdots h_l\}$ with $l = |\mathcal{H}(\mathcal{I}_p)|$, respectively. Specifically, CBF

- Decompose p into a "core" and several basic constructs called "crystals". The "core" is used to generate the helves h_i of $\mathcal{I}_p$ while the crystals are used to generate the image sets for each helve.
- Compute the instances of the "core" by recursively calling CBF, since "core" is itself a pattern graph.
- Precompute the code of the "crystal"s' instance sets.
- Assemble $\mathsf{code}(\mathcal{I}_p|h_i)$ with instance h_i of the "core" and the corresponding codes of the crystals.

4.1 Framework Overview

CBF adopts a core-crystal decomposition to reduce the intermediate results. This enables a one-off assembly of the targeted $\mathsf{code}(\mathcal{I}_p)$. Start with three key components of CBF:

1. Crystals: a group of pattern graphs whose instance sets are precomputed and coded using VCBC.
2. Core-crystal decomposition: decompose the pattern graph into a "core" and crystals in a particular way.
3. One-off assembly: compute $\mathsf{code}(\mathcal{I}_p)$ by assembling the each instance of the "core" with the code of crystals.

Crystals. A crystal is a special pattern graph that is derived from cliques, defined as below.

DEFINITION 5 (CRYSTAL). *Let x and y be two positive integers. A crystal $\mathcal{Q}_{x,y}$ is a graph g with $x + y$ nodes such that there exists a set $V' \subseteq V(g)$ of x nodes and $\overline{V'} = V(g) \setminus V'$ with y nodes satisfying the following conditions.*

- *The induced subgraph $g(V')$ is a clique. $g(V')$ is called the core of the crystal, denoted as $\mathsf{core}(\mathcal{Q}_{x,y})$*
- *The induced subgraph $g(\overline{V'})$ is an independent set. The nodes in $\overline{V'}$ are called bud nodes. The edges incident to bud nodes are called bud edges.*

Table 3: Codes of conditional instance sets.

Conditional instance set	Helves on V_c			Image sets on $\overline{V_c}$	
	u_1	u_2	u_3	u_4 and u_5	u_6
$\mathcal{I}_{p_1}\|h_1$	v_1			v_3, v_4, v_5, v_6, v_7	
$\mathcal{I}_{p_2}\|h_2$		v_2	v_3	v_4, v_5, v_6, v_7	
$\mathcal{I}_{p_3}\|h_3$		v_2			$v_3, v_4, \cdots, v_9$
$\mathcal{I}_p\|h$	v_1	v_2	v_3	v_4, v_5, v_6, v_7	$v_3, v_4, \cdots, v_9$
$h_1 = (v_1)$,	$h_2 = (v_2, v_3)$,			$h_3 = (v_2)$,	$h = (v_1, v_2, v_3)$

- *Each bud node v is fully connected to the core, that is, $(u, v) \in E(g)$ for $\forall u \in V'$.*

LEMMA 3. *$\mathsf{core}(\mathcal{Q}_{x,y})$ is the induced subgraph of a vertex cover of $\mathcal{Q}_{x,y}$, that is, $\mathsf{core}(\mathcal{Q}_{x,y})$ covers all edges in $\mathcal{Q}_{x,y}$.*

Example 7. Figure 2 shows three crystals with cores marked in bold cycles. p_1 is a $\mathcal{Q}_{1,2}$ with core u_1. p_2 is a $\mathcal{Q}_{2,2}$ with core (u_2, u_3). u_4 and u_5 are bud nodes with bud edges $(u_2, u_4), (u_2, u_5)$, (u_3, u_4) and (u_3, u_5). p_3 is a crystal $\mathcal{Q}_{1,1}$.

A crystal $\mathcal{Q}_{x,y}$ is a pattern graph itself. As such, concepts subject to a pattern graph introduced in Section 3 apply: $\mathcal{Q}_{x,y}$ has its own instance set $\mathcal{I}_{\mathcal{Q}_{x,y}}$, its own helves $\mathcal{H}(\mathcal{I}_{\mathcal{Q}_{x,y}})$, its own conditional instance sets and conditional image sets.

The instance set of a crystal $\mathcal{Q}_{x,y}$ can be coded by VCBC with the instances of $\mathcal{C}_{x+1}$ — the clique of $x+1$ vertexes. Let $\mathcal{C}_x$ be a clique with nodes $v_1, v_2, \cdots, v_x$ in increasing identifiers. Let $\mathcal{Q}_{x,y}$ be a crystal with core nodes $u_1, u_2, \ldots, u_x$ and bud nodes $u_1, u_2, \ldots, u_y$. Define the partial order sets.

DEFINITION 6. *Let $\mathsf{ord}_{\mathcal{C}_x}$ include the orders of $v_1 \prec v_2 \prec \cdots \prec v_x$. Let $\mathsf{ord}_{\mathcal{Q}_{x,y}}$ include the following orders:*

$$u_1 \prec u_2 \prec \cdots \prec u_x, \text{ and } u_1 \prec u_j \prec \cdots \prec u_j.$$

LEMMA 4. *Given the instance set of clique $\mathcal{C}_{x+1}$, the code of the instance sets of crystals $\mathcal{Q}_{x,1}$ and $\mathcal{Q}_{x,y}$ can be obtained in a sorting time of $\mathcal{I}(\mathcal{C}_{x+1})$, if x and y are $O(1)$.*

PROOF. If symmetry breaking is not considered, $\mathcal{I}_{\mathcal{Q}_{x,1}} = \mathcal{I}_{\mathcal{C}_{x+1}}$ since $\mathcal{Q}_{x,1}$ is $\mathcal{C}_{x+1}$; besides, for each helve of $\mathcal{I}_{\mathcal{Q}_{x,y}}$, the image sets of y bud nodes are identical to the image set of the bud node of the same helve in $\mathcal{I}_{\mathcal{Q}_{x,1}}$. Next, we impose the orders defined in Definition 6 to the three pattern graphs and then compute their codes. $\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x,1}})$ is obtained in two steps in a sorting time of $\mathcal{I}_{\mathcal{C}_{x+1}}$:

- Generate $x + 1$ instances of $\mathcal{Q}_{x,1}$ from an instance g in $\mathcal{I}(\mathcal{C}_{x+1})$ by mapping the bud node of $\mathcal{Q}_{x,1}$ to each node of $\mathcal{C}_{x+1}$, respectively;
- Group the instances of $\mathcal{I}(\mathcal{Q}_{x,1})$ by their images on $\mathsf{core}(\mathcal{Q}_{x,1})$. The group of an image h of $\mathsf{core}(\mathcal{Q}_{x,1})$ and an image set of the bud node constitutes $\mathsf{code}(\mathcal{Q}_{x,1}|h)$.

$\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x,y}})$ is obtained by scanning $\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x,1}})$ y times. Specifically, let u be the bud node of $\mathcal{Q}_{x,1}$, and $u_1, u_2, \cdots, u_y$ be the bud nodes of $\mathcal{Q}_{x,y}$. Let h be a helve of $\mathcal{I}_{\mathcal{Q}_{x,1}}$. Assume that $\mathsf{Img}_{\mathcal{Q}_{x,1}}(u|h)$ has l nodes $\{v_1, v_2, \ldots, v_l\}$ where $v_1 < v_2 < \cdots < v_l$. If $l < y$ then h is not a helve of $\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x,y}})$; otherwise, $\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x,y}}|h)$ consists of image sets: $\mathsf{Img}_{\mathcal{Q}_{x,y}}(u_i|h) = \{v_i, v_{i+1}, \ldots, v_{l-y+i}\}$ for $i \in [1, y]$. □

Example 8. Table 3 shows the codes of the conditional instance sets of crystals in Figure 2. Note that p_1 is a crystal

$\mathcal{Q}_{1,2}$. When the core of p_1, u_1, sticks to the helve h_1 of node $v_1 \in V(d)$, all instances of $p_1|h_1$ are coded in two image sets $\mathsf{Img}_{p_1}(u_4|h_1) = \{v_3, \cdots v_6\}$, $\mathsf{Img}_{p_1}(u_5|h_1) = \{v_4, \cdots v_7\}$. These can be derived from the $\mathcal{C}_2$ instances on v_1 which is coded as a set of $\{v_3, \cdots, v_7\}$. Similarly, the code for p_2, a crystal of $\mathcal{Q}_{2,2}$, can be derived from the instance set of $\mathcal{C}_3$.

