BOOMER: A Tool for Blending Visual P-Homomorphic Queries on Large Networks

Yinglong Song[1,2] Huey Eng Chua[1] Sourav S Bhowmick[1]
Byron Choi[3] Shuigeng Zhou[2]

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

ABSTRACT

The paradigm of *interleaving* (*i.e., blending*) visual subgraph query formulation and processing by exploiting the latency offered by the GUI brings in several potential benefits such as superior *system response time* (SRT) and opportunities to enhance usability of graph databases. Recent efforts at implementing this paradigm are focused on subgraph isomorphism-based queries, which are often restrictive in many real-world graph applications. In this demonstration, we present a novel system called BOOMER to realize this paradigm on more generic but complex *bounded 1-1 p-homomorphic* (BPH) queries on large networks. Intuitively, a BPH query maps an *edge* of the query to *bounded paths* in the data graph. We demonstrate various innovative features of BOOMER, its flexibility, and its promising performance.

ACM Reference Format:
Yinglong Song, Huey Eng Chua, Sourav S Bhowmick, Byron Choi, S. Zhou. 2020. BOOMER: A Tool for Blending Visual P-Homomorphic Queries on Large Networks. In 2020 International Conference on Management of Data (SIGMOD '20), June 14–June 19, 2020, Portland, OR, USA. ACM, NewYork, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384680

1 INTRODUCTION

Visual graph query interfaces (a.k.a GUI) make it easy for non-expert users to query graphs. A recent subgraph querying paradigm [4, 5] exploits the *latency* offered by a GUI to *blend* visual query construction and processing by generating and refining candidate result matches iteratively during query formulation. It brings in several potential benefits such as superior *system response time* (SRT) and opportunities to enhance usability of graph databases by facilitating query suggestions and feedback. The early efforts, however, focused on subgraph isomorphism-based queries, which can be restrictive in many real-world applications [2, 3].

Fan *et al.* introduced the notion of *1-1 p-homomorphic* (*p*-hom) queries [2] to alleviate the restrictive usage of subgraph isomorphism-based graph queries. Intuitively, a 1-1 *p*-hom query maps an *edge* of a query to *paths* instead of edges in a data graph and measures *similarity* of vertices that goes beyond vertex label equality. In this demonstration, we present a novel framework called BOOMER (**Bo**unded 1-1 *p*-h**om** Qu**e**ry Blende**r**) [6] to realize the aforementioned visual graph querying paradigm for a class of 1-1 *p*-hom graph queries referred to as *bounded 1-1 p-hom* (BPH) queries.

Intuitively, a BPH query is a connected, undirected, simple labeled graph where an edge is mapped to a path of *bounded* length (*i.e.,* it satisfies certain length constraints) in the underlying data graph. Specifically, each edge $e = (q_i, q_j)$ 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 matches q_i and q_j (based on vertex label equality), respectively. The query in Figure 2(a) is an example of a BPH query.

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

BOOMER exploits an online, adaptive data structure called CAP *index* to facilitate the interleaving of visual BPH query formulation and query processing. Specifically, it efficiently

SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384680

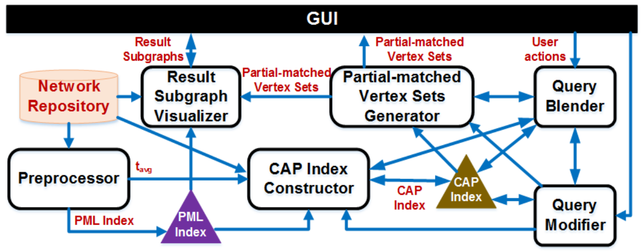


Figure 1: Architecture of Boomer.

