ABSTRACT

Subgraph matching finds a set $I$ of all occurrences of a pattern graph in a target graph. It has a wide range of applications while suffering 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 $I$ as a preprocessing/intermediate/final result or an index, the cost of the export of $I$ 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 $I$? 2) Will the compression of $I$ reversely boost the computation of $I$? For the problem 1), we propose a technique called VCBC to compress $I$ to code($I$) which serves effectively the same as $I$. For problem 2), we propose a subgraph matching computation framework CBF which computes code($I$) instead of $I$ 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.

1. INTRODUCTION

The subgraph matching of a pattern graph $p$ on a target graph $d$ reports the set $I_p$ 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 [3]. 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. 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 [8] 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 $I_p$ whereas subgraph enumeration does not. Such a distinction separates the output cost—the $O(n^p)$ I/Os of exporting $I_p$ to the disk—from the enumeration cost—the cost of subgraph enumeration [16, 26].

The second approach is to study subgraph matching 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, $I_p$ is computed from an intermediate result $I_p^{(i)}$ 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 $I_p^{(i)}$ to initiate the computation of $I_p$. 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

---

Footnote 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 \cite{2}, but their practical performances on real target graphs may not be appealing \cite{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 an application \cite{10} \cite{18} \cite{23} \cite{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 $\Omega(\frac{|\mathcal{I}_p|}{\rho})$ 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 \cite{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 code($\mathcal{I}$). VCBC features an impressive compression ratio, that is, the size of code($\mathcal{I}$) is significantly smaller than that of $\mathcal{I}$. Moreover, 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 code($\mathcal{I}$) in $\Theta(\frac{|\mathcal{I}_p|}{\rho})$ 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 drastically reduces the output cost of subgraph matching by targetting on code($\mathcal{I}_p$) instead of $\mathcal{I}_p$. This breaks the barrier of $\Omega(\frac{|\mathcal{I}_p|}{\rho})$ I/Os when the output cost is the bottleneck of a subgraph matching, yielding a significant improvement in overall performance. Apart from that, CBF leads to up to orders of magnitude speedup over existing approaches in terms of enumeration—computing $\mathcal{I}_p$ without materializing $\mathcal{I}_p$. 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.

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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p, d$</td>
<td>The pattern graph $p$ and target graph $d$.</td>
</tr>
<tr>
<td>$n_p, m_p$</td>
<td>$n_p =</td>
</tr>
<tr>
<td>$g(V')$</td>
<td>The induced subgraph of $g$ on vertex set $V'$.</td>
</tr>
<tr>
<td>code($\cdot$)</td>
<td>The compressed code of a piece of data.</td>
</tr>
<tr>
<td>$\rho$</td>
<td>The compression ratio: Equation 1.</td>
</tr>
<tr>
<td>$\mathcal{I}_p$</td>
<td>The instance set of $p$ — the set of subgraphs of $d$ that are isomorphic to $p$.</td>
</tr>
<tr>
<td>$f_g$</td>
<td>The instance-bijection of instance $g \in \mathcal{I}_p$.</td>
</tr>
<tr>
<td>ord.</td>
<td>The order on $V(p)$ for symmetry breaking.</td>
</tr>
<tr>
<td>$\mathcal{H}_i(g)$</td>
<td>The helve of instance $g$: $f_g(u)$ for all $u \in V_i$.</td>
</tr>
<tr>
<td>$\mathcal{H}(\mathcal{I}_p)$</td>
<td>The set of helves of instances in $\mathcal{I}_p$.</td>
</tr>
<tr>
<td>$\text{img}_u([h])$</td>
<td>${f_g(u)</td>
</tr>
<tr>
<td>$\mathcal{I}_p[h]$</td>
<td>The set of instances in $\mathcal{I}_p$ with helve $h$.</td>
</tr>
<tr>
<td>${V_c, \lambda, P}$</td>
<td>A core-crystal decomposition of $p$.</td>
</tr>
<tr>
<td>$V_c$</td>
<td>A vertex cover of $p$.</td>
</tr>
<tr>
<td>core($p$)</td>
<td>$p(V_c)$, the induced subgraph of $p$ on $V_c$.</td>
</tr>
<tr>
<td>$V_v$</td>
<td>The complement of $V_c$, that is, $V(p) \setminus V_c$.</td>
</tr>
<tr>
<td>$p_1, p_2, \cdots, p_\lambda$</td>
<td>A subgraphs of $p$, where $p_i$ is a crystal $q_{x_1, y_1}, \cdots, q_{x_\lambda, y_\lambda}$, for $i \in [1, \lambda]$.</td>
</tr>
<tr>
<td>$q_{x,y}$</td>
<td>A graph with $y$ nodes fully connected to a $\lambda$.</td>
</tr>
<tr>
<td>$\lambda_x$</td>
<td>A clique of size $x$.</td>
</tr>
<tr>
<td>$M$</td>
<td>Size of the main memory.</td>
</tr>
<tr>
<td>$B$</td>
<td>Size of a disk block.</td>
</tr>
<tr>
<td>$\sigma, \eta$</td>
<td>Two constants defined in the assumption.</td>
</tr>
</tbody>
</table>

#### 2.1 Subgraph Matching

This paper focuses on the subgraph matching on unlabeled and undirected graphs. A graph is 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 $V(g)$, $e(u, v) \in E(g)$. A clique of size $k$ is denoted as $\lambda_k$.