Core-Crystal Decomposition. A *core-crystal decomposition* of pattern graph p is a triple $\{V_c, \lambda, \mathcal{P}\}$ that satisfies:

DC_1 $V_c \subseteq V(p)$ is a vertex cover of p. The induced subgraph $p(V_c)$, is called the *core* of p, denoted as $\mathsf{core}(p)$.

DC_2 $\lambda \leq \sigma$ is an integer. $\mathcal{P}$ is a set $\{p_1, p_2, \ldots, p_\lambda\}$ of λ subgraphs of p, such that

(a) For each subgraph p_i, $i \in [1, \lambda]$:

i. p_i is a crystal $\mathcal{Q}_{x_i,y_i}$ for some integers x_i, y_i. Denote the core of $\mathcal{Q}_{x_i,y_i}$ as $\mathsf{core}(p_i)$.

ii. p_i intersects with $\mathsf{core}(p)$ *exclusively* on p_i's core, that is, $\mathsf{core}(p) \cap p_i = \mathsf{core}(p_i)$.

(b) The union of the subgraphs and the core is exactly p, that is, $(\bigcup_{i\in[1,\lambda]} p_i) \cup \mathsf{core}(p) = p$.

The above core-crystal decomposition conditions are designed for reducing the intermediate results, and facilitate an efficient one-off assembly. Astute readers may have noticed a redefinition of "core" on both a pattern graph and a crystal. Actually, Lemma 3 indicates their consistency.

LEMMA 5. *The induced subgraph $p(\overline{V_c})$ has no edge.*

Example 9. For the pattern p in Figure 1, $V_c = \{u_1, u_2, u_3\}$ is a vertex cover of p. The three subgraphs p_1, p_2 and p_3 of p in Figure 2 are crystals $\mathcal{Q}_{1,2}$, $\mathcal{Q}_{2,2}$ and $\mathcal{Q}_{1,1}$, with cores u_1, (u_2, u_3) and u_2 respectively. The triple $\{V_c, 3, \mathcal{P} = \{p_1, p_2, p_3\}\}$ is a valid core-crystal decomposition.

One-Off Assembly. For a core-crystal decomposition of $\{V_c, \lambda, \mathcal{P}\}$, the one-off assembly computes $\mathsf{code}(\mathcal{I}_p)$ with instances of core $p(V_c)$ and $\mathsf{code}(p_i)$ for each $p_i \in \mathcal{P}$, $i \in [1, \lambda]$.

The core-crystal decomposition is designed such that the core and the subgraphs are connected in a particular way. For example, $p(V_c)$ is a subgraph of p; the $\mathsf{core}(p_i)$ of p_i is a subgraph of both p_i and $\mathsf{core}(p)$ (recall the word "exclusive" in Condition ii, (a), DC_2). The subgraph relationships among the pattern graphs are mapped to their instances.

An instance g of the pattern graph p brings an instance-bijection which maps node $\forall u \in V(p)$ to node $g(u) \in V(d)$.

DEFINITION 7 (SUBGRAPH PROJECTION). *Let p' and p'' be two pattern graphs with $p' \subseteq p''$. Let g'' be an instance of p''. The projection of g'' on p', denoted as $g''(p')$, is defined as a graph with vertex set $\{g''(v)|v \in V(p')\}$ and edge set $\{(g''(u), g''(v))|(u, v) \in E(p')\}$. $g''(p')$ is a subgraph of g''.*

LEMMA 6. *$g''(p')$ is an instance of p'.*

PROOF. For any edge $(u, v) \in E(p')$, $(u, v) \in E(p'')$ since p' is a subgraph of p'', thus, $(g''(u), g''(v)) \in E(d)$ because g'' is an instance of p''. Therefore, g' is an instance of p'. □

Now we are ready to unveil the assembly of the instances.

DEFINITION 8. *Given a core-crystal decomposition, let h be an instance of $\mathsf{core}(p)$. For a subgraph p_i in $\mathcal{P}$, $h(\mathsf{core}(p_i))$ is the projection of h on $\mathsf{core}(p_i)$. For simplicity, $h(\mathsf{core}(p_i))$ is denoted as h_i and called the projection of h on p_i* [2].

Algorithm 1: Assembly

Input: An instance h of $\mathsf{core}(p)$ with, for each subgraph $p_i \in \mathcal{P}$, $i \in [1, \lambda]$, projections h_i on p_i and conditional $\mathsf{code}(\mathcal{I}_{p_i}|h_i)$.

Output: $\mathsf{code}(\mathcal{I}_p|h)$.

1 **for** *each* $u \in V(p)$ **do**

2 $\quad \mathsf{Img}'_p(u|h) \leftarrow \bigcap_{i\in[1,\lambda] \text{ with u in } p_i} \mathsf{Img}_{p_i}(u|h_i)$;

3 $\mathsf{code}'(\mathcal{I}_p|h) \leftarrow$ apply step C_3 on $\mathsf{Img}'_p(u|h)$, $u \in V(p)$;

4 $\mathsf{code}''(\mathcal{I}_p|h) \leftarrow$ Trim $\mathsf{code}'(\mathcal{I}_p|h)$: remove a node v in an image set $\mathsf{Img}'_p(u|h)$ if v cannot generate, via step D_2, any tuple that survives step D_3;

5 **return** $\mathsf{code}''(\mathcal{I}_p|h)$;

Algorithm 1 shows the one-off assembly under a decomposition $\{V_c, \lambda, \mathcal{P}\}$. For an instance h of $\mathsf{core}(p)$, the aim is to generate the image sets of $\mathsf{code}(\mathcal{I}_p|h)$. Obviously, it is not necessary to load the entire instance set of each subgraph $p_i \in \mathcal{P}$. All we need are conditional $\mathsf{code}(\mathcal{I}_{p_i}|h_i)$, for $\forall i \in [1, \lambda]$, where h_i is the projection of h on p_i (Definition 8). Line 2 obtains the tentative image set of $v \in \overline{V_c}$ in $\mathcal{I}_p|h$ by intersecting over corresponding image sets of $\mathcal{I}_{p_i}|h_i$, $i \in [1, \lambda]$. With these image sets, Line 3 simulates the compression step C_3 to generate a tentative $\mathsf{code}'(\mathcal{I}|h)$. Line 4 trims $\mathsf{code}'(\mathcal{I}|h)$ by simulating decompression step D_2 and D_3 to ensure that $\mathsf{code}''(\mathcal{I}|h)$ returned in Line 5 is compact.

Example 10. For the pattern p in Figure 1, let the decomposition have $V_c = \{v_1, v_2, v_3\}$ and $\mathcal{P} = \{p_1, p_2, p_3\}$ in Figure 2. The helve $h = (v_1, v_2, v_3)$ of pattern p is projected to $h(p_1) = v_1$, $h(p_2) = (v_2, v_3)$ and $h(p_3) = v_2$. The image sets of conditional instance sets of crystals and p are shown in Table 2. The image sets of $\mathcal{I}_p|h$ is obtained by intersecting the image sets column by column (Line 2, Algorithm 1).

Theorem 3 demonstrates the correctness of Algorithm 1.

