BOOMER: Blending Visual Formulation and Processing of P-Homomorphic Queries on Large Networks

Yinglong Song [‡,§] Huey Eng Chua[‡] Sourav S Bhowmick[‡] Byron Choi[†] Shuigeng Zhou[§]

[‡]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[§]Shanghai Key Lab of Intelligent Information Processing, School of Computer Science, Fudan University, China
[†]Department of Computer Science, Hong Kong Baptist University, Hong Kong
hechua|assourav@ntu.edu.sg,bchoi@comp.hkbu.edu.hk,ylsong15|sgzhou@fudan.edu.cn

ABSTRACT

Visual graph query interfaces (a.k.a GUI) make it easy for non-expert users to query graphs. Recent research has laid out and implemented a vision of a novel subgraph query processing paradigm where the *latency* offered by the GUI is exploited to *blend* visual query construction and processing by generating and refining candidate result matches iteratively during query formulation. This paradigm brings in several potential benefits such as superior *system response time* (SRT) and opportunities to enhance usability of graph databases. However, these early efforts focused on subgraph isomorphism-based graph queries where blending is performed by iterative edge-to-edge mapping. In this paper, we explore how this vision can be realized for more generic but complex *1-1 p-homomorphic* (p-hom) queries introduced by Fan *et al.* A 1-1 p-hom query maps an *edge* of the query to *paths* in the data graph. We present a novel framework called Boomer for blending *bounded* 1-1 p-hom (BPH) queries, a variant of 1-1 p-hom where the length of the path is bounded instead of arbitrary length. Our framework is based on a novel online, *adaptive indexing scheme* called CAP *index*. We present two strategies for CAP index construction, *immediate* and *deferment-based*, and show how they can be utilized to facilitate judicious interleaving of visual BPH query formulation and query processing. Boomer is also amenable to modifications to a BPH query during visual formulation. Experiments on real-world datasets demonstrate both efficiency and effectiveness of Boomer for realizing the visual querying paradigm on an important type of graph query.

ACM Reference Format:
Yinglong Song [‡,§] Huey Eng Chua[‡] Sourav S Bhowmick[‡] Byron Choi[†] Shuigeng Zhou[§]. 2018. BOOMER: Blending Visual Formulation and Processing of P-Homomorphic Queries on Large Networks. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3183713.3196902

1 INTRODUCTION

Given two vertex-labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of graph homomorphism (resp. subgraph isomorphism) is to find a (resp. 1-1) mapping from V_1 to V_2 such that each vertex in V_1 is mapped to a (resp. distinct) vertex in V_2 with the same label, and each edge in E_1 is mapped to an edge in E_2. Querying graphs based on this conventional notion of graph homomorphism or subgraph isomorphism is often restrictive in many real-world graph applications [13, 14]. Consequently, Fan et al. [13] extended these two notions to *p-homomorphism* (p-hom) and *1-1 p-hom* (1-1 refers to the 1-1 mapping [13]), respectively, by mapping *edges* from one graph to *paths* to another and by measuring *similarity* of vertices instead of simply vertex label equality. It has been reported in [13, 14] that due to its generality, p-hom and 1-1 p-hom-based queries have a variety of real-world applications such as website matching, identification of suspects in criminal networks, plagiarism, and complex objects (*e.g.,* face) recognition. However, formulating these queries textually using a graph query language requires a user to have programming and debugging expertise. Unfortunately, this assumption makes it harder for non-programmers to take advantage of a p-hom search framework.

A popular approach to make formulation of p-hom query accessible to non-programmers is to provide a visual query interface (GUI) for interactive construction of queries. This has recently paved the way for several visual graph query-based techniques and applications [2]. In particular, Jin *et al.* [21, 22] and Hung *et al.* [20] proposed an innovative *visual substructure search* (*i.e.,* exact subgraph and substructure similarity [4, 18, 37]) query processing paradigm where instead of processing a query graph after its construction, it *blends* (*i.e.,* interleaves) visual query construction and processing. When a user visually constructs an edge during query formulation, the current query fragment is evaluated by exploiting the GUI *latency* (*i.e.,* time to construct a query vertex or an edge visually) and a set of candidate matches containing the query fragment is generated using underlying *index structures.* Next, when she constructs another edge, the candidate set is refined by filtering irrelevant matches. This continues until the user has completed formulation of her query and the final results are computed by utilizing the refined candidate set.

The above visual querying paradigm brings in at least two key benefits [2]. First, it improves *system response time* (SRT), which is the duration between the time a user presses the Run icon to the time when she gets the query results. Second, it opens up opportunities to enhance usability of graph databases (*e.g.,* exploratory search [19]).

Early efforts in [20–22] that implemented the aforementioned vision limited their focus on blending subgraph isomorphism-based graph queries. Consequently, it is an open problem whether more complex but generalized graph queries such as p-hom queries can be realized in this paradigm. In this paper, we present a novel framework called Boomer (**Bo**unded 1-1 p-h**om** Qu**e**ry Blende**r**) to realize the visual graph querying paradigm for a class of p-homomorphic graph queries referred to as *bounded 1-1 p-hom* (bph) queries. A bph query is derived from 1-1 p-hom query introduced by Fan et al [13]. Intuitively, in a bph query an edge is mapped to a path of *bounded* length (*i.e.,* it satisfies certain length constraints) in the underlying data graph instead of mapping it to a path of arbitrary length. Specifically, each edge (q_i, q_j) in a bph query is labeled with a pair of integers $[lower, upper]$, referred to as *lower* and *upper bounds*, respectively, representing the minimum and maximum allowable path lengths connecting a vertex pair (v_i, v_j) in a data graph that *match* q_i and q_j, respectively. Consider the following user problem.

Example 1.1. A key task for exploring new drugs at preclinical and clinical trials during drug development is the identification of suitable animal models with disease-related biological processes that are highly similar to that in human [26]. Bob, a biologist is interested in finding out if *C. elegans* is a suitable animal model for studying apoptosis in *human*. Given the knowledge of protein-protein interaction (ppi) of genes related to apoptosis in *C. elegans* (Figure 1(a)), he first needs to identify the corresponding homologs[1] in its *human* counterpart (Figure 1(b)). This can be done through a literature search or using databases such as *OrthoList* [27]. Consequently, he extracts the subgraph involving four relevant genes (*i.e.*, `egl-1`, `ced-3`, `ced-4`, `ced-9`) in *C. elegans* as shown in Figure 1(a) (color-coded vertices). Next, he finds corresponding homolog genes (`bid`, `casp 3`, `apaf1`, and `bcl2`, respectively) in *human*. Using these homolog genes and the subgraph structure, Bob wishes to search the *human* ppi (Figure 1(b)) to determine if *C. elegans* is an appropriate organism for studying apoptosis process in human. Bob is aware that there may not be strict correspondence between the two models due to evolution. Consequently, there may not exist an exact match of the subgraph on *human* ppi. However, the pairs of genes (e.g., (`bcl2`, `casp 3`)) should not be far apart in the *human* ppi either as too large distance between them indicates that the interaction is unlikely to be conserved. Hence, Bob wishes to formulate a bph query as shown in Figure 1(c). Note that the upper bound constraints can be varied depending on prior knowledge of potential deviation due to evolution. As Bob is a non-programmer, he wishes to formulate it using a user-friendly gui using click-and-drag. A matching result of this query on the *human* ppi is shown in Figure 1(d). *How can Bob formulate and process such queries?*

Another application of bph query is in the identification of putative drug targets in cancer where certain oncogenes[2] (*e.g.*, `myc`) are deemed to be "undruggable" [9]. We can formulate a bph query with a lower bound[3] of 2 or 3 to find these putative targets.

Observe that bph queries are more general than existing subgraph isomorphism-based queries as the edge-to-edge mapping of

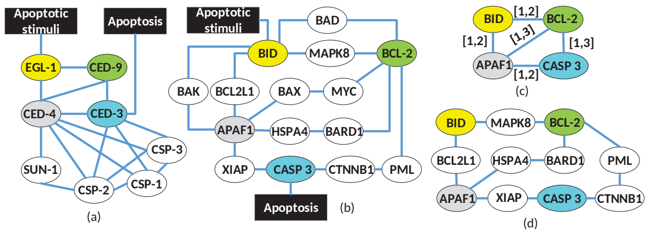

Figure 1: Apoptotic pathway of (a) *C. elegans* and (b) *human* in BioGRID database; (c) bph query on (b); (d) A matching result.

the latter is unable to specify such connectivity constraints in a data graph. Particularly, when $lower = upper = 1$, the bounded 1-1 p-hom-based matching reduces to the subgraph isomorphism-based matching. Hence, our Boomer framework can be utilized to formulate both bph and exact subgraph search queries (detailed in Section 4). Similar to [13, 14], in this work we assume that the bph queries are *ad hoc* queries (*i.e.,* user's queries cannot be anticipated and optimized for) formulated on a large network residing in a *single machine*.

Designing a framework to support efficient blending of bph query formulation and processing is challenging. Predecessor efforts [20–22] follow the *immediate evaluation* strategy, *i.e.,* they follow the edge formulation sequence to efficiently process visually constructed edges iteratively by utilizing the gui latency. Although this strategy is effective for substructure search queries as it involves iterative edge-to-edge mapping, for bph queries we need to match a query edge $e = (q_i, q_j)$ to paths of bounded length within the available gui latency. This can be prohibitively expensive when there are many matches to q_i and q_j in the data graph and the upper bound constraint associated with e is relatively large. Consequently, this may delay the processing of subsequent vertices/edges in the query leading to a larger srt. Besides, since the processing of a constructed vertex or edge depends on the results of the previous step, it is important to design an efficient online, adaptive index structure that can maintain the partial candidate matches to the (partially) formulated query by utilizing the limited gui latency. However, it is expensive to maintain all intermediate vertices and edges of paths matching to a query edge as it may result in indexing a large portion of the underlying network.

In this paper, we present a novel online, adaptive indexing scheme called cap ***(C**ompact **A**daptive **P**ath) index* to facilitate judicious interleaving of visual bph query formulation and query processing. Intuitively, it is a graph-structured index that efficiently stores the candidate matching vertices that satisfy the *upper* bound constraints of the edges in a (partially) constructed bph query. Note that we *defer* the checking of paths satisfying the *lower* bound greater than 1 *until* the Run icon is clicked to execute the query so that it does not impose additional cost during cap construction. Specifically, we exploit the gui latency in the following novel ways to construct and maintain the index. First, we take a non-traditional strategy of judiciously *deferring* evaluation of current query edge, if necessary, so that it can exploit the latency associated with the formulation of a future edge and the clicking of Run icon. Second, since visualizing large graphs is cognitively challenging, it is extremely difficult to comprehend the structural relationships among the vertices in a

[1]Homolog genes which have conserved interactions across multiple organisms form the basis for translating knowledge of biological processes from one organism to another organism [34].
[2]Genes which have the potential to transform a cell into a tumor cell.
[3]Cancer targets are typically found to be 1 to 2 hops away from oncogenes [33].

matching result overlaid on a large network. Hence, in Boomer *each* result match of a query is displayed by visualizing a small subgraph of the network that contains it. Consequently, the gui latency associated with visualizing the result matches iteratively is exploited by Boomer to filter matching vertices based on *lower* bounds specified in the query. We also show how the cap index can efficiently support query modifications by a user. In summary, the main contributions of this paper are as follows.

- *A novel framework*. We present a novel framework for blending visual bounded 1-1 p-hom (bph) query formulation and processing. To the best of our knowledge, this is the first effort that presents an effective solution to interleave visual formulation and processing of a type of p-hom queries.
- *A new online index*. We present a novel adaptive, on-the-fly, space-efficient index called cap to facilitate efficient pruning and retrieval of matching vertices satisfying a (partial) bph query. It is also amenable to modification to the visual query at any time during its construction.
- *Fast evaluation of BPH queries*. We present algorithms for efficient evaluation of bph queries and visual query modifications by efficiently exploiting the cap index and gui latency offered by the visual querying paradigm.
- *Experimental study*. Using real-world datasets and user study, we show the effectiveness of our framework in supporting bph queries in this new visual querying paradigm. Specifically, our *deferment-based strategy* to evaluate query vertices and edges is superior to the *immediate evaluation strategy*.

Paper Outline. Section 2 defines the preliminary concepts. We formally introduce the visual bph querying problem and the Boomer framework in Sections 3 and 4. We present the cap index and how it can be leveraged to facilitate blending of bph query formulation and processing in Section 5. Section 6 discusses efficient query modification. Experimental results are presented in Section 7. We review related research in Section 8. The last section concludes the paper. The proofs of all lemmas and formal code of key procedures are presented in Appendices A and B, respectively.

2 BACKGROUND

A *data graph* is an undirected, simple graph $G = (V, E, L)$ where V is the set of vertices, $E \subseteq V \times V$ is a set of undirected edges, and L is a vertex labelling function. We denote a *path* between a pair of vertices (v, v') in G as $p : v \rightarrow \cdots \rightarrow v'$. The *length* of a path p counts the number of edges in p and is denoted as $length(p)$. *Distance* between (v, v') is defined as the length of the shortest path between these vertices and is denoted as $dist(v, v')$. A degree of a vertex $v \in V$ is denoted as $deg(v)$.

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a mapping function $\varphi : V_1 \longrightarrow V_2$ is a called a *homomorphism* if and only if, for all $v_i, v_j \in V_1$, if $(v_i, v_j) \in E_1$, then $(\varphi(v_i), \varphi(v_j)) \in E_2$ [28]. In contrast, *subgraph isomorphism* seeks to identify an injective mapping ϕ between G_1 and G_2 such that for every pair of vertices $v_i, v_j \in V_1$, $(v_i, v_j) \in E_1$ implies $(\phi(v_i), \phi(v_j)) \in E_2$.

Fan et al. [13] introduced the notions of *p-homomorphism* and *1-1 p-homomorphism* for directed graphs to relax graph homomorphism and subgraph isomorphism, respectively. *P-homomorphism* extends homomorphism by mapping every edge from one graph to a path

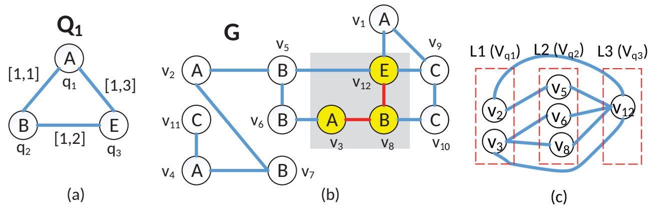

Figure 2: (a) bph query, (b) data graph, and (c) cap index.

(instead of an edge) in another and match the vertices based on their *similarity* [13]. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a *similarity matrix* $\mathbb{M}$[4] and a *similarity threshold t*, G_1 is *p-homomorphic* (p-hom) to G_2 if there exists a mapping ξ from V_1 to V_2 such that for every vertex $v \in V_1$, (1) if $\xi(v) = u$, then $\mathbb{M}(v, u) \geq t$ where $u \in V_2$, and (2) for each edge $(v, v') \in E_1$, there exists a non-empty path $p : u \rightarrow \cdots \rightarrow u'$ in G_2 such that $\xi(v') = u'$. Briefly, every edge from v is mapped to a path originating from u and $\xi(.)$ is a p-hom mapping function from G_1 to G_2.