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 \cite{12}).** Given two graphs $g_1$ and $g_2$, an isomorphism from $g_1$ and $g_2$ is a bijection $f : V(g_1) \rightarrow V(g_2)$ 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 $g_2$. 

### Table 1: Notations
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 $V' = \{v_1, v_2, \ldots, v_6\}$. $d(V')$ 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 $\text{ord}_p \subseteq V(p) \times V(p)$ of node pairs in the pattern graph. For each pair $(u,v)$ in $\text{ord}_p$, a partial order $< \in \text{ord}_p$ is imposed such that $u < v$. 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 $u < v$. Each instance $g$ has exactly one valid isomorphism $f_g$ under $\text{ord}_p$. $f_g$ is called the instance-bijection of $g$.

**Example 2.** In Figure 1, pattern graph $p$ uses $\text{ord}_p = \{(u_4,u_5)\}$ 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 $f'(v_6) = u_5$ and $f'(v_5) = u_4$. $\text{ord}_p$ invalidates $f'^{-1}$ since $f'^{-1}(u_4) > f'^{-1}(u_5)$ violates $u_4 < u_5$. The instance-bijection $f_g$ of $g$ under $\text{ord}_p$ is $f_g(u_1) = v_1$, for $i \in [1,6]$.

A mapping function maps a source to its image. For an instance $g$ and its instance bijection $f_g$, we can call $f_g(u)$ the image of $u$ under $g$. We call $\text{Im}_g(u) = \{f_g(u)\} \subseteq \mathcal{I}_p$ 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 $v_1 \in \text{Im}_g(u_1)$.

### 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 $\sigma$.
- $A_2$ $|V(d)| = O(M)$, that is, $|V(d)| \leq \frac{2}{\sigma} M$ for a constant $\eta < 1$ such that $V(d)$ fits in a memory of $M/\sigma$ 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 $\text{code}(D)$, is the compressed form of $D$. $D$ can be restored from $\text{code}(D)$ if the compression is lossless. The compression ratio on $D$ is defined as:

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

In EM model, any algorithm that lists $D$ needs $\Omega\left(\frac{|D|}{B}\right)$ 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 $\text{code}(D)$ in $\Theta\left(\frac{|D|}{B}\right)$ I/Os.

In other words, a $d$-optimal compression guarantees that $\text{code}(D)$ 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 high $\rho(\mathcal{I}_p)$?

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

Problem 2 is dependent on the solution of Problem 1. The cost for exporting $\text{code}(\mathcal{I}_p)$ 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 if $v$ is incident to $v$. Formally, $V_c$ is a vertex cover of $p$ if for all $e(u,v) \in E(p)$, $V_c \cap \{u,v\} \neq \emptyset$.

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 vectorized images of $V_c$ under the instance-bijection $f_g$:

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

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

---

**Figure 1:** Target graph $d$ and pattern graph $p$
3.2 Decompression

Finally, report $I_p \subseteq I'_p$ in two directions.
1. $I_p \subseteq I'_p$: For any instance $g \in I_p$, $g$ will be recovered in the Cartesian product of $S$ in step $D_2$ and pass the validation of $ordp$ in step $D_5$, and thus, $g \notin I'_p$.
2. $I'_p \subseteq I_p$: $I'_p|_h \subseteq I_p|_h$ for all helve $h$. Let $t = \{v_1, v_2, \ldots, v_{n_p}\}$ be a tuple in $I'_p|_h$. To prove $t \in I_p|_h$, it suffices to show that for any edge $(u_i, u_j) \in E(p)$, $(v_1, v_j) \in E(d)$ as $t$ survived through step $D_5$. From the origin of $t$ ($D_2$), there must be an instance $g_0 \in I_p$ with helve $h$, and for each $u_i \notin V_c$, there must be an instance $g_i \in I'_p|_h$ with $f_d(u_i) = v_i$. There is no edge between two nodes in $V_c$. If $u_i$ and $u_j$ are both in $V_c$ then $(v_1, v_j) \notin E(g_0) \subseteq E(d)$; if $u_i$ is in $V_c$ and $u_j$ is in $V_c$ then $(v_1, v_j) \notin E(g_i) \subseteq E(d)$. Thus, $I'_p|_h \subseteq I_p|_h$.