THEOREM 3 (ONE-OFF ASSEMBLY). *For a given decomposition $\{V_c, \lambda, \mathcal{P}\}$ of p, Algorithm 1 assembles $\mathsf{code}(\mathcal{I}_p|h)$ for each helve h of $\mathcal{I}_p$ with the codes of subgraphs in $\mathcal{P}$.*

The proof of Theorem 3. To prove, we need to step into the technique of SimB [15]. Recall that SimB specifies a partial order set ord_p to avoid duplicated enumeration (Section 2). Actually, SimB identifies ord_p from the equivalences among nodes in V_p: two nodes are equivalent if there is an automorphism of p that maps one node to the other. The equivalence relationship is transitive, which draws equivalence classes in $V(p)$. SimB determines ord_p in rounds. Initially, $\mathsf{ord}_p = \emptyset$. Each round, SimB identifies an equivalence class—a set of nodes $V' \subseteq V(p)$ that are mutually equivalent under ord_p. SimB breaks the class by imposing partial orders on V': pick a node $v \in V'$ as the *anchor node* and then add (v, v') to ord_p for every $v' \in V' \setminus \{v\}$. SimB repeats the rounds until no equivalence class exists.

CBF, though has a single pattern graph p, decomposes p into a core and subgraphs in $\mathcal{P}$; each of which is a pattern

[2] Safe abuse since p_i intersects $\mathsf{core}(p)$ exclusively on $\mathsf{core}(p_i)$.

graph itself. The problem is to consist the orders in CBF for all decomposed pattern graphs. This can be achieved by levering SimB's freedom in choosing the anchor node for an equivalence class. Given a pattern p and its decomposition $\{V_c, \lambda, \mathcal{P}\}$, CBF imposes *extra rules* to SimB in anchor node selection in determining ord_p, $\mathsf{ord}_{\mathsf{core}(p)}$, and the partial orders of subgraphs in $\mathcal{P}$, crystals and cliques in preprocessing.

Specifically, CBF identifies nodes in $V(p)$ with integers from 1 to n_p such that the identifiers of V_c nodes are smaller than that of non-V_c nodes. Then compute ord_p with SimB: in each round, the anchor node of an equivalence class is designated to the node with the smallest identifier. For any (u, v), or equivalently, $u \prec v$, in ord_p, the identifier of u is smaller than v. Let $\mathsf{ord}_{\mathsf{core}(p)} = \mathsf{ord}_p \cap \{u \prec v | \forall u, v \in V_c\}$.

LEMMA 7. *Given a pattern p, its decomposition $\{V_c, \lambda, \mathcal{P}\}$, the partial order sets for p, $\mathsf{core}(p)$, crystals, and cliques are defined by* CBF *as above, respectively. Let g be an instance of p under ord_p. 1) The projection $g(p_i)$ of g on $p_i \in \mathcal{P}$ is an instance of p_i under ord_{p_i}, for $i \in [1, \lambda]$. 2) The projection of g on $\mathsf{core}(p)$ is an instance of $\mathsf{core}(p)$ under $\mathsf{ord}_{\mathsf{core}(p)}$. 3) g can be restored from $\mathsf{code}'(\mathcal{I}_p|h)$ in Line 3, Algorithm 1.*

PROOF. 1) p_i is a crystal $\mathcal{Q}_{x_i,y_i}$. ord_p indicates that for a core node u and a bud node v of p_i, $g(u) < g(v)$. Note that, there is a hidden mapping from core (bud, resp.) nodes in p_i to the core (bud, resp.) nodes crystal $\mathcal{Q}_{x_i,y_i}$. Let this mapping to be instance dependent, that is, map nodes u in $\mathsf{core}(p_i)$ to $\mathsf{core}(\mathcal{Q}_{x_i,y_i})$ in ascending order of $g(u)$; and do the same for bud nodes. In this way, $g(p_i)$ follows $\mathsf{ord}_{\mathcal{Q}_{x,y}}$ and thus is in $\mathcal{I}(p_i)$. 2) $g(\mathsf{core}(p))$ is an instance of $\mathsf{core}(p)$ since $\mathsf{ord}_{\mathsf{core}(p)}$ is a subset of ord_p. 3) g can be restored from $\mathsf{code}'(\mathcal{I}_p|h)$ since for each $u \in V(p)$, $g(u)$ is in the image set of u over all subgraphs that contains u, and is thus in $\mathsf{Img}'_p(u|h) = \bigcap_{i \in [1,\lambda] \text{ with u in } p_i} \mathsf{Img}_{p_i}(u|h_i)$ (Line 2). □

LEMMA 8. *In Algorithm 1, $\mathsf{code}''(\mathcal{I}_p|h)$ reported in Line 5 is exactly $\mathsf{code}(\mathcal{I}_p|h)$.*

PROOF. We first show that any tuple t decompressed from $\mathsf{code}'(\mathcal{I}_p|h)$ via step D_2 and D_3 is an instance of p.

Recall that t was decompressed from the Cartesian product over the image sets of $\mathsf{Img}'(u|h)$ (step D_2), namely, every node in t is an image of a node in p. Denote by $t(v)$ the image of $v \in V(p)$ in t. Mapping t is a bijection and follows ord_p since t had survived through decompression step D_3.

To show that t is isomorphic to p, that is, for every edge $(u, v) \in E(p)$, $(t(u), t(v)) \in E(d)$, consider the intersection in Line 2. If $u, v \in V_c$, then $t(u)$ and $t(v)$ are specified by h. Since h is an instance of $\mathsf{core}(p)$, $(t(u), t(v)) \in E(d)$. If $u \in V_c$ and $v \in \overline{V_c}$, due to condition 2(b), there exists p_i with $(u, v) \in E(p_i)$, thus $(t(u), t(v)) \in E(d)$. Lemma 5 guarantees that there is no edge between two nodes in $\overline{V_c}$. Therefore, t is isomorphic to p and is thus an instance of p.

For any instance g in $\mathcal{I}_p|h$, g is in the decompression of $\mathsf{code}'(\mathcal{I}_p|h)$ (Lemma 7). Note that removing any node in $\mathsf{code}''(\mathcal{I}_p|h)$ will lead to a different decompression set (Line 4), violating the fact that the decompression sets of $\mathsf{code}'(\mathcal{I}_p|h)$, $\mathsf{code}''(\mathcal{I}_p|h)$ and $\mathsf{code}(\mathcal{I}_p|h)$ are identical. Therefore, $\mathsf{code}''(\mathcal{I}_p|h)$ is exactly $\mathsf{code}(\mathcal{I}_p|h)$. □

This subsection has explained the essence of the framework, that is, decompose the pattern graph p into a core and λ crystals, compute their instances/codes respectively, and assemble their instances back to the code of $\mathcal{I}_p$ in a one-off manner. Section 4.2 to 4.5 describe each component in details under external memory model. Section 4.2 shows the preprocessing step which codes the instances of crystals. Section 4.3 shows the computation of $\mathsf{core}(p)$ instances. Section 4.4 elaborates the one-off assembly (Algorithm 1). Section 4.5 shows how to decompose the pattern graph. Section 4.6 parallelizes the one-off assembly.

4.2 Preprocessing: Clique Listing

Based on Lemma 4, to code the instance set of a crystal of $\mathcal{Q}_{x,y}$, it suffices to list the instances of clique $\mathcal{C}_{x+1}$. This can be trivially done for $\mathcal{C}_1$ and $\mathcal{C}_2$ whose instance sets are the vertex and edge sets, respectively, of the target graph. The instances of a clique of $\mathcal{C}_k$ can be either computed from scratch using the hypercube approach [1] or inductively by resorting to Loomis-Whitney Join (LW-Join) [24]. The worst-case complexity of these approaches conforms when the target graph is a clique: the complexity for computing $\mathcal{I}_{\mathcal{C}_k}$ is dominated by the output cost $\Theta(\frac{1}{B}|E(d)|^{k/2})$.

This preprocessing step aims at computing, for a parameter k_0, the instance sets of all cliques $\mathcal{C}_k$ with k from 1 to a certain k_0. LW-join suits sparse graphs whose total number of instances of clique $\mathcal{C}_k$ is far less than $|E(d)|^{k/2}$. LW-join scan $\mathcal{I}_{\mathcal{C}_k}$ for $(\frac{|\mathcal{I}_{\mathcal{C}_k}|}{M})^{\frac{1}{k}}$ times to obtain $\mathcal{I}_{\mathcal{C}_{k+1}}$ (Lemma 10).