Further, G_1 is *1-1 p-homomorphic* (1-1 p-hom) to G_2 if $\xi(.)$ is a 1-1 (injective) p-hom mapping from G_1 to G_2. That is, distinct vertices in G_1 have to be mapped to *distinct* matches in G_2. Observe that subgraph isomorphism and graph homomorphism are special cases of 1-1 p-hom and p-hom, respectively [13]. It is intractable to determine whether a graph is p-hom or 1-1 p-hom to another [13].

3 BOUNDED 1-1 P-HOMOMORPHIC QUERY

In this section, we introduce the notion of *bounded* 1-1 p-homomorphic query and the *visual bounded 1-1 p-hom search* problem.

3.1 Definition

Given a data graph $G = (V, E, L)$, intuitively a 1-1 p-hom query Q_P finds one or more subgraphs in G that are 1-1 p-hom match to Q_P. Recall that in 1-1 p-hom matching an edge is mapped to a path of arbitrary length. However, in reality the relationship between two entities in a network become less relevant as the distance between them increases [6]. Consequently, it is appropriate for *matching paths* of an edge in Q to be *bounded* (i.e., they satisfy certain length constraints). We refer to this variant of 1-1 p-hom query as *bounded 1-1 p-hom query*. Specifically, a *bounded 1-1 p-hom query* (bph query) $Q_B = (V_B, E_B, L, \lambda)$ is a connected, undirected, simple labeled graph where V_B and E_B are the sets of query vertices and edges, respectively. Q_B has the same labelling function L as G and $q_i \in V_B$ *matches* $v_j \in V$ if $L(q_i) = L(v_j)$. Further, λ is an edge labelling function that assigns an edge $e_q = (q_i, q_j)$ in Q_B to a pair of integers $[lower, upper]$, referred to as *lower* and *upper bounds*, respectively, representing the minimum and maximum allowable path lengths connecting a vertex pair (v_i, v_j) where $v_i \in V$, $v_j \in V$, $L(q_i) = L(v_i)$ and $L(q_j) = L(v_j)$. That is, $\lambda(e_q) = [lower, upper]$. We shall refer to the lower and upper bounds of e_q as $e_q.lower$ and $e_q.upper$, respectively. Note that $e_q.lower \geq 1$, and $e_q.lower \leq e_q.upper$. Figure 2(a) shows an example of a bph query.

Note that in practice often all edges in a bph query have $e_q.lower = 1$. However, in certain applications *lower* of an edge may be greater than 1. For example, given a user A in a social network, we may wish to explore the friends-of-friends (fof) neighborhood of A. In

[4]For each pair (v, u) of vertices in $V_1 \times V_2$, $\mathbb{M}(v, u)$ is a number in $[0, 1]$, indicating how close labels of v and u are.

this case the query edge connecting A to a vertex in FOF has a lower bound of 2. Another example is given in Example 1.1.

Definition 3.1. **[Bounded 1-1 p-hom-based Matching]** *Given a data graph $G = (V, E, L)$ and a BPH query $Q_B = (V_B, E_B, L, \lambda)$, a set of distinct vertices $V_D \subseteq V$ is a **bounded 1-1 p-hom match** of Q_B if and only if the following constraints are satisfied:*

(1) ***Label constraint**: $L(q) = L(v), \forall q \in V_B, v \in V_D$,*
(2) ***Query size constraint**: $|V_D| = |V_B|$, and*
(3) ***Edge bound constraint**: if there exists an edge $e_q = (q_i, q_j)$, $e_q \in E_B$ with bounds $[lower, upper]$, there must exist a non-empty path $p : v \to \cdots \to v'$ in G where $L(q_i) = L(v)$, $L(q_j) = L(v')$ and $length(p) \in [lower, upper]$. Such a path p is referred to as a **matching path** of e_q.*

Observe that when $e_q.lower = e_q.upper = 1$, the bounded 1-1 p-hom-based matching reduces to the subgraph isomorphism-based matching.

We refer to the order in which the vertices of a BPH query is matched as the *matching order* (denoted as M). Note that in visual graph querying environment, this may simply be the order in which a user constructs the query vertices. For example, a matching order of Q_1 in Figure 2 is $M : q_1 \to q_2 \to q_3$ (*i.e.,* a user may visually construct the vertices in this sequence). We denote $|M| = |V_B|$.

3.2 Structure of Visual Interface

Several real-world academic [7, 17, 22, 24, 25, 32] and commercial (*e.g., Pubchem*) visual interfaces for subgraph query construction largely focus on formulating substructure search queries. We can easily extend these interfaces to support visual formulation of BPH queries. Such interface consists of the following main panels. (a) *Data Panel*: This panel displays the data graphs (networks) available for querying. (b) *Attribute Panel:* It displays a list of attributes (*e.g.*, labels) of vertices or edges of a network. In BOOMER, labels of the vertices are displayed. (c) *Query Panel:* It is used for constructing a BPH query graphically. (d) *Results Panel:* This panel displays the result subgraphs of a BPH query.

Then, we can construct a BPH query by performing the following steps: (1) Move the mouse cursor to the *Attribute Panel.* (2) Scan and select an attribute (*e.g.*, label `BCL2`). (3) Drag the selected attribute to the *Query Panel* and drop it to create a single vertex of the BPH query. (4) Repeat Steps 1-3 to construct another vertex. (5) Construct an edge between the vertex pair by clicking on the vertex pairs. (6) Specify the lower and upper bounds of the edge by clicking on it and filling a combo box. The default value is [1,1]. (7) Repeat Steps 4-6 until the entire query is formulated.

3.3 Visual Bounded 1-1 p-Hom Search Problem

Given a BPH query Q_B visually constructed on a visual query interface and a data graph $G = (V, E, L)$, the goal of *visual bounded 1-1 p-hom search* problem is to retrieve *all* bounded 1-1 p-hom matches of Q_B in G by interleaving (*i.e.,* blending) formulation and processing of Q_B. For example, consider the BPH query in Figure 2(a) and the data graph in Figure 2(b). Assume that the query is formulated using a visual interface. Then the goal of the search problem is to find the result matches (*i.e.,* result subgraphs) in G by blending its formulation and processing. For instance, a valid result match of the query is $q_1 \to v_2, q_2 \to v_5, q_3 \to v_{12}$.

4 THE BOOMER FRAMEWORK

BOOMER consists of the following two major components.

The Preprocessor. BOOMER preprocesses the input data graph G to compute the *average edge processing time t_{avg}* and the *pruned landmark labelling* (PML) index $\mathbb{P}$ [1] which are used during CAP index construction (in BPH query blender module). The former is used for *prioritizing* the edges of a BPH query and *determining* when they should be processed. The latter is used for processing distance queries. Briefly, PML is based on *2-hop cover* and used for fast exact shortest-path distance computation[5].

For a given data graph G, this module first computes the PML index which facilitates 2-hop queries of any vertex pair. Next, it computes t_{avg} of G as follows. It utilizes the PML index to process 1 million randomly selected distance queries between vertex pairs in G. t_{avg} is then computed as the average processing time of these distance queries. We observe that t_{avg} computation time is cognitively negligible and PML computation time is less than 15 min for several large networks (t_{avg} and PML computation time for a set of data graphs are reported in [29]). Hence, the empirical determination of t_{avg} can be completed within several minutes for a large network. We emphasize that the preprocessing step is only one-time cost as it is performed offline and only once for a given data graph.

The BPH Query Blender. Algorithm 1 outlines the procedure for blending formulation and processing of a visual BPH query as well as query modification during construction. Let Q_B be a visual BPH query being formulated by a user. The framework monitors four visual actions, namely, `NewVertex` for adding a new vertex to Q_B, `NewEdge` for adding a new edge to Q_B, `Modify` for removing an existing edge or updating the bounds, and `Run` for executing the current BPH query. In particular, BOOMER uses an *action stream* (denoted as *$stream$*) to keep track of the vertices and edges added during query construction and processes these actions to construct the online CAP index (Lines 3 to 8). Observe that initiation of index construction happens during query construction and completes only after the query is completely formulated (when the `Run` icon is clicked). Specifically, three CAP index construction strategies are proposed, namely, *Immediate*, *Defer-to-Run*, and *Defer-to-Idle*, to judiciously utilize the availability of GUI latency during query formulation (detailed in Section 5). Since a user may modify the bounds of a previously formulated edge or delete an existing edge during query formulation, the modification of BPH query is handled in Lines 9-10, which updates the CAP index efficiently (discussed in Section 6). When the user clicks the `Run` icon, BOOMER proceeds to complete the CAP index construction (Lines 12 to 16). Using the constructed CAP index and matching order M, BOOMER computes a list of result vertex set that satisfies the upper bound constraints of the edges in Q_B (Line 18). Finally, when a user iteratively selects a particular query result G_B to visualize, BOOMER generates the result subgraph in G_B that satisfies the lower bound constraints of the edges in Q_B (Lines 20 to 24). We elaborate on Lines 18-23 in Section 5.4.

Interested reader may refer to [29] for a case study of BOOMER framework based on Example 1.1.

[5]Although we use the PML index, our framework is orthogonal to the choice of exact shortest-path distance computation technique. Any existing efficient technique can be plugged into our framework.

Algorithm 1 Boomer Query Blender.

```
Require: GUI action $a$, data graph $G = (V, E, L)$, a BPH query $Q_B$, PML index $\mathbb{P}$, average edge
    processing time $t_{avg}$;
Ensure: Set of bounded 1-1 p-hom result subgraphs $H$;
 1: $pool, stream, \mathbb{C}, V_\Delta \leftarrow \phi$
 2: $CAPCompleted \leftarrow false$
 3: if $a$ is NewVertex or $a$ is NewEdge then
 4:     $stream \leftarrow$ Insert($a$, $stream$)
 5:     $M \leftarrow$ UpdateMatchingOrder($a$, $M$)
 6:     for $a \in stream$ do
 7:         $\mathbb{C}, pool \leftarrow$ ConstructCAP($a, stream, \mathbb{C}, G, Q_B, \mathbb{P}, t_{avg}, pool$) /* Section 5*/
 8:     end for
 9: else if $a$ is Modify on edge $e_q$ then
10:     $\mathbb{C}, pool \leftarrow$ UpdateCAP($a, stream, \mathbb{C}, Q_B, G, \mathbb{P}, pool, t_{avg}, e_q$) /* Section 6 */
11: else if $a$ is Run then
12:     if $stream \neq \phi$ then
13:         for $a \in stream$ do
14:             $\mathbb{C}, pool \leftarrow$ ConstructCAP($a, stream, \mathbb{C}, G, Q_B, \mathbb{P}, t_{avg}, pool$)
15:         end for
16:     end if
17:     $CAPCompleted \leftarrow true$
18:     $V_\Delta \leftarrow$ PartialVertexSetsGen($Q_B, M, \mathbb{C}$)/* Section 5.4*/
19: end if
20: while $CAPCompleted = true$ do
21:     if $a$ is $G_B$ with partial-matched vertex set $V_P \in V_\Delta$ then
22:         $H \leftarrow$ FilterByLowerBound($V_P$, $G$) /* Section 5.4 */
23:     end if
24: end while
```

Generality of the framework. The above framework is generic and extensible. Specifically, it can also be utilized to process exact subgraph search query, which is a special case of BPH query. Recall from Section 3.2, $e.upper$ and $e.lower$ of a new edge is set to 1 by default. Hence, if a user does not enter any value for these bounds (Line 3) for all edges of a BPH query, then it reduces to an exact subgraph search query. Boomer can be utilized to process such queries as the CAP index maintains all candidate matching vertices that satisfy the upper bound constraints (in this case $e.upper = 1$).

Our framework can also work with different types of GUIs. Observe that the actions in Lines 3, 9, and 11 are orthogonal to specific steps taken by a GUI to realize them. For instance, different visual graph query interfaces may follow different steps to add a vertex or an edge to a BPH query. Boomer is independent of these steps.

Lastly, our framework is amenable to different indexing and results generation schemes. Lines 7 and 14 can adopt different instantiations of online indexing schemes to support BPH queries. Similarly, Line 10 can adopt different modification policies. Line 22 also can be implemented using different results generation schemes.

5 CAP-BASED BLENDING OF QUERIES

In this section, we describe three strategies for constructing and maintaining the CAP (**C**ompact **A**daptive **P**ath) index, namely, *Immediate*, *Defer-to-Run*, and *Defer-to-Idle*.

5.1 CAP Index

The CAP index is an adaptive online data structure which efficiently stores the candidate matching vertices in G that satisfy the upper bound constraints of the edges in a (partially) constructed BPH query. This helps us to efficiently retrieve candidate matching vertices during query formulation and results generation as the distances between vertices in G do not have to be computed repeatedly. In addition, unsuitable candidates (*i.e.,* those which subsequently violate the upper bound constraints) can be pruned immediately.

Given a (partial) BPH query $Q_B = (V_B, E_B, L, \lambda)$, a matching order M, and a data graph $G = (V, E, L)$, intuitively a CAP index $\mathbb{C} = (V_C, E_C)$ is a $|V_B|$-level undirected graph containing vertices of V that match V_B, *i.e.,* $V_C \subseteq V$. Each edge in E_C connects a pair of matching vertices in two different levels of $\mathbb{C}$ if these vertices are connected by a path in G that satisfies the upper bound constraint specified in the corresponding edge in E_B. For example, consider the data graph G and the query graph Q_1 in Figure 2(a) with matching order $M : q_1 \rightarrow q_2 \rightarrow q_3$. The corresponding CAP index after the construction of Q_1 is shown in Figure 2(c). Observe that it has three levels as there are three vertices in Q_1. The matching vertices of q_1, q_2, and q_3 that satisfy the upper bound constraints of edges in Q_1 are $V_{q_1} = \{v_2, v_3\}$, $V_{q_2} = \{v_5, v_6, v_8\}$, and $V_{q_3} = \{v_{12}\}$, respectively. Observe that although v_1 matches q_1, it is pruned out after the formulation of (q_1, q_2) edge as the paths connecting vertices matching q_1 and q_2 do not satisfy the upper bound constraint. Hence, although v_1 is initially retrieved when q_1 is constructed by the user, it is subsequently removed from the index as soon as the corresponding upper bound constraint is violated. On the other hand, v_2 and v_{12} are connected by an edge in the index as the distance between (v_2, v_{12}) is 2, which satisfies the upper bound constraint of (q_1, q_3) in Q_1. For the same reason, (v_6, v_{12}) are connected in the index.