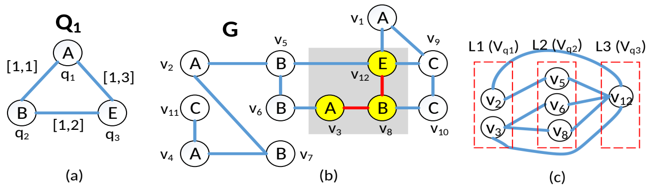


Figure 2: (a) BPH query; (b) data graph; (c) CAP index [6].

stores the candidate matching vertices that satisfy the *upper* bound constraints of edges in a (partially) constructed BPH query. It *defers* the checking of paths satisfying *lower* > 1 *until* the visualization of result matches in order to reduce the CAP construction cost. Furthermore, since iterative edge-to-path matching can be expensive, it exploits the *GUI latency* (*i.e.,* time to construct a query vertex or an edge visually) judiciously by *deferring* evaluation of an *expensive* query edge to a more opportune time during query formulation instead of evaluating it immediately after its formulation. Lastly, since visualizing result subgraphs in a large graph is cognitively challenging, *each* result match is displayed by visualizing a small subgraph of the network that contains it.

In this demo, an audience can experience Boomer's generality by easily formulating both BPH and exact subgraph queries visually on real-world networks. She will be able to view deferred evaluation of expensive edges (if any) in her query and appreciate fast processing of practical BPH queries due to this blending strategy. Finally, we shall demonstrate how an audience can visualize result matches effectively.

2 SYSTEM OVERVIEW

Figure 1 depicts the architecture of Boomer, which mainly consists of the following modules. The reader may refer to [6] for detailed algorithms and performance study of Boomer.

The GUI Module. Figure 3 depicts the GUI of Boomer. It consists of the following panels: (a) *Panel 1* depicts the datasets available for querying. (b) *Panel 2* is the canvas to visually formulate a BPH query. (c) *Panel 3* contains a list of distinct node/edge labels for the chosen dataset to aid query formulation. (d) *Panel 4* facilitates a user to initiate construction of a new query (in which case, a new query tab will be generated in *Panel 2*), execute a BPH query, and visualize result matches. (e) *Panel 5* contains a list of *partial-matched vertex sets* that satisfies a BPH query. Selection of a vertex set allows a user to visualize (in *Panel 6*) the corresponding region of the data graph containing the result subgraph (*i.e.,* matching subgraph) involving these vertices. (f) *Panel 6* displays the result subgraphs for a selected partial-matched vertex set. (g) *Panel 7* displays the query processing plan (*i.e.,* edge processing order, processing time, deferment (if any)), which is updated dynamically as blending occurs.

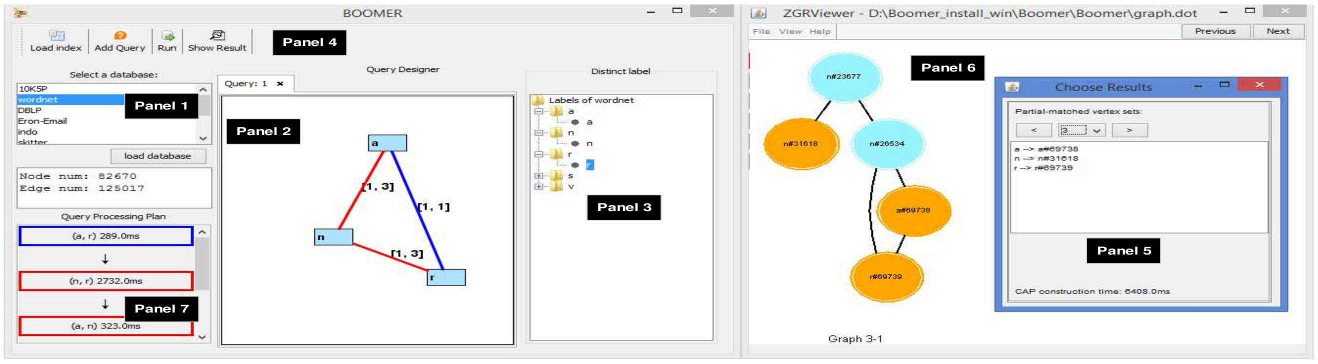
The Preprocessor Module. This module preprocesses the input data graph G to compute the *average edge processing time* t_{avg} and the *pruned landmark labelling* (PML) index[1], which are used during CAP index construction. The former is used to determine whether an edge in a BPH query is *expensive* and the latter is used for processing distance queries efficiently.