DEFINITION 9 (LOOMIS-WHITNEY JOIN(LW-JOIN)[24]). *Denote by $\mathcal{A}$ attributes $\{a_1, a_2, \cdots, a_{k+1}\}$. Loomis-Whitney Join on $\mathcal{A}$ is a join of $k+1$ relations, $R_1, \ldots, R_{k+1}$, where each relation R_i has a schema of $\mathcal{A} \setminus \{a_i\}$, for $i \in [1, k+1]$.*

For example, when $k = 2$, the schema of $k+1 = 3$ relations are $R_1(a_2, a_3)$, $R_2(a_1, a_3)$, and $R_3(a_1, a_2)$.

LEMMA 9. *Given the instance set of clique $\mathcal{C}_k$, the problem of computing the instance set of $\mathcal{C}_{k+1}$ is a LW-Join.*

PROOF. Let relation R_i, $i \in [1, k+1]$, be the instance set $\mathcal{I}_{\mathcal{C}_k}$. Compute the instance set $\mathcal{I}_{\mathcal{C}_{k+1}}$ via the LW-join $\bowtie_{i \in [1,k+1]} R_i$. □

The algorithm and analysis in [16] show the overall complexity (Lemma 10) where $\Theta(\frac{1}{B}|\mathcal{I}_{\mathcal{C}_{k+1}}|)$ is the output cost.

LEMMA 10 ([16]). *The worst-case I/O complexity for computing the instance set of clique $\mathcal{C}_{k+1}$ from that of $\mathcal{C}_k$ is*

$$\widetilde{\Theta}\left(\frac{1}{B}|\mathcal{I}_{\mathcal{C}_k}|\left(\frac{|\mathcal{I}_{\mathcal{C}_k}|}{M}\right)^{\frac{1}{k}} + \frac{1}{B}|\mathcal{I}_{\mathcal{C}_{k+1}}|\right).$$

4.3 Core Instance Computation

The core of p is a pattern graph itself. CBF can compute the instances of $\mathsf{core}(p)$ recursively until p is a crystal. Lemma 11 shows that such a recursion terminates in constant rounds if a minimum vertex cover is chosen by each core-crystal decomposition. Specifically, if each recursion reduces the pattern size by at least 2 then the total number of recursions is at most $|V(p)|/2 \leq \sigma/2$, a constant.

LEMMA 11. *Let V_c be a minimum vertex cover of p. If p is not a clique, then $|V_c| \leq |V(p)| - 2$.*

PROOF. p is not a clique, there is an edge $(u, v) \notin E(p)$ with $u, v \in V(p)$, then $V(p) \setminus \{u, v\}$ is a vertex cover of p. □

Remarks. When $\mathsf{core}(p)$ has multiple connect components, the instance set of each connected components are computed respectively. CBF combines the instances from different connected components with the one-off assembly, as shall be introduced in the next subsection.

4.4 One-off Assembly

We now adapt Algorithm 1 to EM model. Recall that given a core-crystal decomposition $\{V_c, \lambda, \mathcal{P}\}$ with $\mathcal{P} = \{p_1, p_2, \cdots, p_\lambda\}$, each p_i is crystal $\mathcal{Q}_{x_i,y_i}$ for $i \in [1, \lambda]$. Algorithm 1 assembles $\mathsf{code}(\mathcal{I}_p)$. Specifically, an instance h of the $\mathsf{core}(p)$ is recursively computed (Section 4.3); $\mathsf{code}(\mathcal{I}_{p_i})$ is pre-computed for each $p_i \in \mathcal{P}$ (Section 4.2). With the projection h_i of h on each p_i (Definition 8), Algorithm 1 assembles $\mathsf{code}(\mathcal{I}_p|h)$ with $\mathsf{code}(\mathcal{I}_{p_i}|h_i)$ for all $p_i \in \mathcal{P}$.

The performance of Algorithm 1 under EM model is largely affected by *fractional disk accesses* — even if $\mathcal{I}_{p_i}|h_i$ has only one instance, Line 2 has to pay one I/O for h_i. In other words, each helve in $\mathcal{H}(\mathcal{I}_p)$ consumes at least λ I/Os, rendering at least $\lambda|\mathcal{I}_p|$ I/Os in the worst-case. Alike the hash-joins in external memory, we resort to hash functions.

4.4.1 Hash-Assembly

The aim of a hash-assembly is to partition the instances of the core and each subgraph in $\mathcal{P}$ into buckets, a bucket can be held in main memory such that the one-off assembly can be performed by enumerating the combinations of buckets. In this way, fractional disk accesses can be avoided.

Hash function on clique instances. Lemma 4 suggests that a helve h of $\mathcal{I}_{\mathcal{Q}_{x+1,y}}$ is a helve of $\mathcal{I}_{\mathcal{Q}_{x+1,1}}$ and an instance of clique $\mathcal{C}_x$. We define, for h, a weight $w(h)$, as the total number of instances of $\mathcal{Q}_{x+1,1}$ under helve h. Note that $w(h)$ is also the size of the only image set of $\mathsf{code}(\mathcal{I}_{\mathcal{Q}_{x+1,1}}|h)$.

Example 11. Table 3 shows the codes of the three crystals p_1, p_2 and p_3 in Figure 2, respectively. For p_1, $h_1 = v_1$ is an instance of $\mathsf{core}(p_1)$, the weight $w(h_1)$ is therefore $5 = |\{v_3, v_4, v_5, v_6, v_7\}|$ — the size of the image set of the bud node of p_1. Similarly, for crystal p_2, the weight $w(h_2)$ with $h_2 = (v_2, v_3)$ is 4; for p_3, the weight of helve v_2 is 7.

LEMMA 12. *Consider clique $\mathcal{C}_{x-1}$ and its instances $\mathcal{I}_{\mathcal{C}_{x-1}}$. There exists a mapping function ξ_x with $c_x = O(|\mathcal{I}_{\mathcal{C}_x}|/M)$ $\xi_x : \mathcal{I}_{\mathcal{C}_{x-1}} \mapsto \{1, 2, \cdots, c_x\}$, such that for each $j \in [1, c_x]$, $\Sigma_{h \text{ with } \xi(h)=j} w(h) \leq (\eta/\sigma)M$.*

PROOF. Let $L = \frac{\eta}{\sigma}M$. The mapping function can be obtained with a greedy algorithm. Consider a conceptual sequence of buckets numbered $1, 2, \cdots$ with capacity L initially labeled empty. Scan instances of $\mathcal{C}_{x-1}$ in non-increasing order of their weights. For each instance h, find the largest non-empty bucket, or the first bucket if all buckets are empty. If this bucket can hold the current instance without exceeding the capacity limit, add the instance to the bucket; otherwise, label the bucket as full and insert the instance to the next bucket. After scanning all the instances of $\mathcal{C}_x$, we denote the total number of used bucket as c. To bound c, we notice that each used bucket except the last one has a weight in $[L/2, L]$. Thus, $c \leq \frac{2x|\mathcal{I}_{\mathcal{C}_x}|}{L} + 1 = O(\frac{|\mathcal{I}_{\mathcal{C}_x}|}{L})$. □

Hash function on core instances. For each crystal $p_i = \mathcal{Q}_{x_i,y_i} \in \mathcal{P}$, its helve h_i is an instance of clique $\mathcal{C}_{x_i-1}$. Therefore, hash function ξ_{x_i} defined above can map h_i to a number in $[1, c_{x_i}]$. For an instance h of the $\mathsf{core}(p)$, recall that h determines its projections h_i on each subgraph p_i (Definition 8). The hash function over the core instances is derived:

$$\xi(h) = (\xi_{x_1}(h_1), \xi_{x_2}(h_2), \ldots, \xi_{x_\lambda}(h_\lambda)).$$

Hash-Assembly. Raise an *assembly-job* for each vector

$$vec = (s_1, s_2, \ldots, s_\lambda) \in [1, c_{x_1}] \times [1, c_{x_2}] \times \cdots \times [1, c_{x_\lambda}].$$

An assembly-job of vec loads, for each $i \in [1, \lambda]$ and each instance h_i of $\mathcal{C}_{x_i-1}$ with $\xi_{x_i}(h_i) = s_i$, the $\mathsf{code}(\mathcal{I}(\mathcal{Q}_{x_i,1}|h_i))$ in main memory in the entirety. This is doable since all these codes fit in the main memory, as suggested by Lemma 12. After that, scan over all the core instances h with $\xi(h) = vec$ and run Algorithm 1 for each of such instances.