Definition 5.1. **[CAP Index]** *Given a data graph $G = (V, E, L)$ and a BPH query $Q_B = (V_B, E_B, L, \lambda)$ with matching order $M : q_1 \rightarrow \cdots \rightarrow q_{|V_B|}$, the **CAP index** is an undirected graph $\mathbb{C} = (V_C, E_C)$ where $V_C \subseteq V$ and satisfies the following conditions: (1) $V_C = V_{q_1} \bigcup \cdots \bigcup V_{q_{|V_B|}}$ where $\forall u \in V_{q_i}$, $L(u) = L(q_i)$. (2) $E_C = E_{e_1} \bigcup \cdots \bigcup E_{e_{|E_B|}}$ where $e_i \in E_B$ and $\forall (u, v) \in E_{e_i}$, there exists a path $p : u \rightarrow \cdots \rightarrow v$ in G such that $length(p) \leq e_i.upper$.*

We now introduce some auxiliary terminology. Given a matching order $M : q_1 \rightarrow \cdots \rightarrow q_{|V_B|}$ of a BPH query Q_B on a data graph G, consider a matching candidate vertex v of query vertex q_i (*i.e.,* $v \in V_{q_i}$) in $\mathbb{C}$. Note that $L(v) = L(q_i)$. An *adjacent indexed vertex set* (AIVS) of v, denoted as $V_{q_i}^{q_j}(v)$, refers to the set of candidate vertex matches X of query vertex $q_j \in Q_B$ that has an edge connecting to v in $\mathbb{C}$ such that $\forall u \in X$, there exists at least one path in G, between u and v, that satisfies $e_q.upper$ where $e_q = (q_i, q_j)$. For example, consider v_2 in G which is a matching vertex of q_1, *i.e.,* $v_2 \in V_{q_1}$. Then $V_{q_1}^{q_3}(v_2) = \{v_{12}\}$ and $V_{q_1}^{q_2}(v_2) = \{v_5\}$. Observe that $V_{q_1}^{q_3}(v_2) \subseteq V_{q_3}$ and $V_{q_1}^{q_2}(v_2) \subseteq V_{q_2}$. They imply that there exists at least one path $v_2 \rightarrow \cdots \rightarrow v_{12}$ whose length is in the range $[1, 3]$ and the path $v_2 \rightarrow v_5$ has length of exactly 1.

Next, we introduce the notion of *partial-matched vertex set* (denoted as V_P) in $\mathbb{C}$. Consider the connected subgraph consisting of v_2, v_5, and v_{12} in Figure 2(c). Observe that there is a 1-1 mapping between these vertices and the query vertices. Specifically, v_2 matches q_1, v_5 matches q_2, v_{12} matches q_3 and the edges (v_2, v_5), (v_5, v_{12}), and (v_2, v_{12}) in the CAP index satisfy the upper bounds of the corresponding edges in Q_1. We refer to these vertices as partial-matched vertex set. Note that at this point, we do not check the satisfaction of lower bound constraints of V_P. Such check is necessary only when $e.lower > 1$ and we defer it to the result subgraphs generation phase (Section 5.4). Note that when $e.lower = 1$, all results satisfying the upper bound constraints also satisfy the lower bound constraints. Observe that the number of vertices and edges in the subgraph are identical to those of the BPH query. Clearly, there can be many partial-matched vertex sets in a CAP index for a BPH query. We denote the set of all partial-matched vertex sets as $V_\Delta = \{V_{P_1}, \cdots, V_{P_{|V_\Delta|}}\}$. For example, consider Figure 2(c). For Q_1, $V_\Delta = \{\{v_2, v_5, v_{12}\}, \{v_3, v_6, v_{12}\}, \{v_3, v_8, v_{12}\}\}$ where v_2 and v_3 map to q_1; v_5, v_6 and v_8 map to q_2; and v_{12} maps to q_3.

LEMMA 5.2. *Given a data graph $G = (V, E, L)$ and a BPH query $Q_B = (V_B, E_B, L, \lambda)$, the space complexity of the CAP index is $O(\sum_{q_i \in V_B} |V_{q_i}| + \sum_{e(q_i,q_j) \in E_B} |V_{q_i}| \times |V_{q_j}|)$ in the worst case.*

Although the worst case space complexity of CAP index is quadratic in theory, it is smaller in practice due to the pruning of vertices that do not satisfy the upper bound constraints during query formulation. We shall demonstrate this in Section 7.

5.2 Immediate Construction (IC)

Intuitively, we can construct the CAP index during visual formulation of a BPH query by processing the query vertices and edges iteratively in the order they are constructed. We refer to this strategy as *immediate construction* (IC). Formally, let Q_B be a BPH query and a user follows a matching order $M : q_1 \to q_2 \to \cdots \to q_k$ to construct it visually. Correspondingly, she follows a specific sequence of edge construction $\mathcal{E} : e_1 \to e_2 \to \cdots \to e_j$ based on M to complete formulation of Q_B. For example, consider Q_1 in Figure 2(a) and the matching order $M : q_1 \to q_2 \to q_3$. Hence, a user may visually construct the following sequence of edges $\mathcal{E} : (q_1, q_2) \to (q_2, q_3) \to (q_3, q_1)$ to construct the query. Then in the IC strategy, we process the vertices and edges in the order they are constructed (*i.e.,* M and $\mathcal{E}$) by utilizing the GUI latency available during query formulation. For instance, in the above example, the GUI latency available during the visual construction of (q_2, q_3) is utilized to process the edge (q_1, q_2) and maintain the CAP index.

Algorithm 2 outlines this procedure. Briefly, for every vertex q_i that is added to the BPH query, it identifies candidate vertices V_{q_i} in the data graph having the same label as q_i (Lines 1- 4); and for every edge (q_i, q_j) in the query (Lines 7-8), it first identifies the AIVS of q_i and q_j ($V_{q_i}^{q_j}$ and $V_{q_j}^{q_i}$) that satisfy the upper bound constraint of (q_i, q_j) by issuing distance queries. Specifically, it first expands the partial CAP index by creating AIVS associated with q_i and q_j, respectively ($V_{q_i}^{q_j}$ and $V_{q_j}^{q_i}$) and initialize them as empty . Then, it populates these vertex sets by checking each vertex pair $(v_i, v_j) \in V_{q_i} \times V_{q_j}$ and adding v_j to $V_{q_i}^{q_j}(v_i)$ and v_i to $V_{q_j}^{q_i}(v_j)$ if and only if $dist(v_i, v_j) \leq upper$. We refer to this procedure for populating vertex sets as POPULATEVERTEXSET (denoted as PVS). Observe that this procedure can be expensive when the upper bound constraint is relatively large (≥ 3). Hence, we adopt three different strategies, namely *neighbor search*, *2-hop search* and *large upper search*, to efficiently populate the AIVS. Intuitively, for a new edge $e_q = (q_i, q_j)$ we first retrieve the candidate matching vertices V_{q_i} and V_{q_j}. If $e_q.upper$ is 1 or 2 then the *neighbor search* or *2-hop search* is used. Otherwise, the *large upper search* scheme is utilized.

Neighbor Search. When the upper bound of (q_i, q_j) is 1, the AIVS can be found by either (a) performing a scan of the neighbours of each $v \in V_{q_i}$ to identify all vertices occurring in V_{q_j} (referred to as *out-scan*) or (b) by scanning V_{q_j} to identify all vertices adjacent to v (referred to as *in-scan*). The cost of such a scan can be expensive for the former if v has many neighbours (denoted as $Cost_{out}$) and if V_{q_j} contains many vertices that are not adjacent to v in the latter (denoted as $Cost_{in}$). Specifically, $Cost_{out}$ is computed by performing a linear scan of v_i's neighbour and for each neighbour vertex v_j having the same label as q_j, it checks for its existence in V_{q_j}, leading to a cost of $O(deg(v_i) + deg(v_i) \times p_{L(q_j)} \times log(|V_{q_j}|))$ (Lemma 5.3) where $p_{L(q_j)}$ is the probability of a vertex in G labelled as $L(q_j)$. In comparison, $Cost_{in}$ is computed by performing linear scan on V_{q_j} and checking iteratively if the vertex is adjacent to v_i, resulting in $Cost_{in} = O(|V_{q_j}| \times log(deg(v_i)))$ (Lemma 5.3). When $|V_{q_j}| \gg deg(v_i)$, $Cost_{in} > Cost_{out}$ and BOOMER performs an out-scan. Otherwise, an in-scan is performed.

Algorithm 2 Algorithm *ConstructCAP* (Immediate construction)

```
Require: GUI action $a$, action stream $stream$, $\mathbb{C} = (V_C, E_C)$, $G = (V, E, L)$, (partial) BPH
    query $Q_B = (V_B, E_B, L, \lambda)$, PML index $\mathbb{P}$;
Ensure: CAP index $\mathbb{C}$;
1: if IsNewVertex($a$) = $true$ then
2:     $q_i \leftarrow$ GetNewVertex($a$)
3:     $V_{q_i} \leftarrow \{v : v \in V, L(v) = L(q_i)\}$
4:     $\mathbb{C} \leftarrow$ UpdateCAPVertex($\mathbb{C}$, $V_{q_i}$)
5: else
6:     $(q_i, q_j)$, $upper \leftarrow$ GetNewEdge($a$)
7:     $\mathbb{C} \leftarrow$ ProcessEdge($\mathbb{C}$, $(q_i, q_j)$, $upper$, $G$, $Q_B$, $\mathbb{P}$)
8: end if
```

LEMMA 5.3. *Given a query edge (q_i, q_j), a vertex $v_i \in V_{q_i}$ and a candidate vertex set V_{q_j}, the costs of identifying the AIVS (based on neighbor search strategy) using out-scan and in-scan are $O(deg(v_i) + deg(v_i) \times p_{L(q_j)} \times log(|V_{q_j}|))$ and $O(|V_{q_j}| \times log(deg(v_i)))$, respectively, in the worst case.*

Two-hop Search. When $e_q.upper = 2$, BOOMER examines the 2-hop neighbourhood of v_i. In particular, the out-scan and in-scan cost in the above algorithm are replaced with $Cost_{out} = TwoHop(v_i) + TwoHop(v_i) \times p_{L(q_j)} \times log(|V_{q_j}|)$ and $Cost_{in} = |V_{q_j}| \times (deg(v_i) + deg(v_j))$, respectively (Lemma 5.4). Recall from the preceding section we pre-compute the 2-hop neighbourhood of each vertex in G. Note that we only record the count and not the exact vertex set since the edge processing cost is estimated based on the number of 2-hop neighbours. This results in efficient use of space.

LEMMA 5.4. *The costs for identifying the AIVS (based on two-hop search strategy) using out-scan and in-scan are $O(TwoHop(v_i) + TwoHop(v_i) \times p_{L(q_j)} \times log(|V_{q_j}|))$ and $O(|V_{q_j}| \times (deg(v_i) + deg(v_j)))$, respectively, in the worst case.*

Large Upper Search. For larger upper bounds, adopting the aforementioned strategies would involve scanning significantly larger number of vertices, rendering it impractical. Hence, BOOMER exploits the PML index to process the distance queries when $upper \geq 3$. Specifically, it scans V_{q_i} and V_{q_j} and for each pair of vertices $(v_i, v_j) \in V_{q_i} \times V_{q_j}$, v_i and v_j are added to $V_{q_j}^{q_i}(v_j)$ and $V_{q_i}^{q_j}(v_i)$, respectively, if $dist(v_i, v_j) \leq upper$.

LEMMA 5.5. *The cost of identifying AIVS when $upper \geq 3$ is $O(|V_{q_i}||V_{q_j}| \times (|C(v_i)| + |C(v_j)|))$ in the worst case, where $C(v_i)$ is the distance-aware 2-hop cover of v_i.*

In practice, even with the usage of PML, PVS can still be costly especially when $|V_{q_i}|$ or $|V_{q_j}|$ is very large. BOOMER *prunes* all *isolated* vertices whose adjacent candidate vertex set is null (*i.e.,* $V_{q_i}^{q_j}(v_i) = \phi$ and $V_{q_j}^{q_i}(v_j) = \phi$). Note that removal of a vertex v affects the candidate vertex set of neighbouring vertex sets (*i.e.,* $V_{q_{i-1}}$ and $V_{q_{i+1}}$) since v may be the sole vertex in the candidate vertex set. Hence, removing it may result in an empty candidate vertex set for

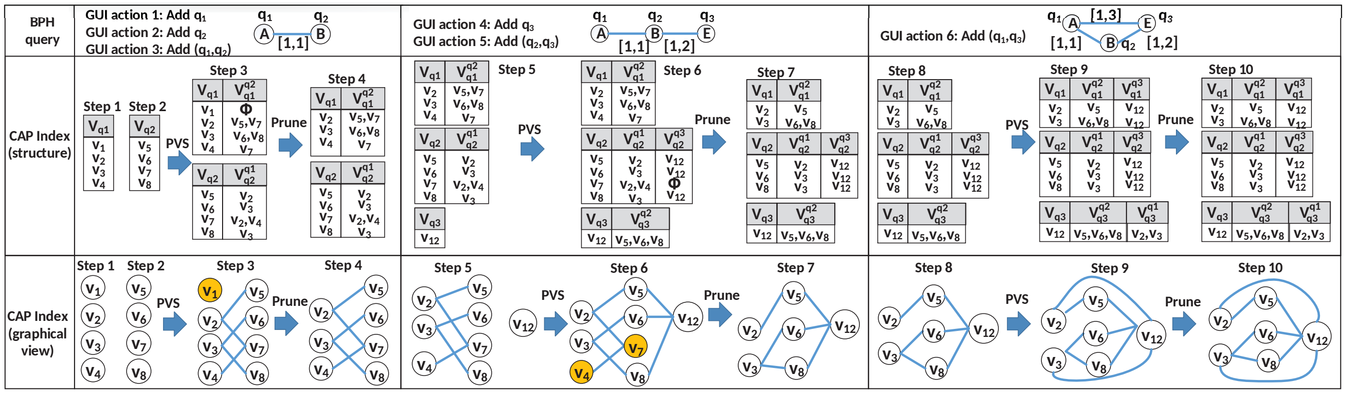

Figure 3: Example of CAP index construction (PVS stands for *PopulateVertexSet* procedure).

$V_{q_{i-1}}$ or $V_{q_{i+1}}$. In order to address this, Boomer performs pruning recursively by checking the candidate vertex set of neighbouring vertex sets for every vertex v removed.

Lemma 5.6. *The maximum number of pruning steps to be performed is* $\sum_{q_i \in V_B} |\{v | L(v) = L(q_i), \forall v \in V\}|$.