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 $I_p$ 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 $code(I_p)|_{V_c}$ is not longer than that under $V'_c$. Assume, without loss of generality, $V_c = \{v_j | i \in [1, x]\}$ and $V'_c = \{v_j | i \in [1, y]\}$ where $x \leq y$. Let $h$ be a helve of $V_c$. $I_p|_h$ is a disjoint union of $I'_p|_{h'}$ for $\forall h' \in prec(h)$. Here $prec(h)$ 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 $e \in V_c$ and $v \in V(d)$ with $v \notin img_p(u|h)$ under $V_c$, there must be an $e' \in prec(h)$ such that $v \notin img_p(u|h')$ under $V'_c$. Therefore, the length of $code(I_p|_h)$ is no longer than the summation of the lengths of $code(I'_p|_{h'})$, for $\forall h' \in prec(h)$, 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 $C_k$, the compression ratio of the vertex-cover based compression is $O(k)$.
Remarks. This subsection provides two findings on the compression ratio of VCBC, Lemma 4 and 5. 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 5.1.

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 \( \text{code}(I_p) \) from the target graph \( d \) and the pattern graph \( p \).

This section will introduce a Crystal-Based Computation Framework (CBF). CBF computes \( \text{code}(I_p) \) by computing \( \text{code}(I_p|h_i) \) for each helve \( h_i \) in helves \( \mathcal{H}(I_p) = \{ h_1, h_2, \cdots, h_l \} \) with \( l = |\mathcal{H}(I_p)| \), 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 \( I_p \) 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 \( \text{code}(I_p|h_i) \) 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 \( \text{code}(I_p) \). Start with three key components of CBF:

1. Crystals: a pattern of graph patterns 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 \( \text{code}(I_p) \) by assembling 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 \( Q_{x,y} \) is a graph \( g \) with \( x + y \) nodes such that there exists a set \( V' \subseteq V(g) \) of \( x \) nodes and \( V = V(g) \setminus V' \) 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 \( \text{core}(Q_{x,y}) \).
- The induced subgraph \( g(V) \) is an independent set. The nodes in \( V \) are called bud nodes. The edges incident to bud nodes are called bud edges.

**Table 3:** Codes of conditional instance sets.

<table>
<thead>
<tr>
<th>Conditional instance set</th>
<th>Helves on ( V_c )</th>
<th>Image sets on ( V_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_p )</td>
<td>( I_p )</td>
<td>( I_p )</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>( v_2 )</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( u_1, u_2 )</td>
<td>( u_3 )</td>
<td>( u_4, u_5, u_6, u_7 )</td>
</tr>
<tr>
<td>( v_1, v_2, v_3 )</td>
<td>( v_4, v_5, v_6, v_7 )</td>
<td>( v_3, v_4, \cdots, v_9 )</td>
</tr>
</tbody>
</table>

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

**Lemma 3.** \( \text{core}(Q_{x,y}) \) is the induced subgraph of a vertex cover of \( Q_{x,y} \), that is, \( \text{core}(Q_{x,y}) \) covers all edges in \( Q_{x,y} \).

**Example 7.** Figure 2 shows three crystals with cores marked in bold cycles. \( p_1 \) is a \( Q_{1,2} \) with core \( u_1 \), \( p_2 \) is a \( Q_{2,2} \) with core \( (u_2, u_3) \), \( p_3 \) is a \( Q_{2,1} \) with core \( (u_2) \), and \( p_5 \) is a crystal \( Q_{3,1} \).

A crystal \( Q_{x,y} \) is a pattern graph itself. As such, concepts subject to a pattern graph introduced in Section 3.2 apply: \( Q_{x,y} \) has its own instance set \( I_{Q_{x,y}} \), its own helves \( \mathcal{H}(I_{Q_{x,y}}) \), its own conditional instance sets and conditional image sets.

The instance set of a crystal \( Q_{x,y} \) can be coded by VCBC with the instances of \( C_{x+1} \) — the clique of \( x+1 \) vertexes. Let \( C_x \) be a clique with nodes \( v_1, v_2, \cdots, v_x \) in increasing identifiers. Let \( Q_{x,y} \) be a crystal with core nodes \( u_1, u_2, \cdots, u_x \) and bud nodes \( u_1, u_2, \cdots, u_y \). Define the partial order sets.

**Definition 6.** Let \( \text{ord}_x \) include the orders of \( v_1 < v_2 < \cdots < v_x \). Let \( \text{ord}_{Q_{x,y}} \) include the following orders:

\( u_1 < u_2 < \cdots < u_x \), and \( u_1 < u_j < \cdots < u_j \).