LEMMA 13. *A hash-assembly has $O(\Pi_{i\in[1,\lambda]}(|\mathcal{I}_{\mathcal{C}_{x_i+1}}|/M))$ total number of assembly jobs. Each core instance is scanned exactly once in exactly one assembly-job. Each job entails $O(\frac{M}{B})$ I/Os in loading the clique instances into the memory.*

THEOREM 4. *The enumeration cost of the hash assembly of the instance set $\mathcal{I}_p$:*

$$\widetilde{O}\left(\frac{|\mathcal{I}_{\mathsf{core}(p)}|}{B} + \frac{M}{B} \times \Pi_{i\in[1,\lambda]}\left(\frac{|\mathcal{I}_{\mathcal{C}_{x_i}}|}{M}\right)\right) I/Os.$$

4.5 Core-Crystal Decomposition

A core-crystal decomposition $\{V_c, \lambda, \mathcal{P}\}$ supports efficient one-off assembly by restraining itself. Now we are ready to show how these constraints can be satisfied when only the pattern graph p is available. The first question is whether there exists a core-crystal decomposition. We provide a positive answer with the initial decomposition defined below.

DEFINITION 10 (INITIAL DECOMPOSITION). *Let V_c be a vertex cover of p. Let $\lambda = |V_c|$. Denote V_c as $\{u_1, u_2, \cdots, u_\lambda\}$. Create graph p_i for each node $u_i \in V_c$ with $E(p_i) = \{(u_i, v) \in E(p) | v \notin V_c\}$. Let $\mathcal{P} = \{p_1, p_2, \cdots, p_\lambda\}$. $\{V_c, \lambda, \mathcal{P}\}$ is a core-crystal decomposition: p_i is a crystal whose core is u_i.*

After we found the first core-crystal decomposition, the next question is how to optimize a core-crystal decomposition. This goal can be achieved by first setting the objective of the optimization, and then enumerate core-crystal decompositions to optimize the objective.

4.5.1 Optimization Objective

Firstly, V_c should be a minimum vertex cover. Since the output cost $\Theta(\frac{1}{B}|\mathsf{code}(\mathcal{I}_p)|)$ is dependent only on the compression ratio $\rho(\mathcal{I}_p)$. $\rho(\mathcal{I}_p)$ is determined by V_c: Lemma 1.

Secondly, the "best" decomposition is expecting a connected core $p(V_c)$: the complexity for computing the core instances affects the recursion efficiency, which is decided by V_c as well. If $p(V_c)$ is not connected, $p(V_c)$ is the Cartesian product over the instance set of $p(V_c)$'s connected components. Lemma 14 indicates that when $p(V_c)$ has > 1 connected components, few instances of $p(V_c)$ are helves of p.

LEMMA 14. *Let p be a connected graph. Let V' be a vertex cover of p with two connected components cc_1 and cc_2 in $p(V')$. There exist two nodes $u \in V(cc_1)$ and $v \in V(cc_2)$ with u and v that are two-hop away in p.*

PROOF. Let $u' \in V(cc_1)$ and $v' \in V(cc_2)$ be the node pair with the shortest distance in p among all such node pairs. If the distance from u' to v' is more than 2, then there must be an edge on the shortest path between u' and v', uncovered by V', then V' is not a vertex cover of p, contradiction. □

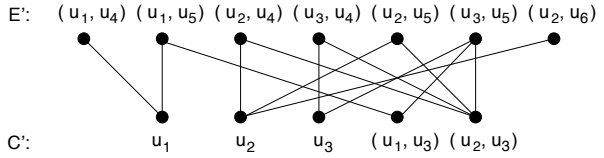


Figure 3: The Cover Graph cover-graph$((u_1, u_2, u_3))$.

Finally, the complexity of one-off assembly (Theorem 4) instructs the "best" decomposition to minimize the function

$$f(\mathcal{P}) = \frac{M}{B} \times \Pi_{i\in[1,\lambda]} \left(\frac{|\mathcal{I}_{\mathcal{C}_{x_i}}|}{M}\right). \quad (2)$$

If statistical information on the total number of cliques is available, one can evaluate the function for each possible core-crystal decomposition. Otherwise, heuristics apply: λ should be minimized, then each x_i should be minimized.

As a conclusion, core-crystal decomposition should select

1. a *minimum* vertex cover V_c of p,
2. $p(V_c)$ with the fewest connected components, and
3. • $\mathcal{P}$ that minimizes $f(\mathcal{P})$ in Equation 2, if the statistical information of $|\mathcal{I}_{\mathcal{C}_i}|$, $i \in [1, \lambda]$, is given;
   • or $\mathcal{P}$ that minimizes λ and then minimizes x_i, for each $i \in \lambda$, if such information is not available.

With the three **objectives** above ready, it remains to enumerate all possible core-crystal decompositions.

4.5.2 Decomposition Enumeration

It is not hard to image how to optimize Objectives 1 and 2 by enumerating all possible minimum vertex covers V_c in $O(2^m)$ time. This subsection shows how to optimize, given a vertex cover V_c, Objective 3 by enumerating crystals of $\mathcal{P}$ that satisfy all constraints of a core-crystal composition.

An invariant largely reduces the search space: Equation 2 is independent with the parameter of "y_i" of each crystal in $\mathcal{P}$. Note that all bud edges should cover all edges between V_c and $\overline{V_c}$. Therefore, when the core(p_i) of a crystal p_i is fixed, all possible bud nodes in $\overline{V_c}$ should be added to p_i to minimize λ. Moreover, the cores of the subgraph are cliques in $p(V_c)$, so it suffices to enumerate all combinations of cliques in $p(V_c)$ and then check, for each combination, if the cliques can "cover" all edges between V_c and $\overline{V_c}$.

To formally describe the above problem, denote by C' the set of all cliques in $p(V_c)$; denote by E' the set of edges in $E(p)$ between V_c and $\overline{V_c}$. We construct a bipartite graph, denoted as cover-graph(V_c), over E' and C'. Specifically, the vertex set of cover-graph(V_c) is the union $E' \cup C'$; and an edge between $g \in C'$ and $e(v, u) \in E'$ with $u \in V_c$ is linked if v is fully connected to C', that is, $\{v\} \times V(g) \subseteq E(p)$.

Example 12. In Figure 1, if $V_c = \{u_1, u_2, u_3\}$, E' includes all edges in $E(p)$ except (u_1, u_3) and (u_2, u_3), whereas C' includes three nodes u_1, u_2, u_3 and two edges (u_1, u_3) and (u_2, u_3). Figure 3 shows the cover graph cover-graph(V_c).

The problem of optimizing $\mathcal{P}$ is then defined as below.

DEFINITION 11 (OPTIMIZE-$\mathcal{P}$). *Given a vertex cover V_c of p, enumerate, all subsets of C' that cover all items in E' in* cover-graph(V_c)*, to optimize Objective 3.*

This is a cover problem on a bipartite graph.