Example 5.7. Consider Q_1 on G in Figure 2. Figure 3 illustrates the steps for constructing the CAP index as Q_1 is visually formulated. In Steps 1 and 2, q_1 and q_2 are drawn and the CAP index consists of $V_{q_1} = \{v_1, v_2, v_3, v_4\}$ and $V_{q_2} = \{v_5, v_6, v_7, v_8\}$. After the edge $e_1 = (q_1, q_2)$ is drawn (Step 3), $V_{q_1}^{q_2}$ and $V_{q_2}^{q_1}$ are populated by the *PopulateVertexSet* (PVS) procedure using the *neighbor search* strategy since $e_1.upper = 1$. In Step 4, it prunes isolated vertices (*i.e.*, v_1). Next, when the query vertex q_3 is added to Q_1 (Step 5), $V_{q_3} = \{v_{12}\}$ is added to the index. In Step 6, when the edge $e_2 = (q_2, q_3)$ is drawn, $V_{q_2}^{q_3}$ and $V_{q_3}^{q_2}$ are populated by the *two-hop search* strategy in PVS since $e_2.upper = 2$. It prunes isolated vertices v_4 and v_7 in Step 7. In Step 8, the last edge $e_3 = (q_1, q_3)$ is constructed and $V_{q_1}^{q_3}$ and $V_{q_3}^{q_1}$ are populated by the *large upper search* strategy in PVS since $e_3.upper = 3$ (Step 9). There are no isolated vertices identified in Step 10. Hence, no pruning is performed.

Remark. Several k-neighborhood indexes have been utilized by existing subgraph matching techniques [4, 18, 36]. [4, 18] utilize 1-hop indexes and hence cannot be efficiently used in our framework for higher upper bounds. SPath [36] uses the k-neighborhood by maintaining for each vertex u in the data graph a structure that contains the labels of all vertices that are at a distance less or equal to k from u. Consequently, it may store a large portion of the entire data graph for larger k. This makes it prohibitively expensive to utilize in our framework as demonstrated in [29]. In contrast, the CAP index is lightweight in practice and is created on-the-fly during query construction only for candidate matches of the query vertices.

More importantly, these approaches exploit knowledge of the *complete* query graph to compute auxiliary data structures and an effective matching order. Although this strategy is reasonable in the traditional querying paradigm, it is infeasible in our visual querying paradigm as construction and processing of query components are interleaved. Hence, CAP index construction needs to proceed with *partial* knowledge of the BPH query. Besides, it is impractical to demand a user to formulate a BPH query by following the most effective matching order.

5.3 Deferment-based Construction (DC)

Observe that the IC strategy is effective if the sequence of formulated edges $\mathcal{E}$ is iteratively processed within the available GUI *latency*. However, as remarked earlier, processing of an edge $e_q = (q_i, q_j)$ can be expensive if $|V_{q_i}|$ and $|V_{q_j}|$ are very large and $e_q.upper$ is large as well (Lemma 5.5). Consequently, when a query involves such *expensive* edges, the CAP index construction may take significantly longer time than the available GUI *latency*. For example, consider the query in Example 5.7. Suppose instead of formulating the edge (q_1, q_3) at the end, a user formulates it first (*e.g.*, $\mathcal{E} : (q_1, q_3) \rightarrow (q_2, q_3) \rightarrow (q_1, q_2)$). Assume that $|V_{q_1}|$ and $|V_{q_3}|$ are large. Then, (q_1, q_3) may be an expensive edge and it may take considerably more time than the available GUI latency to construct the index. Observe that this has a cascading impact on the subsequent construction and maintenance of the CAP index as the processing of subsequent edges are delayed even though (q_2, q_3) and (q_1, q_2) can be efficiently processed, leading to larger SRT. On the other hand, when (q_1, q_3) is formulated at the end (e.g., Example 5.7), the sizes of matching vertices of q_1 and q_3 may reduce significantly due to the removal of isolated vertices in prior steps. Furthermore, as it is formulated at the end, it does not delay processing of other edges in the query. *How can we ensure that such expensive edges do not adversely impact the blending of* BPH *query formulation and processing?* We present a novel *deferment-based construction* (DC) strategy to address it.

We need to first determine whether a newly-constructed edge is potentially *expensive*. Second, if an expensive edge is not processed immediately, we need to have a strategy to determine *when* it should be subsequently processed. Let T_{est} be the estimated time to process an edge e in Q_B. Then, e is an *expensive* edge if:

$$T_{est} > t_{lat} \tag{1}$$

where t_{lat} is the *minimum* GUI *latency time* available for processing e. Hence, to determine whether an edge is expensive we need to address the following two questions: (a) How do we estimate T_{est}? and (b) How do we estimate t_{lat}? We now elaborate on them.

Estimation of T_{est}. It is worth noting that the cost of processing an edge in a BPH query depends on the topology of the underlying data graph, which may vary widely. Our estimation of T_{est} must be performed fast so that the GUI latency is not wasted for this activity. Recall from Section 4, we have already computed the average edge

Algorithm 3 Algorithm *ConstructCAP* (Defer-to-Run)

```
Require: GUI action $a$, action stream $stream$, $\mathbb{C} = (V_C, E_C)$, $G = (V, E, L)$, $Q_B = (V_B, E_B, L, \lambda)$, edge pool $pool$, average edge processing time $t_{avg}$, $\mathbb{P}$;
Ensure: $\mathbb{C}$, edge pool $pool$;
1: if IsNewVertex($a$) = $true$ then
2:     $q_i \leftarrow$ GetNewVertex($a$)
3:     $V_{q_i} \leftarrow \{v : v \in V, L(v) = L(q_i)\}$
4:     $\mathbb{C} \leftarrow$ UpdateCAPVertex($\mathbb{C}$, $V_{q_i}$)
5: else if IsNewEdge($a$) = $true$ then
6:     $(q_i, q_j)$, $upper \leftarrow$ GetNewEdge($a$)
7:     if $upper \leq 2$ or $|V_{q_i}| \times |V_{q_j}| \times t_{avg} \leq t_{lat}$ then
8:         $\mathbb{C} \leftarrow$ ProcessEdge($\mathbb{C}$, $(q_i, q_j)$, $upper$, $G$, $Q_B$, $\mathbb{P}$)
9:     else
10:        $pool \leftarrow$ Insert($pool$, $(q_i, q_j)$)
11:    end if
12: else
13:    while $pool \neq \emptyset$ do
14:        $(q_i, q_j)$, $upper \leftarrow$ GetMinEdge($pool$)
15:        $\mathbb{C} \leftarrow$ ProcessEdge($\mathbb{C}$, $(q_i, q_j)$, $upper$, $G$, $Q_B$, $\mathbb{P}$)
16:        $pool \leftarrow$ Remove($pool$, $(q_i, q_j)$)
17:    end while
18: end if
```

processing time t_{avg} in a given data graph G empirically while preprocessing it. Hence, we utilize this to compute T_{est} of a query edge $e = (q_i, q_j)$ as follows: $T_{est} = |V_{q_i}| \times |V_{q_j}| \times t_{avg}$.

Estimation of t_{lat}. Let q_i be the constructed query graph fragment at i-th step during visual query formulation using the class of GUI described in Section 3.2. Note that q_i is processed (including construction and maintenance of the CAP index) by utilizing the GUI latency available during the successor step $i + 1$. Specifically, the successor step can be (a) construction of a query vertex or (b) construction of an edge connecting an existing vertex pair. Let us denote the time taken to perform these two actions as T_{node} and T_{edge}, respectively. Hence, $t_{lat} = min(T_{node}, T_{edge})$. In order to add a vertex, a user may undertake Steps 1-3 described in Section 3.2. Let us refer to the time taken to complete Steps 1, 2, and 3 as *movement time* (denoted by t_m), *selection time* (t_s), and *drag time* (t_d), respectively. Then $T_{node} = t_m + t_s + t_d$. On the other hand, construction of an edge involves Steps 5 and 6. Let us refer to the time taken to complete these steps as *edge construction time* (t_e) and *bound formulation time* (t_b), respectively. Then $T_{edge} = t_e + t_b$. Note that in the case $e.upper = 1$ (default value), $t_b = 0$ (the default bounds on an edge can be set to [1, 1]). Consequently, minimum value of T_{edge} is t_e. Then, the *minimum* GUI latency time available is $min(t_m + t_s + t_d, t_e)$. Since $(t_m + t_s + t_d) > t_e$ [3], $t_{lat} = t_e$.

Consequently, Equation 1 can be rewritten as follows.

$$|V_{q_i}| \times |V_{q_j}| \times t_{avg} > t_e \tag{2}$$

In summary, we can use Equation 2 to determine efficiently whether a query edge $e = (q_i, q_j)$ is expensive to process.

Definition 5.8. **[Expensive Edge]** *Given a BPH query $Q_B = (V_B, E_B, L, \lambda)$ on data graph G, an edge $e = (q_i, q_j)$, $e \in E_B$ is expensive if $|V_{q_i}| \times |V_{q_j}| \times t_{avg} > t_e$ and $e.upper \geq 3$.*

Next, we describe two strategies for deferring processing of expensive edges, namely *Defer-to-Run* and *Defer-to-Idle*.

Defer-to-Run Strategy. This strategy defers processing of expensive edges until the Run icon is clicked. The intuition is that by efficiently processing all inexpensive edges early by utilizing the GUI latency, we can reduce the sizes of V_{q_i} and V_{q_j} by removing isolated vertices, leading to more efficient processing of expensive edges. Furthermore, since the expensive edges are processed after

Algorithm 4 Algorithm *ConstructCAP* (Defer-to-Idle)

```
Require: GUI action $a$, action stream $stream$, $\mathbb{C} = (V_C, E_C)$, $G = (V, E, L)$, $Q_B = (V_B, E_B, L, \lambda)$, edge pool $pool$, $t_{avg}$, $\mathbb{P}$;
Ensure: $\mathbb{C}$, edge pool $pool$;
1: $t_{idle} \leftarrow t_{lat}$
2: if IsNewVertex($a$) = $true$ then
3:     Lines 2-4 of Algorithm 3
4:     $t_{idle} \leftarrow$ UpdateIdleTime($t_{idle}$)
5:     $\mathbb{C}$, $pool \leftarrow$ ProbePool($stream$, $\mathbb{C}$, $pool$, $G$, $Q_B$, $t_{avg}$, $\mathbb{P}$, $t_{idle}$)
6: else if IsNewEdge($a$) = $true$ then
7:     Lines 6-11 of Algorithm 3
8:     $t_{idle} \leftarrow$ UpdateIdleTime($t_{idle}$)
9:     $\mathbb{C}$, $pool \leftarrow$ ProbePool($stream$, $\mathbb{C}$, $pool$, $G$, $Q_B$, $t_{avg}$, $\mathbb{P}$, $t_{idle}$)
10: else
11:    Lines 13-17 of Algorithm 3
12: end if
```

the formulation of complete query, this strategy prevents unnecessary delay in evaluating inexpensive edges.

Algorithm 3 describes the *Defer-to-Run* strategy. An *edge pool* (denoted by *pool*) is used to keep track of expensive edges that have not been processed yet. We use a priority queue to implement it. Then, when a new edge (q_i, q_j) is added, it is added to *pool* if $upper \geq 3$ and $|V_{q_i}| \times |V_{q_j}| \times t_{avg} > t_e$ (Line 10). Finally, after the Run icon is clicked, it processes the edges in the pool iteratively. In each iteration, the least expensive edge is removed from *pool* and processed using the procedure *ProcessEdge* (Lines 13 to 17).

Note that even though the entire query graph is connected, the order in which expensive edges in the *pool* is processed may result in an intermediate structure of a CAP index comprised of multiple connected components where edges in each component are *only* processed edges of the query. As edges are processed, these components merge and eventually become a single connected component when the index construction is completed. In Section 6, we shall discuss how this feature is utilized to handle query modification.

Defer-to-Idle Strategy. Observe that in *Defer-to-Run* strategy *all* expensive edges are processed after clicking of the Run icon. However, expensive edges may reduce to inexpensive ones with the addition of new query edges during query formulation as candidate vertices V_{q_i} and their related AIVs may reduce significantly in size. Let us elaborate further. Consider a sequence of edges $\mathcal{E} : e_1 = (q_1, q_3) \rightarrow e_2 = (q_2, q_3) \rightarrow e_3 = (q_1, q_2)$ constructed by a user during query formulation. Assume that e_1 is an expensive edge. Then according to the *Defer-to-Run* strategy, it will be processed after the entire query is formulated. Now assume that e_2 is an inexpensive edge and takes $t' \ll t_{lat}$ time. That is, it takes significantly less time than the available GUI latency to process e_2. Further, it prunes many isolated vertices in V_{q_3}. Consequently, our query blender framework is now "idle" as there is no new vertices or edges to process and existing inexpensive edges have already been processed. Now assume that after processing e_2, e_1 is no more an expensive edge as the size of V_{q_3} has reduced significantly after the removal of isolated vertices. In particular, $|V_{q_1}| \times |V_{q_3}| \times t_{avg} < t_{lat} - t'$. Then, it is possible to process e_1 now by leveraging on the idle time instead of waiting for the complete query to be formulated. Clearly, this will reduce the number of expensive edges waiting to be processed after the Run icon is clicked, leading to superior SRT. We refer to this as *Defer-to-Idle* strategy.

Algorithm 4 outlines the *Defer-to-Idle* strategy. Instead of processing edges in *pool* after Run icon is clicked, it *probes* the pool after processing a new vertex (Line 5) or a new edge (Line 9). The

UpdateIdleTime procedure (Lines 4, 8) keeps track of the time available for processing an edge in *pool* by subtracting the processing time of current edge/vertex from the available GUI latency (*i.e.,* $t_{lat} - t'$). During pool-probing, it checks if a new GUI action has joined the *stream* and terminates the probing to handle any new action. Prioritizing new GUI action processing over edge processing in *pool* allows us to further reduce the size of V_{q_i} and the AIVS where applicable, reducing T_{est} of edges in *pool*. Suppose *stream* has no new action. It selects the topmost edge for processing if its $T_{est} \leq t_{lat} - t'$. Otherwise, it terminates the pool-probing to await the next GUI action since edges in *pool* are still too expensive to be processed.

Algorithm 5 Procedure *Deletion*

```
Require: Action stream $stream$, $\mathbb{C} = (V_C, E_C)$, deleted query edge $e_q = (q_i, q_j)$, edge pool
    $pool$, $G = (V, E, L)$, $Q_B = (V_B, E_B, L, \lambda)$, $t_{avg}$;
Ensure: (Partial) CAP index $\mathbb{C} = (V_C, E_C)$, edge pool $pool$;
 1: $Q_c = (V_c, E_c) \leftarrow$ GetConnectedComponent($Q_B, \mathbb{C}, q_i, q_j$)
 2: for vertex $q_k \in Q_c$ do
 3:     $V_{q_k} \leftarrow \emptyset$
 4:     RemoveAIVS($q_k$)
 5:     for vertex $\{v | L(v) = L(q_k)\} \in G$ do
 6:         $V_{q_k} \leftarrow V_{q_k} \bigcup \{v\}$
 7:     end for
 8: end for
 9: for edge $e_k \in E_c \setminus \{e_q\}$ do
10:     $pool \leftarrow pool \bigcup \{e_k\}$
11: end for
12: $\mathbb{C}, pool \leftarrow$ ProbePool($stream, \mathbb{C}, pool, G, Q_B, t_{avg}, t_{idle}$)
```

5.4 Generation of Result Subgraphs

Once the complete query is formulated, a user may click the Run icon to execute it. This triggers the evaluation of unprocessed edges (if any) and completion of the CAP index construction. Then, we can utilize it to generate and visualize the matching results (*i.e.,* result subgraphs) satisfying the BPH query. We describe this step now.