**Lemma 4.** Given the instance set of clique \( C_{x+1} \), the code of the instance sets of crystals \( Q_{x,1} \) and \( Q_{x,y} \) can be obtained in a sorting time of \( T(C_{x+1}) \), if \( x \) and \( y \) are \( O(1) \).

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

- Generate \( x+1 \) instances of \( Q_{x,1} \) from an instance \( g \) in \( I(C_{x+1}) \) by mapping the bud node of \( Q_{x,1} \) to each node of \( C_{x+1} \), respectively:
- Group the instances of \( I(Q_{x,1}) \) by their images on \( \text{core}(Q_{x,1}) \). The group of an image \( h \) of \( \text{core}(Q_{x,1}) \) and an image set of the bud node constitutes \( \text{code}(Q_{x,1}|h) \).

\( \text{code}(I_{Q_{x,y}}) \) is obtained by scanning \( \text{code}(I_{Q_{x,1}}) \) \( y \) times. Specifically, let \( u \) be the bud node of \( Q_{x,1} \), and \( u_1, u_2, \cdots, u_y \) be the bud nodes of \( Q_{x,y} \). Let \( h \) be a helve of \( I_{Q_{x,y}} \). Assume that \( \text{img}_{Q_{x,1}}(u|h) \) has \( f \) nodes \( \{ v_1, v_2, \cdots, v_f \} \) where \( v_1 < v_2 < \cdots < v_f \). If \( f < y \) then \( h \) is not a helve of \( I_{Q_{x,y}} \); otherwise, \( \text{code}(I_{Q_{x,y}}|h) \) consists of image sets: \( \text{img}_{Q_{x,y}}(u|h) = \{ v_i, v_{i+1}, \cdots, v_{i+y-1} \} \) 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.
Example 9. For the pattern graph $p$ is a vertex cover of $V$, it is a set of $\mathcal{P}$, every instance of $p$ is made up of a subgraph of $\mathcal{P}$, denoted as $\text{core}(p)$.

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

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

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

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

i. $p_i$ is a crystal $Q_{x_i}, y_i$ for some integers $x_i, y_i$.

Denote the core of $Q_{x_i}, y_i$ as $\text{core}(p_i)$.

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

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

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 5 indicates their consistency.

Lemmma 5. The induced subgraph $p(V_i)$ has no edge.

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

One-Off Assembly. For a core-crystal decomposition of $\{V_i, \lambda, \mathcal{P}\}$, the one-off assembly computes $\text{code}(I_{p_i})$ with instances of core $p_i(V_i)$ and $\text{code}(p_i)$ 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, $p_i(V_i)$ is a subgraph of $p$; the core $p_i$ of $p_i$ is a subgraph of both $p$ and core $p_i$ (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 $g''$ 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 $p'$. $\square$

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

Definition 8. Given a core-crystal decomposition, let $h$ be an instance of $\text{core}(p)$. For a subgraph $p_i \in \mathcal{P}$, $h(\text{core}(p_i))$

Algorithm 1: Assembly

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

Output: code$(I_{p_i}|h)$.

1 for each $u \in V(p)$ do
2 $\text{img}_{h_i}^p(u|h_i) \leftarrow \bigcap_{i \in [1, \lambda]} \text{img}_{h_i}^p(u|h_i)$;
3 $\text{code}(I_{p_i}|h) \leftarrow \text{apply step } C_3 \text{ on } \text{img}_{h_i}^p(u|h_i), u \in V(p)$;
4 $\text{code}''(I_{p_i}|h) \leftarrow \text{Trim code} \text{ code}''(I_{p_i}|h)$: remove a node $v$ in an image set $\text{img}_{h_i}^p(u|h_i)$ if $v$ cannot generate, via step $D_2$, any tuple that survives step $D_3$;
5 return code''$(I_{p_i}|h)$;

is the projection of $h$ on $\text{core}(p_i)$. For simplicity, $h(\text{core}(p_i))$ is denoted as $h_i$, and called the projection of $h$ on $p_i$.

Algorithm 1 shows the one-off assembly under a decomposition $\{V_i, \lambda, \mathcal{P}\}$. For an instance $h$ of $\text{core}(p_i)$, the aim is to generate the image sets of $\text{code}(I_{p_i}|h)$. Obviously, it is not necessary to load the entire image set of each subgraph $p_i \in \mathcal{P}$. All we need are conditional code$(I_{p_i}|h)$, for $\forall i \in [1, \lambda]$, where $h_i$ is the projection of $h$ on $p_i$. (Definition 6). Line 2 obtains the tentative image set of $v \in V(p)$ in $I_{p_i}|h$ by intersecting over corresponding image sets of $I_{p_i}|h$, $i \in [1, \lambda]$. With these image sets, Line 3 simulates the compression step $C_3$ to generate a tentative code''$(I|h)$. Line 4 trims code''$(I|h)$ by simulating depression step $D_2$ and $D_3$ to ensure that code''$(I|h)$ returned in Line 5 is compact.