THEOREM 5. *Optimize-$\mathcal{P}$ can be solved with an algorithm in $O(2^{n_p} m_p (2^{m_p} + 2^{n_p}))$ time with space $O(2^{m_p})$.*

PROOF. Objective 3 has two cases: Case 1 is provided with statistical information while Case 2 uses heuristics. *Case 1* has a function $f(\mathcal{P})$ to evaluate cost:

$$\log(f(\mathcal{P})) = \log(M/B) + \Sigma_{i\in[1,\lambda]} \log(|\mathcal{I}_{\mathcal{C}_{x_i}}|/M),$$

is decided by the summation of $\log(|\mathcal{I}_{\mathcal{C}_{x_i}}|/M)$ over the selected cliques in C'. The problem can then be resolved with memorized search — a dynamic programing algorithm. Use an array DP of size $2^{|E'|}$ to denote, for each subset E'' of E', the subset C'' of C' that covers E'' with minimum cost—the summation of $\log(|\mathcal{I}_{\mathcal{C}_{x_i}}|/M)$ over $\mathcal{C}_{x_i}$ selected by C''. DP$[E'']$ does not have to store C''. C'' can be restored by tracing from DP$[E'']$ back to the state where the minimum cost came from. It suffices to progressively add cliques to C', each takes $O(m_p 2^{|E'|})$ time to update each state in DP, until C' includes $O(2^{n_p})$ cliques in V_c. *Case 2.* To find the $\mathcal{P}$ with the minimum λ, we start our search with $\lambda = |V_c|$ provided by the initial decomposition (Definition 10). It remains to enumerate $O(|C'|^{|V_c|}) = O(2^{2n_p})$ combinations of elements in C' with no more than $|V_c|$ elements. This can be implemented as a depth-first-search, with the coverage status over E' maintained along the recursion. Each combination in C' consumes $O(m_p)$ time to update the status. □

This section concludes the introduction to CBF in external memory. Next section extends CBF to parallel platforms.

4.6 Parallelization

Recall that in Section 4.4, a hash-assembly method is used to chop the one-off assembly into $par = O(\Pi_{i\in[1,\lambda]}(|\mathcal{I}_{\mathcal{C}_{x_i}}|/M))$ assembly-jobs, where each job fits in the memory of $O(M)$.

This partition naturally fits parallel platforms: the jobs are mutually *independent*, that is, they don't communicate at all. Let M be a number smaller than the memory size of a slave machine, the parallelism is determined by the total number of assembly-jobs. The communication complexity of the one-off assembly conforms to Theorem 4:

$$\widetilde{O}\left(|\mathcal{I}_{\text{core}(p)}| + M \times \Pi_{i\in[1,\lambda]}\left(\frac{|\mathcal{I}_{\mathcal{C}_{x_i}}|}{M}\right)\right).$$

Besides, the loading process, since each bucket is stored consecutively, can be completed in λ network reads on the distributed file system. No shuffle—the most expensive operation on a parallel platform—is required. The practical performance, therefore, could be superior than the approaches with the same communication complexity that relies on shuffling, as observed in a recent paper [27]. The independence between tasks enables a *near linear* speedup with the parallelism, as will be confirmed in our experiments.

This section has introduced CBF, a framework that computes, for a subgraph matching, the instance set $\mathcal{I}_p$, in a compressed form, directly from the pattern graph and target graph. CBF can be easily deployed on parallel platforms.

5. RELATED WORK

This section first discusses output crisis of subgraph matching computation, then overviews subgraph matching computation and finally surveys other relevant research.

Compression. This is the first attempt, in the literature, on resolving the *output crisis* of subgraph matching using

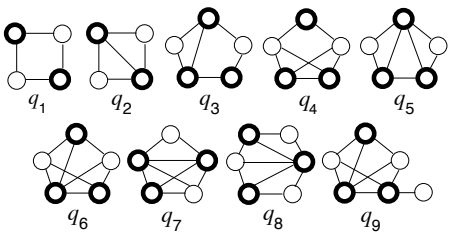


Figure 4: Query patterns

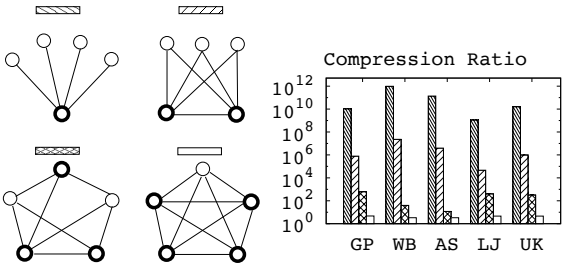


Figure 5: Compression ratio - Freedom

output compression. In subgraph matching, output compression is vertical to input compression [14, 31, 28]. Input compression techniques leverage symmetries in the target graph nodes such that the computation on one node alleviates the computation on other nodes. Other existing research either blindly export the instance set $\mathcal{I}_p$ entirely to the disk [1, 20, 2, 29, 19], or choose not to output at all, see the seminal work of [26]. The former ones, unavoidably, entail $\Omega(\frac{|\mathcal{I}_p|}{B})$ I/Os for export; whereas the latter ones, suffer a re-computation cost of $\mathcal{I}_p$ upon every following request.

Computation. In main memory, subgraph matching computation has been investigated extensively (see seminar work [30, 8]). As an instance of multi-join — subgraph matching is a join over m_p binary-relations on n_p attributes where each relation is materialized with $E(d)$, the upper and lower bounds has been matched [24]. Inspired by this, in external memory, special patterns such as wedges or triangles have been throughly investigated, see [26, 16] as an entrance.

Subgraph matching on parallel platform can be categorized on how they deal with intermediate results. DFS-style approaches [1, 2, 19, 27] avoids intermediate results by using one-round computation while BFS-style approaches, see recent works [29, 20, 21], shuffle a huge number of intermediate results. BFS-style approaches are expensive for its size of the intermediate results, which could be larger than $|\mathcal{I}_p|$.The latest BFS-style approach [21] uses cliques as a unit of each round of expansion; the defect is still shuffling of the intermediate results. DFS-style approach [1] avoids the intermediate results by replicating the target graph; however, in comparison of a BFS-style approach, the performance of a DFS-style approach [1] could be even worse, as reported in [20]. DFS-style parallelism can be deployed in a single machine [19]. An empirical study [27] on triangle enumeration shows the power of *network read* on DFS-style approaches.

Other Related Works. Subgraph counting reports the size of $|\mathcal{I}_p|$ instead of listing $\mathcal{I}_p$. The computation of an approximate count can be very efficient [4]. Triangle counting is an active topic [13] even on dynamic graphs [7].

On labeled data and pattern graphs, subgraph matching computation allows larger pattern and larger target graphs, see a recent work [6] as an entrance. In the worst case, that is, all nodes are marked with the same label, the problem deteriorates to the unlabeled subgraph matching.

6. EXPERIMENTS

This section evaluates our proposed approaches, including the compression ratios of VCBC and the performance of CBF.

Environment. Experiments were deployed on an instance of MapReduce, Apache Hadoop version 2.6.0, upon a cluster with 1 master node and 20 slave nodes. Each node was equipped with 12 cores each of 2.6GHz, and 4 hard drives each of 2 terabytes. The underlying *hadoop distributed file system* (HDFS) had available space of 125 terabytes with a default replication factor of 3. The system was configured to assign each core with one mapper and one reducer and 4 gigabyte memory space unless otherwise specified.

Approaches. Four approaches were examined.

- `Crystal` and `Crystal-1`: our approach;
- `DualSim`[19]: the state-of-the-art DFS-style solution;
- `TwigTwin`[20]: the state-of-the-art BFS-style solution;
- `SEED`[21]: the state-of-the-art BFS-style solution.

The core-crystal decomposition (Section 4.5.2) was implemented as a main-memory algorithm in C++ on one of our slave machines. We assumed no statistical information on target graphs in the decomposition optimization. `Crystal` is a parallel implementation of CBF in Java 1.6 under MapReduce. `Crystal-1` is the single-machine version of `Crystal`. Two groups of comparisons were designed:

- `Crystal-1` against `DualSim` as single-machine parallelisms on one slave machines,
- `Crystal` against `TwigTwin`, and `SEED` as multi-machine parallelisms on the cluster described above.