Upper bound-constrained matching results. Recall that a partial-matched vertex set V_P in a CAP index represents vertices of a matching result of the query that satisfy the upper bound constraints. Hence, first we traverse $\mathbb{C}$ to extract the set of partial-matched vertex sets V_Δ. The original matching order M is first *reordered* in increasing order of $|V_{q_i}|$ to ensure efficient traversal of $\mathbb{C}$. Next, it identifies V_P by traversing the CAP index using *depth-first-search* (DFS) starting from the first query vertex q_1 in M. In particular, it retrieves the set of candidates vertices $V_{candidate}$ for matching the next query vertex q_{next} by identifying vertices in query matched vertices $Q_{matched}$ that are adjacent to q_{next}. A partial-matched vertex set V_P is successfully obtained and added to V_Δ when all the query vertices have been matched. In the event that a query vertex q_{next} is not matched, it backtracks and traverses the next branch of the CAP index. The matching process terminates when CAP index traversal is completed. The time complexity of this step is $O(|V_C| + |E_C|)$.

Visualization-friendly, just-In-time lower bound checking. In the final step, the results (result subgraphs) of a BPH query is generated and visualized on the *Results Panel* of the GUI. Specifically, first we take a *visualization-friendly* approach. As remarked earlier, it is challenging to visualize result subgraphs (ranked or otherwise) on a large data graph as the node-link diagram of the underlying data graph looks like a "hairball" even when it contains a few hundred vertices. In fact, Ware and Mitchell reported that it is visually challenging to comprehend graphs containing tens of vertices in 2D [31]. Hence, the traditional approach of result subgraph visualization on the underlying data graph or on the *maximum match* [14][6] is not a palatable solution. In Boomer, we deploy *small region-based* visualization scheme of matching results where we only iteratively show a small subgraph of the underlying data graph containing a result match. That is, a user may iterate through V_Δ and for each $V_P \in V_\Delta$ we show a small subgraph of G containing it by color coding the result subgraph. Second, we take a *just-in-time* approach to evaluate the lower bound constraints and to generate the result subgraph involving V_P. Given a V_P, we check whether there exists a path satisfying the edge bounds by utilizing the PML index (Section 4). Briefly, Boomer finds a shortest path of a given query edge that satisfies the lower bound constraint (*i.e.,* $dist(v_i, v_j) \geq e.lower$). If $dist(v_i, v_j) < e.lower$, it "detours" to a neighbour of v_i (v_p) to increase the path length between v_i and v_j and then attempts to use a shortest path from v_p to v_j if $dist(v_p, v_j) + d \geq e.lower$ where d is the number of detours taken. Note that if $e.lower = 1$, we do not need to perform any checking as $e.lower \leq e.upper$.

For example, reconsider Figure 2. Consider $V_P = \{v_3, v_8, v_{12}\}$ in Figure 2(c). Boomer first checks the shortest path between v_3 and v_8 in G. Since $dist(v_3, v_8) \geq 1$, the shortest path is selected. For the second and third edges, the shortest paths are also greater than the lower edge bounds (*i.e.,* $dist(v_8, v_{12}) \geq 1$ and $dist(v_{12}, v_3) \geq 1$, respectively). Hence, the shortest paths for these edges are also selected. The final visualized result subgraph is highlighted in grey in Figure 2(b). Note that if the edge bound of (q_1, q_3) is modified to [3,3], then Boomer needs to take a "detour" to v_6 from v_3 instead of taking the shortest path ($v_3 \rightarrow v_8 \rightarrow v_{12}$). At v_6, it then checks if $dist(v_6, v_{12}) + 1 \geq 3$. If so, the shortest path from v_6 to v_{12} is taken. Otherwise, additional detours are required. The time complexity for the lower bound checking is $O(|E_B| \times \theta_{max} \times \mathbb{S})$ where θ_{max} is the maximum degree of a vertex in G and $\mathbb{S}$ is the time complexity of shortest path computation for a given pair of vertices.

6 QUERY MODIFICATION

In this section, we examine how a CAP index is maintained when we allow *query modifications.* In particular, we consider the following types of modifications in our setting: (a) deletion of an existing edge and (b) alteration of bounds of an existing query edge. We shall first discuss our technique in the context of *Defer-to-Idle* strategy and then highlight how it can be extended to other strategies. An example of query modification is given in Appendix C.

Deletion of an edge. Deletion of an edge is handled as follows depending on the feature of the edge. For an unprocessed edge, no change is required on the CAP index and Boomer simply removes the unprocessed edge from the edge pool. Algorithm 5 describes deletion of a processed edge. Briefly, it performs a "rollback" by reconstructing the *affected* region (*i.e.,* connected component) in the CAP index. The *GetConnectedComponent* procedure in Algorithm 5 identifies this connected component (Line 1). Note that only processed edges that are connected to the modified edge in the CAP index are affected. The reconstruction steps involve: (a)

[6]The *maximum match* S_M captures all vertices that match a pattern query P (with V_p vertices) in a network G. Since $|S_M| \leq |V||V_P|$ [14], it is visually challenging to comprehend matches in S_M.

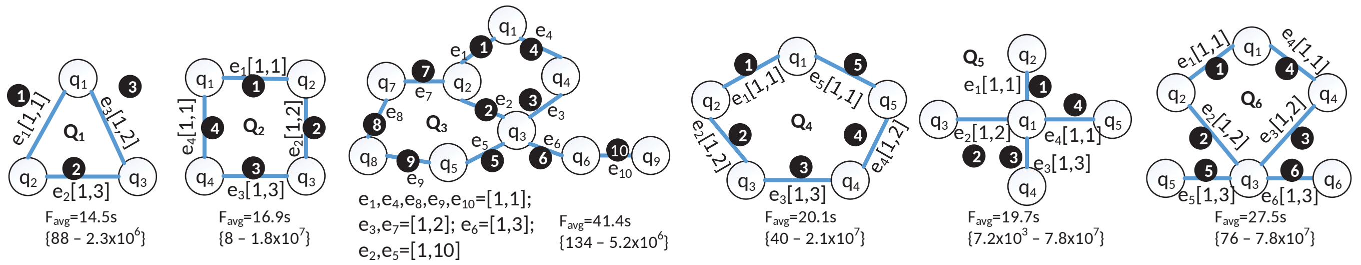

Figure 4: Template BPH queries. The default order for edge construction is given by the number in the filled circle (*e.g.*, e_1 is the first edge processed in Q_1). The default bounds are shown as labels on the edges. F_{avg} denotes the average QFT. The values in curly braces show the min and max result size of all query instances across all datasets for a given template query.

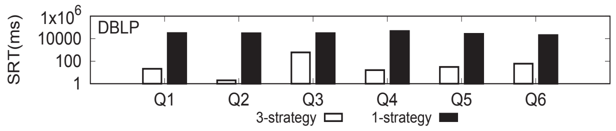

Figure 5: 3-Strategy vs 1-Strategy for IC.

Removal of V_{q_i} (Line 3) and AIVS (Line 4) of all *affected* vertices; (b) Retrieval of the relevant V_{q_i} (Lines 5 to 7); and (c) Addition of the affected edges (except deleted edge) to the edge pool. Observe that the *ProbePool* procedure ensures that where possible, edges in the pool are being handled.

Alteration of lower bound. Recall from Section 5.1 that the CAP index construction during query formulation considers only the upper bound. Hence, when the lower bound is altered, there is no change in the structure of a CAP index.

Alteration of upper bound. Upper bound modification can be made to either a processed or an unprocessed edge e and the modification can either tighten or loosen the bound. For clarity, we denote the original and new upper bound as $e.upper$ and $e.upper'$, respectively. When e is unprocessed, BOOMER updates the bound of the unprocessed edge in the edge pool. Note that no modification is required on the CAP index. On the other hand, if e is processed and the upper bound is modified to be stricter (*i.e.*, $e.upper > e.upper'$), a vertex $v_j \in V_{q_i}^{q_j}(v_i)$ may not satisfy the constraint $dist(v_i, v_j) \leq e.upper'$. Hence, we have to examine every pair of $(v_i, v_j) \in V_{q_i} \times V_{q_j}$ to reassess whether the upper bound constraint is satisfied and remove those that violate the constraint. Observe that removal of (v_i, v_j) may result in some vertices becoming isolated. Hence, we need to perform pruning steps subsequently to remove these isolated vertices (if any).

Conversely, when the upper bound is loosened (*i.e.*, $e.upper < e.upper'$), a vertex v_j that satisfies the upper bound constraint $dist(v_i, v_j) \leq e.upper'$ may not be in the existing $V_{q_i}^{q_j}(v_i)$. For a processed edge, modification to looser bounds is handled very similarly to deletion (Algorithm 5). The only difference is that the edge with loosened bound will be added back to the connected component (*i.e.*, $E_c \setminus \{e_q\}$ in Line 9 is replaced with E_c). Note that unlike the case of tighter upper bound where the edges are processed immediately, for processed edges, we have to add them in the edge pool as there is insufficient GUI latency to process them.

Handling Immediate and Defer-to-Run strategies. For CAP index construction based on IC strategy, no edge is deferred for processing. Hence, the corresponding query of the constructed CAP index is a single connected component. Consequently, handling

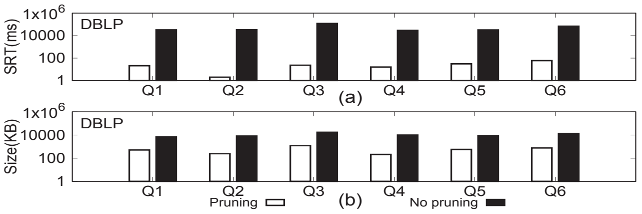

Figure 6: Effect of pruning on (a) SRT and (b) CAP index size.

of the modification of the upper bound corresponds to the processed edge with stricter and looser bound whereas edge deletion corresponds to that of deletion of a processed edge. For *Defer-to-Run* approach, expensive edges are processed only after Run icon is clicked. Hence, similar to *Defer-to-Idle*, edges can be processed or unprocessed and query modification follows the approach of *Defer-to-Idle*.

LEMMA 6.1. *Given $G = (V, E, L)$ and $Q_B = (V_B, E_B, L, \lambda)$, the time complexity for query modification in the worst case is $O(\sum_{q_i \in V_B} |V_{q_i}| + \sum_{e(q_i,q_j) \in E_{ne}} |V_{q_i}| \times |V_{q_j}| + n)$ where $q_i, q_j \in V_B$, E_{ne} is the set of inexpensive edges in the pool that can complete processing within idle time and n is the number of pruning steps.*

7 PERFORMANCE STUDY

We have implemented BOOMER in Java with JDK1.7. The construction of PML index [1] is realized in C++. In this section, we investigate the performance of BOOMER and report the key results. All experiments are performed on a 64-bit Windows desktop with Intel Xeon CPU E5-1630 (3.70GHz) and 32GB of main memory.

7.1 Experimental Setup

Datasets. We use the following three real datasets. (a) The *WordNet* dataset ($|V| = 82K$, $|E| = 125K$) consists of nouns, verbs, adjectives and adverbs of the English language that is grouped into cognitive synonyms differentiating various concepts. In particular, the synonyms sets are given as noun, verb, adjective, adjective satellite and adverb which are represented as character code n, v, a, s and r, respectively. We use these character codes as labels. (b) The *DBLP* dataset ($|V| = 317K$, $|E| = 1.1M$) is a DBLP co-authorship network where vertices represent authors and two vertices are connected if the authors publish at least one paper together. We generate 100 labels and randomly assign each vertex to a label. (c) The *Flickr* dataset ($|V| = 1.8M$, $|E| = 23M$) describes *relationship* between images from *Flickr* where vertices represent *Flickr* images. We generate 3000 labels and randomly assign each vertex to a label.

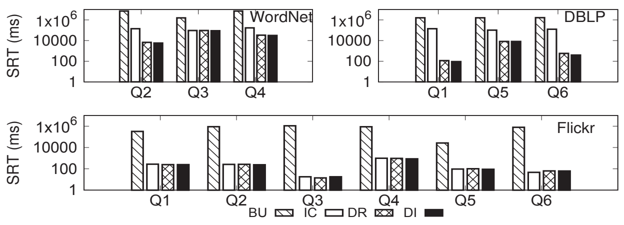

Figure 7: Avg. SRT for BU, IC, DR, and DI.

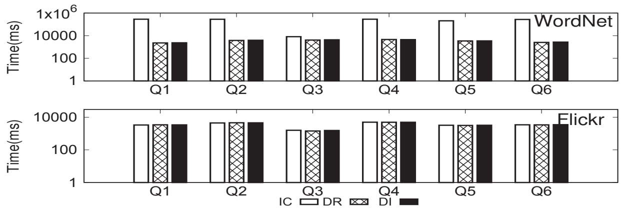

Figure 8: Avg. CAP construction time for IC, DR, and DI.

Algorithms. To the best of our knowledge, there is no existing framework that can evaluate BPH queries using the proposed visual querying paradigm. Note that the work in [20] focuses on substructure search queries on large networks and not on p-hom queries. Specifically, as remarked in Section 1, the index structure in [20] can only support edge-to-edge mapping but not edge-to-path as demanded by BPH queries. On the other hand, as remarked earlier the seminal work in [13] is based on traditional setting which is orthogonal to this visual querying paradigm. Furthermore, they do not seek to find *all* matches to a 1-1 p-hom query (*i.e.,* given a pair of graphs, the goal is to find a p-hom mapping). Hence, we develop the following baseline referred to as BOOMER-*unaware evaluation* (BU). It generates partial matches without utilizing the CAP index after the Run icon is clicked by following the reordered matching order (Section 5.4). For each new vertex q_j and associated edge added to the query, it checks whether upper bound constraints are satisfied using the PML index. If the condition is satisfied, q_j is joined with previously matched portion to form the new partial match. Alternatively, if a new edge is drawn to connect existing query vertices, then it checks each partial match for upper bound constraint and prunes away those that fail the check. Note that BU represents evaluation of BPH query without using the BOOMER framework. In the sequel, we compare it with the *Immediate*, *Defer-to-Run*, and *Defer-to-Idle* strategies, denoted as IC, DR, and DI, respectively.