Example 10. For the pattern graph $p$ in Figure 3, let the decomposition have $V_i = \{v_1, v_2, v_3\}$ and $\mathcal{P} = \{p_1, p_2, p_3\}$ in Figure 3. 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 $I_{p_i}|h$ 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 $\{V_i, \lambda, \mathcal{P}\}$ of $p$, Algorithm 1 assembles code$(I_{p_i}|h)$ for each helve $h$ of $I_{p_i}$ with the codes of subgraphs in $\mathcal{P}$

The proof of Theorem 3. To prove, we need to step into the technique of SimB [13]. Recall that SimB specifies a partial order set $\text{ord}_p$ to avoid duplicated enumeration (Section 2). Actually, SimB identifies $\text{ord}_p$ 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 $\text{V}(p)$. SimB determines $\text{ord}_p$ in rounds. Initially, $\text{ord}_p = \emptyset$. Each round, SimB identifies an equivalence class—a set of nodes $V' \subseteq \text{V}(p)$ that are mutually equivalent under $\text{ord}_p$. 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 $\text{ord}_p$ 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 graph. However, this approach is not scalable to large graphs. So we used a new technique called a core-crystal decomposition. 2Safe abuse since $p_i$ intersects $\text{core}(p)$ exclusively on $\text{core}(p_i)$. 2
graph itself. The problem is to consist the orders in CBF for all decomposed pattern graphs. This can be achieved by leveraging SimB’s freedom in choosing the anchor node for an equivalence class. Given a pattern p and its decomposition \{V, \lambda, \mathcal{P}\}, CBF imposes extra rules to SimB in anchor node selection in determining \text{ord}_p, \text{ord}_\text{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 \(V_i\) nodes are smaller than that of non-\(V_i\) nodes. Then compute \text{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 \((u, v), \text{ or } v, u \text{ in } \text{ord}_p\), the identifier of \(u\) is smaller than \(v\). Let \text{ord}_\text{core}(p) = \text{ord}_p \cap \{u \prec v| \forall u, v \in V\}.

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

Proof. 1) \(p_i\) is a crystal \(\mathcal{Q}_{x_i,y_i}. \text{ ord}_p\) 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 of \text{core}(p) \(\text{core}(\mathcal{Q}_{x_i,y_i})\) in ascending order of \(g(u)\); and do the same for bud nodes. In this way, \(g(p_i)\) can be restored from \text{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 thus is in \(\text{Img}_p(u|h) = \bigcap_{i \in [1,\lambda]} \text{Img}_p(u|h_i)\) (Line 2).

Lemma 8. In Algorithm 7, \text{code}(\mathcal{I}_p|h) reported in Line 5 is exactly \text{code}(\mathcal{I}_p|h).

Proof. We first show that any tuple \(t\) decompressed from \text{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 \(\text{Img}_p(u|h)\) (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 \text{ord}_p, since \(t\) had survived through decomposition 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 \text{core}(p), \((t(u), t(v)) \in E(d)\). If \(u \in V_c\) and \(v \in V_c\), due to condition \((b), \) there exists \(p_i\) with \((u, v) \in E(p_i)\), thus \((t(u), t(v)) \in E(d)\). Lemma 3 guarantees that there is no edge between two nodes in \(V_c\). 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 \text{code}(\mathcal{I}_p|h) (Lemma 7). Note that removing any node in \text{code}(\mathcal{I}_p|h) will lead to a different decompression set (Line 4), violating the fact that the decompression sets of \text{code}(\mathcal{I}_p|h), \text{code}''(\mathcal{I}_p|h) and \text{code}(\mathcal{I}_p|h) are identical. Therefore, \text{code}(\mathcal{I}_p|h) is exactly \text{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 \(\lambda\) 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 4.3 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 \text{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 \(C_{k+1}\). This can be trivially done for \(C_1\) and \(C_2\) whose instance sets are the vertex and edge sets, respectively, of the target graph. The instances of a clique \(C_k\) 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 \(\mathcal{I}_{C_k}\) is dominated by the output cost \(\Theta(\frac{1}{2} |E(d)|^{k/2})\).

This preprocessing step aims at computing, for a parameter \(k_0\), the instance sets of all cliques \(C_k\) with \(k\) from 1 to a certain \(k_0\). LW-Join suits sparse graphs whose total number of instances of clique \(C_k\) is far less than \(|E(d)|^{k/2}\). LW-Join can be trivially done for \(|E(x,y)\) times to obtain \(\mathcal{I}_{C_{k+1}}\) (Lemma 10).