Pattern Graphs. Experiments used graphs in Figure 4 as pattern graphs, q_1 to q_7 have 4-5 nodes, q_8 (from [21]) and q_9 (from our running example) have 6 nodes. The minimum vertex cover computed by the core-crystal decomposition is marked with bold cycles for each pattern graph.

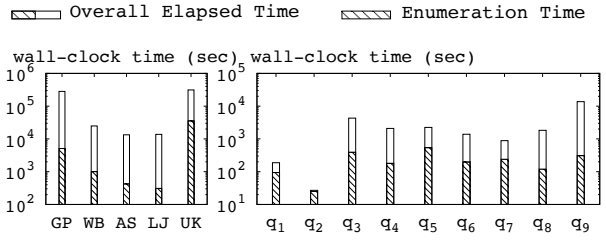
Target Graphs. Experiments used graphs in Table 4 as target graphs. `UK` was downloaded from `http://law.di.unimi.it/datasets.php` while other datasets were downloaded from `https://snap.stanford.edu/data/`. The statistics of the target graphs d include graph size, average degree (avg-deg) and degeneracy. $\textit{avg-deg}(d) = \frac{|2\times E(d)|}{|V(d)|}$, and degeneracy, the smallest integer k such that any subgraph of d has a node with degree $\leq k$, measure the sparseness of d. Below, a "testcase" or simply "case" means a pair of a pattern graph in Figure 4 and a target graph in Table 4.

Metrics. The cost of an algorithm on a testcase is evaluated in the elapsed time. The *enumeration cost* is separated from the *output cost*, in the *overall cost* (Section 2.4).

Guideline. Section 6.1 exhibits the compression ratio of vertex-based compression. Section 6.2 evaluates the performance CBF. Section 6.3 compares CBF with other solutions.

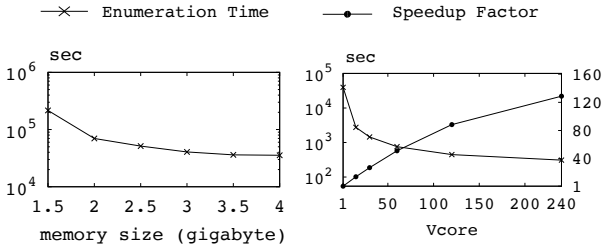
Table 4: Datasets

dataset	$\|V(d)\|$ $\times 10^6$	$\|E(d)\|$ $\times 10^6$	avg-deg	degen-eracy	size(d) in MB
ego-Gplus(GP)	0.1	12.2	244	1504	390
web-BerkStan(WB)	0.7	6.6	19	402	211
as-Skitter(AS)	1.7	11.1	13	222	355
soc-LiveJournal(LJ)	4.8	42.9	18	746	1373
uk-2002(UK)	18.5	298.1	32	1886	9539



(a) Vary d. Let p be q_9. **(b)** Vary p. Let d be LJ.

Figure 6: Costs of Crystal.



(a) vary memory size **(b)** vary parallelism

Figure 7: Enumeration cost of Crystal

6.1 Compression Ratio on Real Datasets

Sensitivity Test. We find that the compression ratio is closely related to the freedom of the vertex cover V_c of the compression. Specifically, let $\overline{V_c} = V(p) \setminus V_c$ be V_c's complement. The freedom of V_c is $|\overline{V_c}|$. If V_c is a minimum vertex cover of p, then $|\overline{V_c}|$ is also called the freedom of p.

Figure 5 shows the compression ratios of $\mathcal{I}_p$ when the pattern graph has different degrees of freedom. The 4 pattern graphs to the left have 5 nodes each and a minimum vertex cover marked in bold cycles. The compression ratios to the right have shown an obvious and consistent trend on all of the 5 target graphs in Table 4, that is, the pattern graph with a higher freedom enjoys a higher compression ratio.

Compression Ratio Test. Table 5 shows the compression ratio of $\mathcal{I}_p$ over all testcases.

$\rho(\mathcal{I}_p)$ is significant: in 98% of the testcases in Table 5, the compression ratio is more than 10; 73% more than 10^2, 31% more than 10^3, 22% more than 10^4 and 11% more than 10^5. Generally, only a small pattern graph (q_1) or a sparse target graph (AS) can refrain $\rho(\mathcal{I}_p)$ from a large value ≥ 100.

The compression ratio $\rho(\mathcal{I}_p)$ is relevant to freedom of the pattern graph. Patten graphs q_8 and q_9 with freedom of 3 have $\rho(\mathcal{I}_p) \geq 39979$ on all target graphs, significantly higher than that of the pattern graphs with freedom of 2.

Table 5: The compression ratio of $\mathcal{I}_p$.

d \ p	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9
GP	333	1435	1263	409	1016	601	862	636433	23871
WB	17	2031	93	27	107	39	127	176833	93842
AS	23	790	80	9	76	12	66	39979	12724
LJ	19	342	581	201	362	400	440	147317	45336
UK	40	787	350	156	348	315	483	238077	130367

Table 6: Preprocessing cost (seconds)

Datasets	GP	WB	AS	LJ	UK
$\mathcal{C}_2$	80	77	76	86	120
$\mathcal{C}_3$	339	155	151	204	1584

Storage Solution. General compression techniques such as *LZO*, *bzip*2 or *snappy* further increases the compression ratio. For example, let the pattern graph be q_9 and the target graph be GP. The storage space of $\mathcal{I}_p$ is 5.5×10^4 petabytes, that of $\mathsf{code}(\mathcal{I}_p)$ is 245 terabytes; by further applying *bzip*2, the space can be brought down to 25 terabytes.

6.2 The Performance of CBF

This section shows the performance of CBF. Table 6 shows the preprocessing time in coding cliques $\mathcal{C}_2$, $\mathcal{C}_3$ for all target graphs. The cost for core-crystal decomposing over all pattern graphs are less than 1 second, conforming Theorem 11.

On Output Crisis. Figure 6 compares the enumeration cost of Crystal against its overall cost in two settings, i) vary the target graph d under a fixed pattern graph q_9 and ii) vary the pattern graph under a fixed target graph LJ.

The output is the bottleneck of the subgraph matching: a shadowed log-scaled bar of enumeration cost takes a small proportion, less than 0.1 on average, of the entire bar of the overall cost. In particular, the compression ratio for q_9 under setting i) is greater than 10^4 on all target graphs. The export of $\mathcal{I}_p$ in a compressed form still dominates the overall cost. This proves the urgency of output crisis and the effectiveness of CBF in its compressed output.

Sensitivity. Crystal was evaluated on a cluster under different memory sizes of each slave and different parallelisms. Parameter virtual core (*Vcore*) of Hadoop adjusts the parallelism of a cluster. Only *enumeration cost* is concerned since output cost is constant under varying system settings.

Figure 7a shows the enumeration cost of Crystal on q_9 and UK when varying the memory size from 1.5 to 4 gigabytes. UK was used since its size of 9.5 gigabytes (Table 4) fitted in the test on memory size. The trend echoes Theorem 4: term $|\mathcal{I}_{\mathsf{core}(p)}|/B$ is invariant under different M while term $M/B \times \Pi_{i\in[1,\lambda]}(|\mathcal{I}_{\mathcal{C}_{x_i}}|/M)$ is linear with $1/M^2$ since the core-crystal decomposition of q_9 (Figure 2) has $\lambda = 3$.

Figure 7b shows the enumeration cost and speedup factor when varying the *Vcore* from 1 to 240. Crystal took about 11 hours to finish using a single core; the enumeration cost was reduced to 309 seconds, gaining a speedup of 128, when employing 240 cores. Such a near-linear speedup is due to the independence among tasks of our serialized algorithm.