Query set: It is impractical to generate a large number of visual BPH queries for experimental study as they are formulated by real users. In fact, visual formulation of too many queries strongly deters end users to participate in the study. Furthermore, there is no benchmark queryset for BPH queries. However, in practice, graph pattern queries are often small; 56.5% of real-life SPARQL queries consists of a single edge (RDF triple) whereas 90.8% uses at most 6 edges [5]. Hence, we select small-sized *template* BPH queries (Figure 4) whose topology can be found in real-life queries such as cycles (Q_1,Q_2,Q_4), stars (Q_5), and flowers (Q_3,$Q6$) [5]. All these templates are formulated on *each* dataset (*i.e.,* 18 queries) by modifying the vertex labels. We also modified the bounds associated with them to generate new queries in order to evaluate our framework from various aspects. *In total, 103 unique* BPH *queries are generated on the three datasets.*

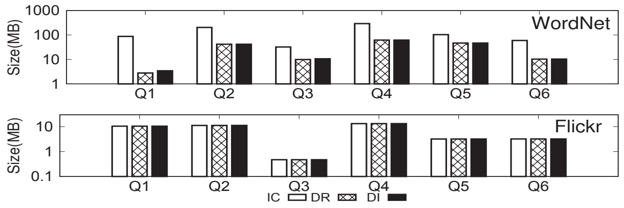

Figure 9: Avg. CAP index size for IC, DR, and DI.

In the sequel, unless mentioned otherwise, we follow the default bounds and edge formulation orders mentioned in Figure 4.

Participants profile. Twenty unpaid volunteers (ages from 21 to 27) participated in the experiments in accordance to HCI research that recommends at least 10 users [16, 23]. None of them are authors of this paper. They were first trained to use the GUI of BOOMER. For every query, the participants were given some time to determine the steps that are needed to formulate the query visually. Note that the faster a user formulates a query, the lesser time BOOMER has for CAP construction. Each query was formulated four times by four different participants selected. Hence, a total of 412 queries (103×4) are executed. Each participant formulated on average 20.6 queries. The average query formulation time (QFT) for each template query (with different bounds) by all participants is shown in Figure 4.

Parameter settings. In order to determine the edge construction time t_e (Equation 2), each participant was tasked to formulate five edges in random queries using our GUI. The edge construction times are recorded for all participants and all trials. Then, t_e is set to average of these times, which is around 2 sec. Hence, $t_{lat} = t_e = 2s$.

7.2 Experimental Results

Exp 1: 3 Strategies vs 1 Strategy for IC. First, we evaluate the effect of adopting three different strategies (*i.e., neighbour search*, *two-hop search* and *large upper search*) vs using only *large upper search* for populating the AIVs in IC. We use the *DBLP* dataset and measure the average *system response time* (SRT) for all queries. Specifically, the SRT measures the duration between the time a user clicks on the Run icon and the upper-bound constrained matching results generation (*i.e.,* user waiting time). Figure 5 reports the results. Observe that using three strategies resulted in significantly smaller SRT for all queries. Hence, adoption of our proposed 3-strategy approach is better than single strategy for IC. The results on *WordNet* and *Flickr* are qualitatively similar. In subsequent experiments, we shall adopt the 3-strategy IC approach.

Exp 2: Pruning vs No Pruning. Next, we evaluate the importance of pruning isolated vertices in terms of the average CAP index size and the average SRT. Figure 6 reports the results for the *DBLP* dataset. Observe that pruning of isolated vertices results in significantly smaller SRT (Figure 6(a)) and yields a more space-efficient CAP index (Figure 6(b)) in practice due to the reduction in $|V_{q_i}|$.

Exp 3: Comparison Between BU, IC, DR, and DI. In this set of experiments, we investigate (1) the SRTs of BU, IC, DR and DI; and (2) CAP construction for IC, DR and DI. We set the maximum limit on the SRT at 2 hours and use new query instances by modifying the upper bounds on the edges of the template queries in the following ways. For *WordNet* (resp. *Flickr*), we modify $e_1.upper$ of all queries but Q_5 (resp. all queries) to 5. In *WordNet*, $e_1.upper$ of Q_5 is set to 4. We set $e_2.upper$ in Q_1 and Q_5 to 1 for *WordNet*. We modify it to 5

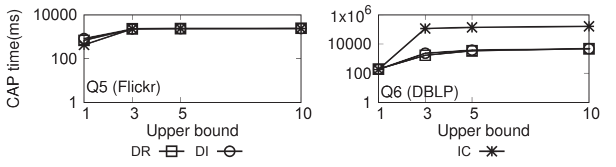

Figure 10: Effect of upper bound on CAP construction.

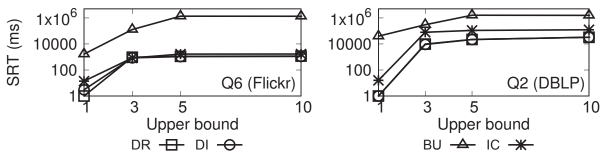

Figure 11: Effect of upper bound on SRT.

for *all* queries in *Flickr*. For both datasets, we modify $e_3.upper$ for Q_3 and Q_5 to 1. Lastly, $e_5.upper$ and $e_6.upper$ are modified to 1 and 2, respectively, for Q_6 for both datasets. *DBLP* uses the same edge bound as *Flickr* for all queries except Q_5 where $e_3.upper$ is set to 3.

Figure 7 plots the results for representative queries on the three datasets. Observe that the SRT of BU is at least one order of magnitude slower than IC, which in turn, is at least one magnitude slower than DR and DI for *WordNet* and *DBLP*. Note that BU did not finish execution for Q_2 and Q_4 in *WordNet* within 2 hrs. This highlights the benefit of the visual querying paradigm in comparison to traditional evaluation. For CAP construction, the results on *DBLP* are qualitatively similar to *WordNet*. Observe that the effect of the deferment strategy is most obvious in *WordNet* where such strategy resulted in superior CAP construction time (Figure 8, *WordNet*) and index size (Figure 9, *WordNet*). This is due to large $|V_{q_i}|$ for *WordNet* which resulted in some expensive edges benefiting from deferment. For instance, in Q_2, $|V_{q_1}| = 5501$ and $|V_{q_2}| = 63099$ and (q_1, q_2) is an expensive edge that is deferred to be processed after Run icon is clicked. In comparison, all edges in Q_2 for *Flickr* are inexpensive and are processed immediately. Hence, the CAP construction times for Q_2 on *Flickr* are similar. Note that even though the same expensive edges are deferred in DR and DI, majority of them in the latter are processed before Run is clicked.

Exp 4: Varying upper bound. We examine the effect of varying the upper bound constraints on average SRT and CAP construction time (impact on CAP size is discussed in Appendix D). We use the template queries Q_2, Q_5, and Q_6 on *DBLP* and *Flickr*. In real-world networks, due to ultra-small world property [8] the average node-to-node distance is approximately 5 [30]. Hence, we vary the upper bound constraints to {1, 3, 5, 10} in the following ways. For *DBLP*, we vary the upper bounds of e_1 and e_2 in Q_2. For Q_6 we set $e_5.upper = e_6.upper = 2$ and vary the upper bounds of e_1 and e_2. Similarly, for *Flickr*, we set $e_3.upper = 1$, $e_4.upper = 2$ and vary upper bound of e_2 for Q_5. For Q_6 , we set $e_4.upper = 2$, $e_5.upper = 2$, $e_6.upper = 1$, and vary the upper bounds of e_1 and e_3.

Figures 10 and 11 plot the results. Naturally, cost increases as we explore larger part of the data graphs. However, as can be seen the increase flattens out in all cases as upper bound increases. This is likely due to the pruning of isolated vertices that arise from stricter bounds associated with neighbouring query edges. Specifically, the deferment strategies are found to be useful in reducing SRT and CAP construction time especially for higher bounds in *DBLP*. In addition, DI has either the same or shorter SRT in a majority of test cases, highlighting the superiority of this strategy. Observe that both these strategies are orders of magnitude faster than BU.

Other results. We have conducted additional experiments that are reported in Appendix D and [29]. In summary, we observe the followings: (a) It typically takes less than 5 sec to check for lower bound to visualize result subgraphs. (b) Query modification can be realized efficiently. (c) The SRT is insensitive to the query formulation sequence (QFS) due to the deferment-based schemes.

8 RELATED WORK

Fan et al. is the first to study the problem of p-homomorphic and 1-1 p-homomorphic matching in [13]. In contrast to our BPH query, they matched vertices based on a similarity matrix and matched an edge to a path of arbitrary length. In [14], they examine the problem of p-hom pattern matching in polynomial time using *bounded simulation* [15]. There are also efforts that investigate pattern matching using distance-join queries [35, 38] in traditional setting. [38] specifies only a *global* upper bound for the query. [35] examines bounded query processing in a distributed environment. In particular, they consider only upper bounds and find vertex matches without enumerating all vertices along the paths. In contrast, BOOMER considers both upper and lower bounds and enumerates all path embeddings of the results. More importantly, all these work follow the conventional query processing paradigm where the formulation (visual or textual) of a query is independent of its evaluation.

Majority of incremental algorithms for graphs [10–12] focus on incremental update of graph query results in response to the changes in the underlying graph. In contrast, given a data graph we focus on updating partial results as the p-hom query fragment evolves during formulation in a visual graph querying environment.

The seminal efforts by Jin et al. [21, 22] investigated this visual querying paradigm in the context of substructure search over a large collection of small or medium-sized data graphs. In [20], it is extended to large networks. These efforts focus on subgraph isomorphism-based queries instead of more generic 1-1 p-homomorphic queries. Furthermore, they focus on edge-to-edge mapping and each visually constructed edge is *immediately* processed by exploiting the GUI latency. In contrast, BOOMER focuses on edge-to-path mapping and the processing of a constructed edge may be deferred to a more opportune time due to the expensive nature of edge-to-path mapping in BPH queries.

9 CONCLUSIONS

In this paper, we present a novel indexing scheme called CAP to efficiently support a visual graph querying framework that efficiently blends a generic but complex type of graph pattern query (BPH query) that is derived from 1-1 p-hom mapping [13]. This is primarily due to the deployment of a novel deferment-based blending strategy that facilitates efficient construction and maintenance of the CAP index. Experimental studies on real data graphs validated the merit of our proposed technique.

Acknowledgments. Y. Song, S. S. Bhowmick and H. E. Chua are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. B. Choi is supported by HK-RGC GRF 12201315 and 12232716. S. Zhou is supported by National Natural Science Foundation of China (NSFC) under grant No. U1636205.

REFERENCES

[1] T. Akiba, Y. Iwata, Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. *In SIGMOD*, 2013.

[2] S. S. Bhowmick, B. Choi, C. Li. Graph Querying Meets HCI: State of the Art and Future Directions. *In SIGMOD*, 2017.

[3] S.S. Bhowmick, H.-E. Chua, B. Choi, C. Dyreson. VISUAL: Simulation of Visual Subgraph Query Formulation To Enable Automated Performance Benchmarking. *In TKDE*, 29(8), 2017.

[4] F. Bi, L. Chang, X. Lin, L. Q, W. Zhang. Efficient subgraph matching by postponing cartesian products. *In SIGMOD*, 2016.

[5] A. Bonifati, W. Martens, T. Timm. An Analytical Study of Large SPARQL Query Logs. *In VLDB*, 2017.

[6] N. Buchan and R. Croson. The boundaries of trust: own and others actions in the US and china. *Journal of Economic Behavior and Organization*, 55(4):485–504, 2004.

[7] D.H. Chau, C. Faloutsos, H. Tong, J.I. Hong, B. Gallagher, T. Eliassi-Rad. GRAPHITE: A visual query system for large graphs. *ICDM Workshop*, 2008.

[8] R. Cohen, S. Havlin. Scale free networks are ultrasmall. *Phys.Rev. Lett.* 90, 2003.

[9] C.V. Dang, E.P. Reddy, K.M. Shokat, L. Soucek. Drugging the 'Undruggable' Cancer Targets. *Nat. Rev. Cancer*, 17:502-508, 2017.

[10] C. Demetrescu, D. Eppstein, Z. Galil, G. F. Italiano. Dynamic Graph Algorithms. *In Algorithms and theory of computation handbook.* Chapman & Hall/CRC, 2010.

[11] W. Fan, C. Hu, C. Tian. Incremental Graph Computations: Doable and Undoable. *In SIGMOD*, 2017.

[12] W. Fan, X. Wang, Y. Wu. Incremental Graph Pattern Matching. *TODS*, 38(3), 2013.

[13] W. Fan, J. Li, S. Ma, H. Wang, Y. Wu. Graph homomorphism revisited for graph matching. *In VLDB*, 2010.

[14] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, Y. Wu. Graph pattern matching: from intractable to polynomial time. *In VLDB*, 2010.

[15] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[16] L. Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3), 2003.

[17] F. Haag, S. Lohmann, S. Bold, T. Ertl. Visual SPARQL Querying based on Extended Filter/flow Graphs. *In AVI*, 2014.

[18] W.-S. Han, J. Lee, J.-H. Lee. TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013.

[19] K. Huang, S. S. Bhowmick, S. Zhou, B. Choi. PICASSO: Exploratory Search of Connected Subgraph Substructures in Graph Databases. *PVLDB*, 10(12): 1861-1864, 2017.

[20] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, S. Zhou. QUBLE: Towards Blending Interactive Visual Subgraph Search Queries on Large Networks. *VLDB J.* 23(3), 2014.

[21] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. Gblender: towards blending visual query formulation and query processing in graph databases. In *ACM SIGMOD*, 2010.

[22] C. Jin, S. S. Bhowmick, B. Choi, S. Zhou. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *In ICDE*, 2012.

[23] J. Lazar, J. H. Feng, H. Hochheiser. Research Methods in Human-Computer Interaction. John Wiley & Sons, 2010.

[24] R. Pienta, A. Tamersoy, A. Endert, S. Navathe, H. Tong, D. H.Chau. VISAGE: Interactive Visual Graph Querying. *In AVI*, 2016.

[25] R. Pienta, F. Hohman, A. Tamersoy, A. Endert, S. B. Navathe, H. Tong, D. H. Chau. Visual Graph Query Construction and Refinement. *In SIGMOD*, 2017.

[26] R.L. Perlman. Mouse Models of Human Disease: An Evolutionary Perspective. *Evol. Med. Public Health* 2016(1):170-176, 2016.

[27] D.D. Shaye, I. Greenwald. OrthoList: a compendium of C. elegans genes with human orthologs. *PloS one* 6(5):e20085, 2011.

[28] B, Schröder. Ordered Sets: An Introduction with Connections from Combinatorics to Topology. *Springer International Publishing*, 2016.

[29] Y. Song, H. E. Chua, S. S. Bhowmick, B. Choi, S. Zhou. BOOMER: Blending Visual Formulation and Processing of P-Homomorphic Queries on Large Networks. *Technical Report*, 2017. Available at http://www.ntu.edu.sg/home/assourav/TechReports/boomer-TR.pdf.

[30] F.W. Takes and W.A. Kosters. Determining the diameter of small world networks. *In CIKM*, 2011.

[31] C. Ware, P. Mitchell. Visualizing Graphs in Three Dimensions. *ACM Transactions on Applied Perception (TAP)*, 5(1), 2008.

[32] S. Yang, Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu, X. Yan. SLQ: A User-friendly Graph Querying System. *In SIGMOD*, 2014.