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

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

Lemma 9. Given the instance set of clique \(C_k\), the problem of computing the instance set of \(C_{k+1}\) is a LW-Join.

Proof. Let relation \(R_i, i \in [1, k+1]\), be the instance set \(\mathcal{I}_{C_k}\). Compute the instance set \(\mathcal{I}_{C_{k+1}}\) via the LW-join

\(\bigotimes_{i \in [1, k+1]} R_i\).

The algorithm and analysis in \(16\) show the overall complexity (Lemma 10) where \(\Theta(\frac{1}{2} |\mathcal{I}_{C_{k+1}}|)\) is the output cost.

Lemma 10 (16). The worst-case I/O complexity for computing the instance set of clique \(C_{k+1}\) from that of \(C_k\) is

\(\Theta\left(\frac{1}{2} |\mathcal{I}_{C_k}| \left(\frac{|\mathcal{I}_{C_{k+1}}|}{M}\right)^{1/2} + \frac{1}{B} |\mathcal{I}_{C_{k+1}}|\right)\).

4.3 Core Instance Computation

The core of \(p\) is a pattern graph itself. CBF can compute the instances of \text{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 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 4 to EM model. Recall that given a core-crystal decomposition \( \{V_c, \lambda, P\} \) with \( P = \{p_1, p_2, \ldots, p_\lambda\} \), each \( p_i \) is crystal \( Q_{x_i,y_i} \) for \( i \in [1, \lambda] \). Algorithm 1 assembles \( \text{code}(I_p) \). Specifically, an instance \( h \) of the core(p) is recursively computed (Section 4.3, \( \text{code}(I_p) \)) is pre-computed for each \( p_i \in P \) (Section 4.2). With the projection \( h_i \) of \( h \) on each \( p_i \) (Definition 5), Algorithm 1 assembles \( \text{code}(I_{p_i}[h_i]) \) for all \( p_i \in P \).

The performance of Algorithm 1 under EM model is largely affected by fractional disk accesses — even if \( I_p[h_i] \) has only one instance, Line 2 has to pay one I/O for \( h_i \). In other words, each helve in \( H(I_p) \) consumes at least \( \lambda I\) I/Os, rendering at least \( \lambda I_p/h \) 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 \( P \) 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 \( I_{Q_{x_{i+1},y}} \) is a helve of \( I_{Q_{x_{i+1}}} \) and an instance of clique \( C_x \). We define, for \( h \), a weight \( w(h) \), as the total number of instances of \( Q_{x_{i+1}} \) under helve \( h \). Note that \( w(h) \) is also the size of the only image set of \( \text{code}(I_{Q_{x_{i+1}}}[h]) \).

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 core \( (p_1) \), the weight \( w(h) \) is therefore \( 5 = \{|v_3, v_4, v_5, v_6, v_7\} \) — the size of the image set of the node pair of \( p_1 \). Similarly, for crystal \( p_2, \) the weight \( w(h) \) of \( h_2 = (v_2, v_3) \) is 4; for \( p_i \), the weight of \( v_2 \) is 7.

Lemma 12. Consider clique \( C_{x_{i-1}} \) and its instances \( I_{Q_{x_{i-1}}} \). There exists a mapping function \( \xi_x : \text{code}(I_{Q_{x_{i-1}}})/M \) such that

\[ \xi_x : I_{Q_{x_{i-1}}} \to \{1, 2, \ldots, c_x\} \]

for each \( j \in [1, c_x] \), \( \Sigma_h \xi(h) w(h) \leq (\eta/\sigma) M \).

Proof. Let \( L = \frac{\sigma}{\eta} M \). The mapping function can be obtained with a greedy algorithm. Consider a conceptual sequence of buckets numbered 1, 2, \ldots with capacity \( L \) initially labeled empty. Scan instances of \( C_{x_{i-1}} \) 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 \( C_{x_{i-1}} \), 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, \( c \leq \frac{L}{2} + 1 = O\left(\frac{L}{2}\right) \).

Hash function on core instances. For each crystal \( p_i = Q_{x_i,y_i} \), its helve \( h_i \) is an instance of clique \( C_{x_{i-1}} \). Therefore, hash function \( \xi_x \) defined above can map \( h_i \) to a number in \( [1, c_x] \). For an instance \( h \) of the core(p), recall that \( h \) determines its projections \( h_i \) on each subgraph \( p_i \) (Definition 5). 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 \( \text{vec} = (s_1, s_2, \ldots, s_\lambda) \in [1, c_{x_1}] \times [1, c_{x_2}] \times \cdots \times [1, c_{x_{\lambda}}] \).