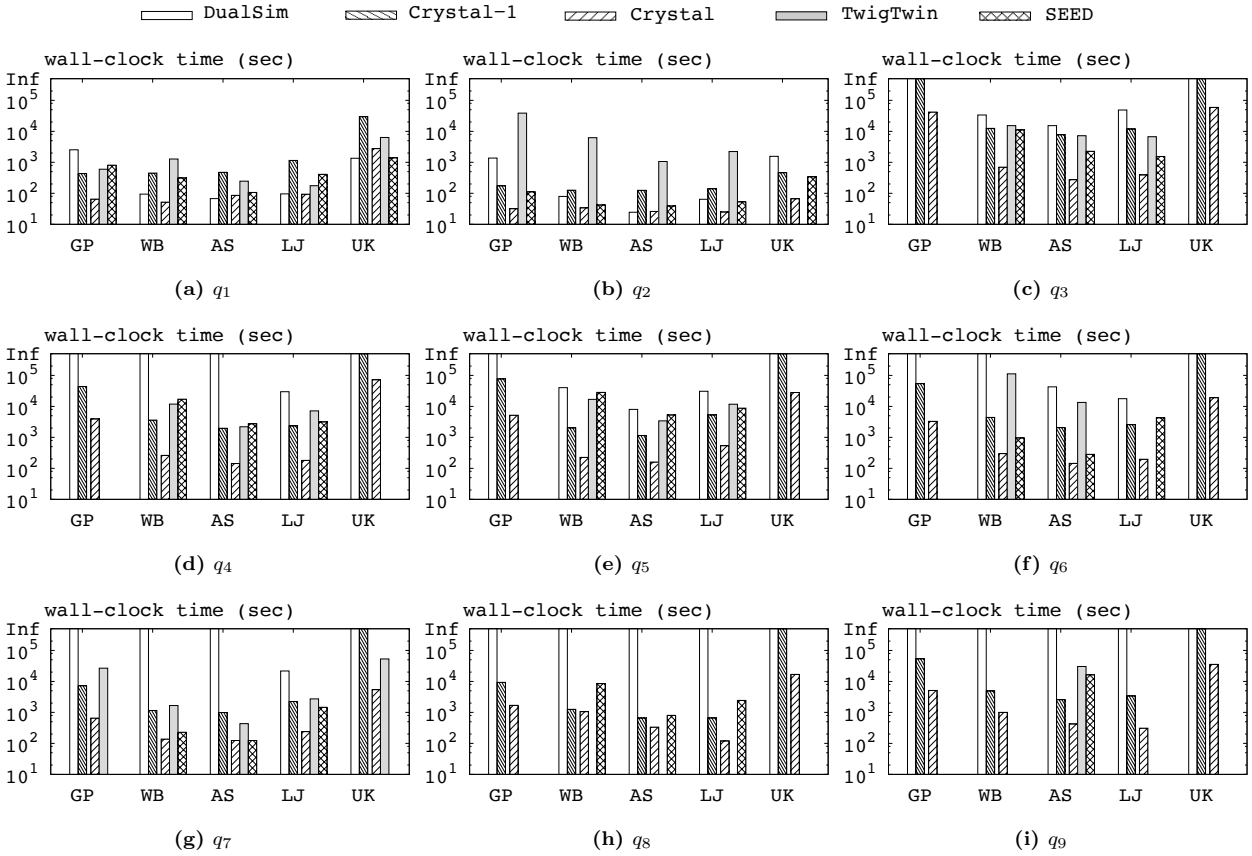


Figure 8: The enumeration time of `Crystal`, `DualSim`, and `SEED`: vary pattern graph

6.3 Compare CBF with Existing Approaches

This section compares our approach against `DualSim`, `TwigTwin` and `SEED` in two groups over all testcases. The output cost of all approaches was discarded for fairness, namely, this section concerns only the *enumeration cost*.

In Figures 8, each cluster of 5 bars compares two groups of approaches on one testcase. Group 1: the first two bars; group 2: the last three bars. Missing bars have either the disk space exceeded the limit of 125 terabytes (SLE) or the memory space exceeded the limit of 4 gigabytes (MLE). The bars reaching the frame-top indicates that the running time exceeded the cut-off time of 1.5 days (RTE). Generally, DFS-style solution `DualSim` failed due to RTE while BFS-style solution `TwigTwin` and `SEED` failed in SLE on gigantic intermediate results. `SEED` got an MLE on `GP` and `UK` for q_7 in loading the $\mathcal{C}_4$ instances in memory in a reduce step.

Group 1: `DualSim` got TLE in 56% cases. In the other cases, `Crystal-1` constantly outperforms `DualSim`. Group 2: `TwigTwin` failed on 42% of the cases, `SEED` failed on 36% of the cases. `Crystal` succeeded on all testcases, is the only survivor on 31% of all cases. `Crystal` outperformed `TwigTwin` in all cases by orders of magnitudes unless the pattern. `Crystal` outperformed `SEED` by a large margin even in log scale in all but one testcases.

In general, our approach is the clear winner in the two groups: it outperforms existing approaches by up to orders of magnitude. In particular, our approach excels in matching complex pattern graphs against dense target graphs.

7. CONCLUSIONS

Subgraph matching has a wide range of applications yet suffers an expensive computation — partially due to the immense size of the instance set $\mathcal{I}$. This paper proposes two techniques for subgraph matching. A vertex-cover based compression (VCBC) provides a storage solution to subgraph matching; a crystal-based framework (CBF) facilitates an efficient subgraph matching computation. VCBC is based on an insight in the structure of $\mathcal{I}$. CBF benefits from 1) exporting $\mathcal{I}$ in a compressed form of VCBC and 2) a refrained export of intermediate results, and is well-suited to parallel computation platforms. Extensive experiments have shown the effectiveness of VCBC and the efficiency of CBF. We shall explore the compression technique on directed or labeled graphs in future.

Acknowledgments

We thank the support of Research Grant for Human-centered Cyber-physical Systems Programme at Advanced Digital Sciences Center from Singapore A*STAR. The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205617].

8. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, pages 62–73, 2013.

[2] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013.

[3] A. Aggarwal and J. S. Vitter. The i/o complexity of sorting and related problems. In *ICALP*, pages 467–478, 1987.

[4] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. In *Proceedings 16th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 241–249, 2008.

[5] U. Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8:450–461, 2007.

[6] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214, 2016.

[7] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[8] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. of Comp.*, 14(1):210–223, Feb. 1985.

[9] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *SIGMOD*, pages 672–680, 2011.

[10] D. J. Cook and L. B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[11] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[12] W. B. Douglas. *Introduction to Graph Theory*. Prentice Hall, 2 edition, 9 2000.

[13] T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. In *FOCS*, pages 614–633, 2015.

[14] W. F. Fan, J. Z. Li, X. W., and Y. H. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[15] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology*, pages 92–106, 2007.

[16] X. Hu, M. Qiao, and Y. Tao. I/o-efficient join dependency testing, loomis-whitney join, and triangle enumeration. *JCSS*, 82(8):1300–1315, 2016.

[17] X. C. Hu, Y. F. Tao, and C. W. Chung. I/o-efficient algorithms on triangle listing and counting. *TODS*, 39(4):27:1–27:30, 2014.

[18] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proceedings of ACM International Conference on Web Search and Data Mining*, pages 673–682, 2012.

[19] H. Kim, J. Lee, S. S. Bhowmick, W. S. Han, J. H. Lee, S. Ko, and M. H. A. Jarrah. DUALSIM: parallel subgraph enumeration in a massive graph on a single machine. In *SIGMOD*, pages 1231–1245, 2016.

[20] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

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

[22] J. Lee, W. S. Han, R. Kasperovics, and J. H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[23] J. Leskovec, A. Singh, and J. M. Kleinberg. Patterns of influence in a recommendation network. In *PAKDD*, pages 380–389, 2006.

[24] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[25] A. Pagh and R. Pagh. Scalable computation of acyclic joins. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 225–232, 2006.

[26] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014.

[27] H. M. Park, S. H. Myaeng, and U. Kang. PTE: enumerating trillion triangles on distributed systems. In *SIGKDD*, pages 1115–1124, 2016.

[28] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[29] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, pages 625–636, 2014.

[30] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[31] Q. Zhang and Y. Xu. Motif mining based on network space compression. *BioData Mining*, 7:29, 2015.