[33] M.A. Yıldırım, K.I. Goh, M.E. Cusick, A.L. Barabási, M. Vidal. Drug-target Network. *Nat. Biotechnol.*, 25(10):1119-1126, 2007.

[34] H. Yu, N.M. Luscombe, H.X. Lu, X. Zhu, Y. Xia, J.D.J. Han, N. Bertin, S. Chung, M. Vidal, M. Gerstein. Annotation transfer between genomes: protein-protein interologs and protein-DNA regulogs. *Genome Res.*, 14(6):1107-1118, 2004.

[35] X. Zhang, L. Chen, M. Wang. Efficient parallel processing of distance join queries over distributed graphs. *TKDE*, 27(3): 740-754, 2015.

[36] P. Zhao, J. Han. On graph query optimization in large networks. *In VLDB*, 2010.

[37] G. Zhu, X. Lin, K. Zhu et al. TreeSpan: Efficiently Computing Similarity All-Matching. *In SIGMOD*, 2012.

[38] L. Zou, L. Chen, M.T. Özsu. Distance-join: Pattern match query in a large graph database. *In VLDB*, 2009.

APPENDIX A PROOFS

Proof of Lemma 5.2. In the worst case, the data graph is fully connected and the upper bounds are high. Hence, all vertices in V_{q_i} and V_{q_j} are connected and no pruning occurs, resulting in a worst case space complexity of $O(\sum_{q_i \in V_B} |V_{q_i}| + \sum_{e(q_i,q_j) \in E_B} |V_{q_i}| \times |V_{q_j}|)$.

Proof Sketch of Lemma 5.3. During an out-scan, the neighbourhood of v_i is scanned linearly resulting in a cost equivalent to the degree of v_i ($deg(v_i)$). Further, it checks the existence of every neighbour vertex v_j having label $L(v_j) = L(q_j)$ in the set V_{q_j}. The cost of such checks is $O(log|V_{q_j}|)$ if V_{q_j} is stored in order. Since this operation has to be performed $deg(v_i) \times P(v_j|L(v_j) = L(q_j))$ where $P(v_j|L(v_j) = L(q_j))$ is the probability of a vertex in G having the same label as q_j and denoted as $P_{L(q_j)}$, cost for the out-scan is $Cost_{out} = O(deg(v_i) + deg(v_i) \times P_{L(q_j)} \times log(|V_{q_j}|))$. During in-scan, it checks whether every vertex $v_j \in V_{q_j}$ having label $L(v_j) = L(q_j)$ is adjacent to v_i. The cost of such a check is $O(log(deg(v_i))$. Since this search is performed $|V_{q_j}|$ times, the overall cost is $Cost_{in} = O(|V_{q_j}| \times log(deg(v_i)))$.

Proof Sketch of Lemma 5.4. During an out-scan, let the 2-hop neighbourhood of v_i be $TwoHop(v_i)$. Then, the cost of a linear scan for the 2-hop neighbours of v_i is $TwoHop(v_i)$. The remainder of this proof follows that in Lemma 5.3 and the cost for the out-scan is $Cost_{out} = O(TwoHop(v_i) + TwoHop(v_i) \times P_{L(q_j)} \times log(|V_{q_j}|))$. During in-scan, it checks whether every vertex $v_j \in V_{q_j}$ having label $L(v_j) = L(q_j)$ is within 2 hops of v_i. This can be achieved by looking for common neighbours existing between the 1-hop neighbourhoods of v_i and v_j using a merge-join-like algorithm ($O(deg(v_i) + deg(v_j))$ worst case time complexity) assuming the neighbourhoods are sorted. Hence, taken together, $Cost_{in} = O(|V_{q_j}| \times (deg(v_i) + deg(v_j)))$.

Proof Sketch of Lemma 5.5. For a given pair of vertices v_i and v_j, PML can compute the 2-hop cover in $O(|C(v_i)| + |C(v_j)|)$ time using a merge-join-like algorithm [1] where $C(v_i)$ is the distance-aware 2-hop cover of v_i. Since, there are $|V_{q_i}||V_{q_j}|$ pair of vertices to consider, the worst case time complexity is $O(|V_{q_i}||V_{q_j}| \times (|C(v_i)| + |C(v_j)|))$.

Proof Sketch of Lemma 5.6. For every vertex q_i, the maximum number of vertices that can be removed is $|V_{q_i}|$. Hence, the maximum number of pruning step is $\sum_{q_i \in V_B} |\{v|L(v) = L(q_i), \forall v \in V\}|$.

Proof Sketch of Just-in-Time lower bound checking. The lower bound checking involves performing *DetectPath* on every query graph edge. In the worst case, *DetectPath* requires *lower* times of "detours" and each detour requires checking all neighbours and performing a shortest path computation. Since *lower* can be considered as a constant value that is usually relatively small, the time complexity is $O(\theta_{max} \times \mathbb{S})$ where θ_{max} is the maximum degree of a vertex in the data graph G and $\mathbb{S}$ is the time complexity of shortest path computation for a given pair of vertices. Shortest path computation using 2-hop index has time complexity of $O(\sqrt{|E|})$ [38]. Hence, the time complexity of Just-in-Time lower bound checking can be rewritten as $O(|E_B| \times \theta_{max} \times \sqrt{|E|})$.

Proof Sketch of Lemma 6.1. In the worst case, query modification CAP for *Defer-to-Idle* occurs on the first edge and all previously constructed query edges have been processed, yielding a single connected component for CAP. During deletion or loosening of bound for first edge, the entire CAP index is destroyed and during "roll-back", $|V_{q_i}|$ is recomputed ($O(\sum_{q_i \in V_B} |V_{q_i}|)$) and the affected edges are added to the pool ($O(1)$). In the worst case, there are inexpensive edges that can be processed in the pool when its probed. Hence, the edge processing cost is ($O(\sum_{e(q_i,q_j) \in E_{ne}} |V_{q_i}| \times |V_{q_j}|)$) where E_{ne} is the set of inexpensive edges in the pool that can be processed within the idle time. Hence, for deletion and loosening of bounds, the worst case time complexity is $O(\sum_{q_i \in V_B} |V_{q_i}| + \sum_{e(q_i,q_j) \in E_{ne}} |V_{q_i}| \times |V_{q_j}| + n)$ where n is the number of pruning steps. In contrast, during tightening of bound, every pair of $(u, v) \in V_{q_i} \times V_{q_j}$ has to be checked again and the worst case time complexity is $O(|V_{q_i}| \times |V_{q_j}| + n)$. Hence, the worst case time complexity for query modification occurs during deletion and loosening of bounds.

APPENDIX B PSEUDOCODE

B.1 Pseudocode for Immediate Strategy

Algorithm 6 Procedure *ProcessEdge*

```
Require: (Partial) $\mathbb{C} = (V_C, E_C)$, new query edge $e_q = (q_i, q_j)$, $e_q$ $e_q.upper$, $G = (V, E, L)$, $Q_B = (V_B, E_B, L, \lambda)$, $\mathbb{P}$;
Ensure: (Partial) CAP index $\mathbb{C}$;
 1: for $v_i \in V_{q_i}$ do
 2:     $V_{q_i}^{q_j}(v_i) \leftarrow \phi$
 3: end for
 4: for $v_j \in V_{q_j}$ do
 5:     $V_{q_j}^{q_i}(v_j) \leftarrow \phi$
 6: end for
 7: $\mathbb{C} \leftarrow$ UpdateCAPEdge($\mathbb{C}$, $V_{q_i}^{q_j}$, $V_{q_j}^{q_i}$)
 8: $\mathbb{C} \leftarrow$ PopulateVertexSet($\mathbb{C}$, $(q_i, q_j)$, $upper$, $G$, $\mathbb{P}$) /* Algorithm 8*/
 9: for vertex $v_i \in V_{q_i}$ do
10:     if $V_{q_i}^{q_j}(v_i) = \emptyset$ then
11:         $C' \leftarrow$ Prune($C'$, $v_i$, $V_{q_i}$, $Q_B$)
12:     end if
13: end for
14: for vertex $v_j \in V_{q_j}$ do
15:     if $V_{q_j}^{q_i}(v_j) = \emptyset$ then
16:         $C' \leftarrow$ Prune($C'$, $v_j$, $V_{q_j}$, $Q_B$)
17:     end if
18: end for
```

Algorithm 7 Procedure *Prune*

```
Require: (Partial) CAP index $\mathbb{C} = (V_C, E_C)$, current CAP index vertex ($v_k$) and the candidate vertex set it belongs to $V_{q_k}$, BPH query $Q_B$;
Ensure: (Partial) CAP index ($\mathbb{C}$);
 1: $V_{q_k} \leftarrow V_{q_k} \setminus \{v_k\}$
 2: for $q_j \in$ Neighbour($q_k$, $Q_B$) do
 3:     for $v_j \in V_{q_k}^{q_j}(v_k)$ do
 4:         $V_{q_j}^{q_k}(v_j) \leftarrow V_{q_j}^{q_k}(v_j) \setminus \{v_k\}$
 5:         if $V_{q_j}^{q_k}(v_j) = \emptyset$ then
 6:             $\mathbb{C} \leftarrow$ Prune($\mathbb{C}$, $v_j$, $V_{q_j}$, $Q_B$)
 7:         end if
 8:     end for
 9: end for
```

Algorithm 8 Procedure *PopulateVertexSet*

```
Require: (Partial) CAP index $\mathbb{C} = (V_C, E_C)$, new edge $e_q = (q_i, q_j)$, upper bound of $e_q$ $e_q.upper$, data graph $G = (V, E, L)$, PML index $\mathbb{P}$;
Ensure: (Partial) CAP index after adding $e_q$;
 1: $V_{q_i} \leftarrow$ GetCandVertexFromCAP($q_i$)
 2: $V_{q_j} \leftarrow$ GetCandVertexFromCAP($q_j$)
 3: if $upper = 1$ then
 4:     $\mathbb{C} \leftarrow$ NeighborSearch($V_{q_i}$, $V_{q_j}$, $G$)
 5: else if $upper = 2$ then
 6:     $\mathbb{C} \leftarrow$ TwoHopSearch($V_{q_i}$, $V_{q_j}$)
 7: else
 8:     for vertex $v_i \in V_{q_i}$ do
 9:         for vertex $v_j \in V_{q_j}$ do
10:             if Distance($v_i$, $v_j$, $G$, $\mathbb{P}$) $\leq upper$ then
11:                 $V_{q_i}^{q_j}(v_i) \leftarrow V_{q_i}^{q_j}(v_i) \bigcup \{v_j\}$
12:                 $V_{q_j}^{q_i}(v_j) \leftarrow V_{q_j}^{q_i}(v_j) \bigcup \{v_i\}$
13:             end if
14:         end for
15:     end for
16: end if
```

Algorithm 9 Procedure *NeighborSearch*

```
Require: Candidate vertex sets $V_{q_i}$ and $V_{q_j}$, data graph $G = (V, E, L)$;
Ensure: (Partial) CAP index $\mathbb{C} = (V_C, E_C)$;
 1: for vertex $v_i \in V_{q_i}$ do
 2:     $Cost_{out} \leftarrow deg(v_i) + deg(v_i) \times p_{L(q_j)} \times log(|V_{q_j}|)$
 3:     $Cost_{in} \leftarrow |V_{q_j}| \times log(deg(v_i))$
 4:     if $Cost_{out} < Cost_{in}$ then
 5:         for vertex $v_j \in$ Neighbour($v_i$, $G$) do
 6:             if $L(v_j) = L(q_j)$ and $v_j \in V_{q_j}$ then
 7:                 $V_{q_i}^{q_j}(v_i) \leftarrow V_{q_i}^{q_j}(v_i) \bigcup \{v_j\}$
 8:                 $V_{q_j}^{q_i}(v_j) \leftarrow V_{q_j}^{q_i}(v_j) \bigcup \{v_i\}$
 9:             end if
10:         end for
11:     else
12:         for vertex $v_j \in V_{q_j}$ do
13:             if $v_j \in$ Neighbour($v_i$, $G$) then
14:                 $V_{q_i}^{q_j}(v_i) \leftarrow V_{q_i}^{q_j}(v_i) \bigcup \{v_j\}$
15:                 $V_{q_j}^{q_i}(v_j) \leftarrow V_{q_j}^{q_i}(v_j) \bigcup \{v_i\}$
16:             end if
17:         end for
18:     end if
19: end for
```

B.2 Pseudocode for Deferment-based Strategy

Algorithm 10 Procedure *ProbePool*

```
Require: GUI action $a$, action stream $stream$, CAP index $\mathbb{C} = (V_C, E_C)$, data graph $G = (V, E, L)$, BPH query $Q_B = (V_B, E_B, L, \lambda)$, edge pool $pool$, average edge processing time $t_{avg}$, PML index $\mathbb{P}$, idle time $t_{idle}$;
Ensure: (Partial) CAP index $\mathbb{C}$, edge pool $pool$;
 1: $t_{max} \leftarrow t_{idle}$
 2: while $t_{max} > 0$ and $stream$ has no $FreshAction$ and $pool \neq \phi$ do
 3:     $(q_i, q_j)$, $upper \leftarrow$ GetMinEdge($pool$)
 4:     if $|V_{q_i}| \times |V_{q_j}| \times t_{avg} \leq t_{max}$ then
 5:         $\mathbb{C} \leftarrow$ ProcessEdge($\mathbb{C}$, $(q_i, q_j)$, $upper$, $G$, $Q_B$, $\mathbb{P}$)
 6:         $pool \leftarrow$ Remove($pool$, $(q_i, q_j)$)
 7:         $t_{max} \leftarrow$ UpdateTime($t_{max}$)
 8:     else
 9:         $t_{max} \leftarrow 0$
10:     end if
11: end while
```

Algorithm *ProbePool*. The procedure is outlined in Algorithm 10. It checks if a new GUI action has joined the action stream and terminates the probing to handle any new action (Line 2). Note that prioritizing new GUI action processing over edge processing in *pool* allows us to further reduce the size of V_{q_i} and the AIVS where applicable, reducing T_{est} of edges in *pool*. Suppose *stream* has no new action, then it selects the topmost edge for processing

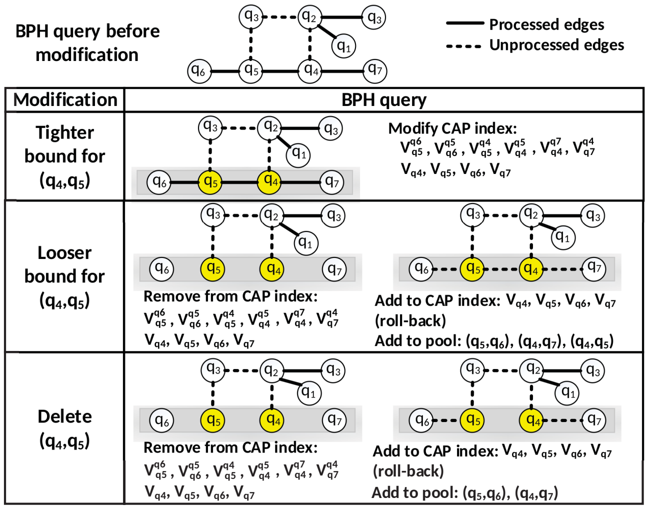
BPH query before modification
Processed edges
Unprocessed edges
Modification
BPH query
Tighter bound for (q4,q5)
Modify CAP index:
V_{q5}^{q6}, V_{q6}^{q5}, V_{q5}^{q4}, V_{q4}^{q5}, V_{q4}^{q7}, V_{q7}^{q4}
V_{q4}, V_{q5}, V_{q6}, V_{q7}
Looser bound for (q4,q5)
Remove from CAP index:
V_{q5}^{q6}, V_{q6}^{q5}, V_{q5}^{q4}, V_{q4}^{q5}, V_{q4}^{q7}, V_{q7}^{q4}
V_{q4}, V_{q5}, V_{q6}, V_{q7}
Add to CAP index: V_{q4}, V_{q5}, V_{q6}, V_{q7}
(roll-back)
Add to pool: (q5,q6), (q4,q7), (q4,q5)
Delete (q4,q5)
Remove from CAP index:
V_{q5}^{q6}, V_{q6}^{q5}, V_{q5}^{q4}, V_{q4}^{q5}, V_{q4}^{q7}, V_{q7}^{q4}
V_{q4}, V_{q5}, V_{q6}, V_{q7}
Add to CAP index: V_{q4}, V_{q5}, V_{q6}, V_{q7}
(roll-back)
Add to pool: (q5,q6), (q4,q7)

Figure 12: Query modification.

if its $T_{est} \leq t_{max}$ (Lines 3 to 7). Otherwise, it terminates the pool-probing to await the next GUI action since edges in *pool* are still too expensive to be processed (Line 9).