An assembly-job of vec loads, for each \( i \in [1, \lambda] \) and each instance \( h_i \) of \( C_{x_{i-1}} \) with \( \xi_{x_i}(h_i) = s_i \), the code \( \text{code}(I_{Q_{x_i}}[h_i]) \) 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 \( \xi(h) = \text{vec} \) and run Algorithm 1 for each of such instances.

Lemma 13. A hash-assembly has \( O(\Pi_{i\in[1,\lambda]}(|I_{Q_{x_{i+1}}}|/M)) \) number of assembly jobs. Each core instance is scanned exactly once in exactly one assembly-job. Each job entails \( O\left(\frac{\eta}{\sigma} M\right) \) I/Os in loading the clique instances into the memory.

Theorem 4. The enumeration cost of the hash assembly of the instance set \( I_p \):

\[ O\left(\frac{|I_{core}|}{B} + \frac{M}{B} \times \Pi_{i\in[1,\lambda]} \left(\frac{|I_{Q_{x_i}}|}{M}\right)\right) \]

I/Os.

4.5 Core-Crystal Decomposition

A core-crystal decomposition \( \{V_c, \lambda, P\} \) supports efficient one-off assembly by restructuring 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 \( \{u_1, u_2, \ldots, u_{\lambda}\} \). Create graph \( p_i \) for each node \( u_i \in V_c \) with \( E(p_i) = \{u_i, v \in V \} \). Let \( P = \{p_1, p_2, \ldots, p_\lambda\} \). \( \{V_c, \lambda, P\} \) is a core-crystal decomposition.

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 \( \Theta\left(\frac{\sigma}{\eta}\right) \) is dependent only on the compression ratio \( p(I_p) \), \( p(I_p) \) is determined by \( V_c \).

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 17 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 \( u \neq v \) 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.
Finally, the complexity of one-off assembly (Theorem 4) instructs the “best” decomposition to minimize the function

\[ f(P) = \frac{M}{B} \times \prod_{i \in [1, \lambda]} \left( \frac{|I_{C_{i}}|}{M} \right). \] (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: \( \lambda \) 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. \( P \) that
   - minimizes \( f(P) \) in Equation (2) if statistical information on the total number of cliques is given;
   - minimizes \( \lambda \) and then minimizes \( x_i \) for each \( i \in \lambda \) if no statistical information is 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 \( P \) 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 \( P \). Note that all bud edges should cover all edges between \( V_c \) and \( V_c \). Therefore, when the core \( p_i \) of a crystal \( P \) is fixed, all possible bud nodes in \( V_c \) should be added to \( P \) to minimize \( \lambda \). 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 \( V_c \).

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 \( V_c \). We construct a bipartite graph, denoted as \( \text{cover-graph}(V_c) \), over \( E' \) and \( C' \). Specifically, the vertex set of \( \text{cover-graph}(V_c) \) 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 the two edges \( (u_1, u_3) \) and \( (u_2, u_3) \). For the vertex cover \( V_c = \{u_1, u_2, u_3\} \) of the pattern graph \( p \) in Figure 1, the cover graph \( \text{cover-graph}(V_c) \) is shown in Figure 3.

Now the optimization problem is defined as a cover-graph problem defined as below.

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

This is a cover problem on a bipartite graph.

**Theorem 5.** Optimize-P can be solved with an algorithm in \( O(2^m m_p (2^m + 2^m)) \) time with space \( O(2^m) \).

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

\[ \log(f(P)) = \log(M/B) + \sum_{i \in [1, \lambda]} \log(\frac{|I_{C_{i}}|}{M}). \]

is decided by the summation of \( \log(\frac{|I_{C_{i}}|}{M}) \) over the selected cliques in \( C' \). The problem can then be resolved with memorized search—a dynamic programming 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 \( \log(\frac{|I_{C_{i}}|}{M}) \) over \( C_{i} \) selected by \( C'' \). \( DP[E''] \) does not have to store \( C'' \). \( C'' \) can be restored by tracing from \( DP[E'' \setminus E'''] \) 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 \( O(2^m) \) cliques in \( V_c \). Case 2. To find the \( P \) with the parameter \( \lambda \), we start our search with \( \lambda = |V_c| \) provided by the initial decomposition (Definition 12). It remains to enumerate \( I'(C''|V_c) = O(2^m 2^{|E|}) \) combinations of elements in \( C'' \) with 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 \( \text{par} = O(\prod_{i \in [1, \lambda]} |I_{C_{i}}|/M) \) 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 3

\[ \tilde{O} \left( |\text{core}(p)| + M \times \prod_{i \in [1, \lambda]} \left( \frac{|I_{C_{i}}|}{M} \right) \right). \]

Besides, the loading process, since each bucket is stored consecutively, can be completed in \( \lambda \) 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 \( Z_p \), 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 output compression. In subgraph matching, output compression is vertical to input compression \([14][21][28]\). 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 \([1][20][29][19]\), or choose not to output at all, see the seminal work of \([20]\). The former ones, unavoidably, entail \(\Omega(\frac{|\mathcal{I}_p|}{p})\) 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 thoroughly 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 \([20][20][21]\), 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 \([21]\) takes 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 \([2]\).