B.3 Pseudocode for Result Subgraphs Generation

Upper bound-constrained matching results.

Algorithm 11 Algorithm *PartialVertexSetsGen*

```
Require: BPH query $Q_B$, matching order $M : q_1 \rightarrow \cdots \rightarrow q_{|V_B|}$ and CAP index $\mathbb{C}$;
Ensure: $V_\Delta = \{V_{P_1}, \cdots, V_{P_{|V_\Delta|}}\}$;
 1: $V_\Delta \leftarrow \emptyset$
 2: $M \leftarrow$ Reorder$(M, \mathbb{C})$
 3: for vertex $v \in V_{q_1}$ do
 4:     $Q_{matched} \leftarrow \{q_1\}$
 5:     $V_P \leftarrow \{v\}$
 6:     $V_\Delta, Q_{matched} \leftarrow$ DFS$(Q_B, M, V_\Delta, V_P, Q_{matched}, q_2)$
 7: end for
```

Algorithm 12 Procedure *DFS*

```
Require: BPH query $Q_B$, matching order $M : q_1 \rightarrow \cdots \rightarrow q_{|V_B|}$, $V_\Delta$, partial-matched vertex
    set $V_P$, matched query vertices $Q_{matched}$, next query vertex to match $q_{next}$;
Ensure: $V_\Delta, Q_{matched}$;
 1: $V_{candidate} \leftarrow V_{q_{next}}$
 2: for vertex $q_k \in Q_{matched}$ do
 3:     if $(q_k, q_{next}) \in E_B$ then
 4:         $V_{candidate} \leftarrow V_{candidate} \bigcap V_{q_k}^{q_{next}}(v_k)$
 5:     end if
 6: end for
 7: $Q_{matched} \leftarrow Q_{matched} \bigcup \{q_{next}\}$
 8: for vertex $v_{next} \in V_{candidate}$ do
 9:     $V_P \leftarrow V_P \bigcup \{v_{next}\}$
10:     if $q_{next} = q_{|V_B|}$ then
11:         $V_\Delta \leftarrow V_\Delta \bigcup V_P$
12:     else
13:         $V_\Delta, Q_{matched} \leftarrow$ DFS$(Q_B, M, V_\Delta, V_P, Q_{matched}, q_{next})$
14:     end if
15:     $V_P \leftarrow V_P \setminus \{v_{next}\}$
16: end for
17: $Q_{matched} \leftarrow Q_{matched} \setminus \{q_{next}\}$
```

Visualization-friendly, just-In-time lower bound checking.

Algorithm 13 Algorithm *FilterByLowerBound*

```
Require: Partial-matched vertex set $V_P$, BPH query $Q_B = (V_B, E_B, L, \lambda)$, matching order $M$,
    data graph $G$;
Ensure: Bounded 1-1 $p$-hom result subgraph $H$;
 1: $H \leftarrow \emptyset$
 2: for $e \in E_B$ do
 3:     $(v_i, v_j) \leftarrow$ GetDataGraphEdge$(V_P, e, M)$
 4:     $P \leftarrow \emptyset$
 5:     $H \leftarrow H \bigcup$ DetectPath$(v_i, v_j, e, 0, P)$
 6: end for
```

Algorithm 14 Algorithm *DetectPath*

```
Require: Current source vertex $v_c$, target vertex $v_j$, query edge $e$, recursion counter $step$, list
    of vertices along path from $v_i$ to $v_j$ $path$;
Ensure: Qualified path $path$;
 1: if $step + dist(v_c, v_j) > e.upper$ then
 2:     Return $path$
 3: end if
 4: Mark $v_c$ as visited
 5: $path \leftarrow path \bigcup v_c$
 6: if $v_c = v_j$ then
 7:     if $step < e.lower$ then
 8:         Mark $v_c$ as unvisited
 9:         $path \leftarrow path \setminus \{v_c\}$
10:     end if
11:     Return $path$
12: end if
13: Assign $S_0 : \forall v \in$ Neighbour$(v_c)$, $dist(v, v_j) = dist(v_c, v_j) - 1$
14: Assign $S_+ : \forall v \in$ Neighbour$(v_c)$, $dist(v, v_j) \neq dist(v_c, v_j) - 1$
15: if $step + dist(v_c, v) \geq e.lower$ then
16:     $S \leftarrow$ ConcatenateSet$(S_0, S_+)$
17: else
18:     $S \leftarrow$ ConcatenateSet$(S_+, S_0)$
19: end if
20: for $p \in S$ do
21:     DetectPath$(p, v_j, e, step + 1, path)$
22: end for
23: Mark $v_c$ as unvisited
24: $path \leftarrow path \setminus \{v_c\}$
25: Return $path$
```

B.4 Pseudocode for Query Modification

Algorithm 15 Procedure *StricterUpperMod*

```
Require: (Partial) $\mathbb{C} = (V_C, E_C)$, new query edge $e_q = (q_i, q_j)$, new upper bound of $e_q$
    $e_q.upper'$, $G = (V, E, L)$, $Q_B = (V_B, E_B, L, \lambda)$;
Ensure: (Partial) CAP index $\mathbb{C}$;
 1: for vertex $v_i \in V_{q_i}$ do
 2:     for vertex $v_j \in V_{q_i}^{q_j}(v_i)$ do
 3:         if Distance$(v_i, v_j, G) > upper'$ then
 4:             $V_{q_i}^{q_j}(v_i) \leftarrow V_{q_i}^{q_j}(v_i) \setminus \{v_j\}$
 5:             $V_{q_j}^{q_i}(v_j) \leftarrow V_{q_j}^{q_i}(v_j) \setminus \{v_i\}$
 6:         end if
 7:     end for
 8: end for
 9: Prune isolated candidate vertices in $V_{q_i}$ and $V_{q_j}$ as per Line 9-18 in Algorithm 6
```

APPENDIX C EXAMPLE OF QUERY MODIFICATION

Figure 12 (top) shows examples of processed (solid lines) and unprocessed edges (dotted lines) and two connected components $\{q_1, q_2, q_3\}$ and $\{q_4, q_5, q_6, q_7\}$ during formulation of a BPH query. The tighter bound for (q_4, q_5) illustrates the CAP index modification for alteration of upper bound. On the other hand, the vertices q_4, q_5, q_6 and q_7 are identified due to looser bound for (q_4, q_5). Then, V_{q_4}, V_{q_5}, V_{q_6} and V_{q_7} are recomputed and the edges (q_5, q_6), (q_4, q_7) and (q_4, q_5) are added to the edge pool. Lastly, consider the deletion

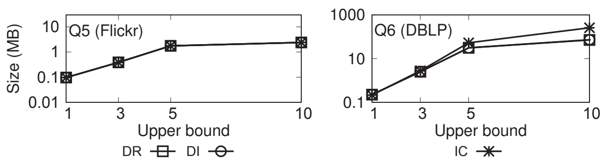

Figure 13: Effect of upper bound on CAP index size.

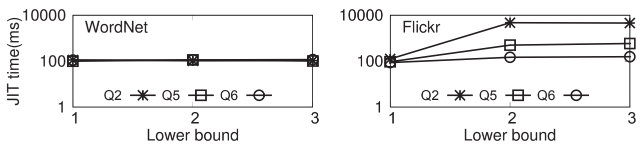

Figure 14: Effect of varying lower bound.

of (q_4, q_5). The deleted edge is not added back to the connected component $\{q_4, q_5, q_6, q_7\}$ (shaded in gray).

APPENDIX D ADDITIONAL EXPERIMENTAL RESULTS

Exp 4: Upper bound vs CAP size. Figure 13 depicts the effect of upper bound on CAP size for *Flickr* and *DBLP*. As expected, increase in upper bound yields large number of matches and hence an increase in CAP size. Importantly, observe that the CAP size is modest and can easily fit in a modern machine.

Exp 5: Cost of lower bound check. In this set of experiments, we examine the cost of checking whether the edges of a result subgraph involving $V_P \in V_\Delta$ satisfy lower bound constraints greater than 1 (Section 5.4). Specifically, we compute the average time taken by 10 random V_Ps for checking this constraint for three representative queries (Q_2, Q_5, and Q_6) over *WordNet* and *Flickr* datasets by varying the lower bounds between 1 and 3. Figure 14 reports the results. We observe that the effects of varying the lower bounds depends on the dataset and query graph. In *WordNet*, the average time is relatively constant at around 100ms for all queries whereas in *Flickr*, it ranges between 87ms to 4584ms. Nevertheless, it is still less than 5 sec, which is acceptable.

Exp 6: Query modification cost. We examine the effect of three types of query modification, namely, deletion of the first edge (to simulate worst case deletion scenario), tightening and loosening of edge bound (e_3 to e_6, if any). We use the queries Q_4, Q_5, and Q_6 on the *WordNet* and *Flickr* datasets and assume DI strategy for this experiment. In the experiments, we tighten the bound from [1,2] to [1,1] and loosen it to [1,3]. Table 1 reports the average time cost of maintaining the CAP index due to each modification. We can make the following observations. First, the cost of tightening of bounds is cognitively negligible compared to loosening of bounds and edge deletion. This is natural as additional distance queries need to be executed for loosening of bounds. However, the cost is reasonable (within 4 sec). Second, it is more expensive on *WordNet* than on *Flickr*. This is because $|V_{q_i}|$ is significantly larger in the former. Hence, the modification cost is not very sensitive to the size of input data graphs.

Exp 7: Impact of QFS. Users can construct a given BPH query by following different query formulation sequences (QFS) by constructing the edges in different order[7]. Here we investigate the effect of QFS (Table 2) for two queries, namely, Q_1 and Q_6. Figures 16, 15 and 17 depict the effect of QFS on average SRT, CAP construction time and index size, respectively. The results on *DBLP* are qualitatively similar to *WordNet*. Observe that for *WordNet*, QFS in general do not affect the deferment-based strategies (DR and DI). On the other hand, for IC, when *expensive* edges are constructed early (Q_1S1, Q_6S1 and Q_6S2), the average SRT (Figure 16, top), CAP construction time (Figure 15, top) and index size (Figure 17, top) are much greater (∼ 2 folds) than if they are constructed later. Hence, another advantage of our deferment-based strategy is that it is insensitive to QFS as it reorders the processing order of edges internally by postponing processing of expensive edges regardless of the order they are formulated.

[7]The vertex processing time is relatively constant (∼ $10ms$). Hence, we define the QFS in terms of query edges instead of query vertices.

Dataset	**Query**	Delete	Tightened Bound				Loosened Bound			
		e_1	e_3	e_4	e_5	e_6	e_3	e_4	e_5	e_6
WordNet	Q_4	2661	17	1	17	-	2474	1952	3863	-
	Q_5	424	32	17	-	-	1776	1749	-	-
	Q_6	2096	1	1	1	1	1893	2234	1752	1963
Flickr	Q_4	591	1	1	1	-	509	345	846	-
	Q_5	514	1	1	-	-	189	141	-	-
	Q_6	654	1	2	1	2	434	455	501	606

Table 1: Query modification costs (msec).

QFS	Q_1	Q_6
S1	$e_1 \rightarrow e_2 \rightarrow e_3$	$e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5 \rightarrow e_6$
S2	$e_2 \rightarrow e_1 \rightarrow e_3$	$e_4 \rightarrow e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_5 \rightarrow e_6$
S3	$e_3 \rightarrow e_2 \rightarrow e_1$	$e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_1 \rightarrow e_5 \rightarrow e_6$
S4		$e_5 \rightarrow e_6 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_1$

Table 2: QFS of Q_1 and Q_6.

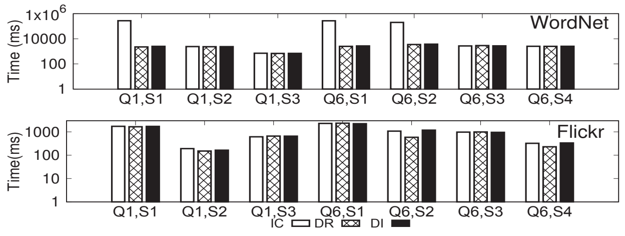

Figure 15: Effect of QFS on CAP construction time.

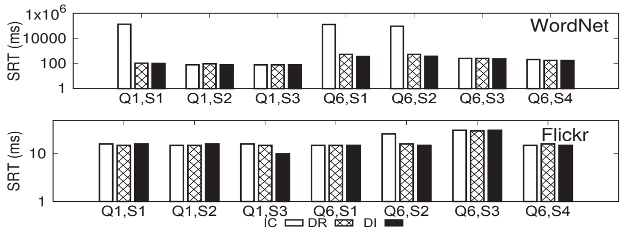

Figure 16: Effect of QFS on SRT.

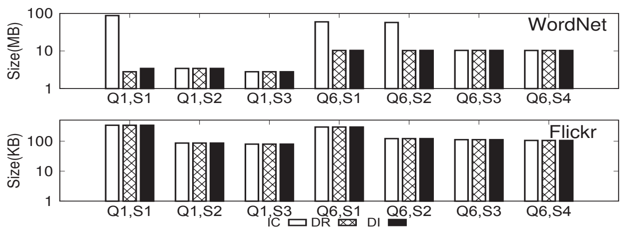

Figure 17: Effect of QFS on CAP index size.