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 \(\text{http://law.di.unimi.it/datasets.php}\) while other datasets were downloaded from \(\text{https://snap.stanford.edu/data/}\). The statistics of the target graphs \(d\) include graph size, average degree (avg-deg) and degeneracy. \(\text{avg-deg}(d) = \frac{2|\mathcal{E}(d)|}{|\mathcal{V}(d)|}\) 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 by the compression ratios of output compression. In subgraph matching, output compression is vertical to input compression \([14][21][28]\). 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 \([1][20][29][19]\), or choose not to output at all, see the seminal work of \([20]\). The former ones, unavoidably, entail \(\Omega(\frac{|\mathcal{I}_p|}{p})\) I/Os for export; whereas the latter ones, suffer a re-computation cost of \(\mathcal{I}_p\) upon every following request.
Table 4: Datasets

| dataset          | $|V(d)| \times 10^6$ | $|E(d)| \times 10^6$ | avg-deg | deg-ecr | 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 $q_0$. (b) Vary $p$. Let $d$ be LJ.

Figure 6: Costs of Crystal.

Table 5: The compression ratio of $I_p$.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
<th>$q_8$</th>
<th>$q_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>1345</td>
<td>1263</td>
<td>409</td>
<td>1016</td>
<td>601</td>
<td>623</td>
<td>364433</td>
<td>93842</td>
<td>23971</td>
</tr>
<tr>
<td>AS</td>
<td>23</td>
<td>790</td>
<td>80</td>
<td>9</td>
<td>12</td>
<td>127</td>
<td>176833</td>
<td>93842</td>
<td>12724</td>
</tr>
<tr>
<td>LJ</td>
<td>19</td>
<td>342</td>
<td>581</td>
<td>201</td>
<td>362</td>
<td>400</td>
<td>440</td>
<td>147317</td>
<td>45336</td>
</tr>
<tr>
<td>UK</td>
<td>40</td>
<td>787</td>
<td>350</td>
<td>156</td>
<td>348</td>
<td>315</td>
<td>483</td>
<td>238077</td>
<td>130367</td>
</tr>
</tbody>
</table>

Table 6: Preprocessing cost (seconds)

<table>
<thead>
<tr>
<th>Datasets</th>
<th>GP</th>
<th>WB</th>
<th>AS</th>
<th>LJ</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_2$</td>
<td>80</td>
<td>77</td>
<td>76</td>
<td>86</td>
<td>120</td>
</tr>
<tr>
<td>$C_3$</td>
<td>339</td>
<td>155</td>
<td>151</td>
<td>204</td>
<td>1584</td>
</tr>
</tbody>
</table>

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.

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 $V_c = V(p) \setminus V_c$ be $V_c$’s complement. The freedom of $V_c$ is $|V_c|$. If $V_c$ is a minimum vertex cover of $p$, then $|V_c|$ is also called the freedom of $p$.

Figure 7 shows the compression ratios of $I_p$ 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 in marked 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 $I_p$ over all testcases.

$\rho(I_p)$ is significant: in 98% of the testcases in Table 5, the compression ratio is more than 10; 73% more than 10 or 31%, 22% more than 10 or 41% more than 10. Generally, only a small pattern graph ($q_1$) or a sparse target graph (AS) can refrain $\rho(I_p)$ from a large value $\geq 100$.

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

Storage Solution. General compression techniques such as LZO, bzip2 or snappy further increases the compression ratio. For example, let the pattern graph be $q_8$ and the target graph be GP. The storage space of $I_p$ is 5.5 $\times 10^4$ petabytes, that of code($I_p$) is 245 terabytes; by further applying bzip2, the space can be brought down to 25 terabytes.

6.2 The Performance of CBF

This section shows the performance of CBF. Table 5 shows the preprocessing time in coding cliques $C_2$, $C_3$ 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 $q_0$ 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 $I_p$ 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 7 shows the enumeration cost of Crystal on $q_0$ 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 1 term $|I_{core(p)}|/B$ is invariant under different $M$ while term $M/B \times \Pi_{c \in \Lambda}(|I_{core(c)}|/M)$ is linear with $1/M^2$ since the core-crystal decomposition of $q_0$ (Figure 2) has $\lambda = 3$.

Figure 7 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...
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 q7 in loading the C4 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.

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 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 I. CBF benefits from 1) exporting 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