For a given data graph G, this module first computes the PML index. 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.

The CAP Index Constructor Module. As a BPH query is being drawn in Panel 2, the set of GUI actions (*i.e.,* adding a vertex, connecting two vertices) is exploited continuously to construct and maintain the *CAP (Compact Adaptive Path) index*. It enables efficient retrieval of candidate matching vertices during query formulation and results generation as distances between vertices in G do not have to be computed repeatedly. In addition, unsuitable candidates (*i.e.,* those which subsequently violate the upper bounds) can be pruned immediately. Note that the construction of a CAP index starts as soon as visual formulation of a BPH query begins and is completed after the user has clicked on the Run icon to execute the query.

Given a (partial) BPH query $Q_B = (V_B, E_B)$, a *matching order* M (*i.e.,* the order in which edges of the query is constructed), and a data graph $G = (V, E)$, 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. Note that the check for $e.lower > 1$ is deferred and handled by *Results Visualizer* module. For example, consider the data graph G and the query graph Q_1 in Figure 2(a) with matching order $M : q_1 \longrightarrow q_2 \longrightarrow q_3$. The corresponding CAP index after the construction of Q_1 is shown in Figure 2(c). The matching vertices of q_1, q_2, and q_3 that satisfy the upper bound constraints of edges in Q_1 are

[1]The PML is based on 2-hop cover and is used for fast exact distance computation. Although we use it in our current implementation, Boomer is orthogonal to the choice of exact distance computation technique.



```
Node num:  82670
Edge num:  125017
```

```
a --> a#69738
n --> n#31618
r --> r#69739
```

```
CAP construction time: 6408.0ms
```

Figure 3: The GUI of Boomer. Best viewed in colour.

$V_{q_1} = \{v_2, v_3\}$, $V_{q_2} = \{v_5, v_6, v_8\}$, 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* 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* is violated (*i.e.,* v_1 is called an *isolated vertex*). In contrast, the edge (v_2, v_{12}) exists in the index as distance between v_2 and v_{12} is 2, and satisfies *upper* of (q_1, q_3). In our framework, three different strategies are adopted to efficiently find these vertices as detailed in [6]. Intuitively, for a new edge $e = (q_i, q_j)$ it first retrieves the candidate matching vertices V_{q_i} and V_{q_j}. If $e.upper$ is 1 or 2 then the *neighbor search* or *2-hop search* is used. Otherwise, the *large upper search* scheme is utilized.

Note that processing of an edge $e = (q_i, q_j)$ can be expensive if $|V_{q_i}|$ and $|V_{q_j}|$ are very large and $e.upper$ is large as well. Specifically, e is *expensive* if $T_{est} > t_{lat}$ and $e.upper \geq 3$ where $T_{est} = |V_{q_i}| \times |V_{q_j}| \times t_{avg}$ is the *estimated time* to process a query edge e and t_{lat} is the minimum GUI latency available for processing e [6]. Consequently, the CAP index construction for an expensive edge may take significantly longer than the available GUI latency. To address this challenge, this module realizes a *deferment-based strategy* called *Defer-to-Idle* (DI) for constructing the index. The key idea here is to defer processing of expensive edges during query formulation to a time when the query processor is idle.

Let us elaborate on the DI strategy with a simple example. 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. Hence, after it is formulated, its processing is deferred. Assume now the processing time of e_2 is $t' \ll t_{lat}$. Further, it prunes many isolated vertices in V_{q_3}. Consequently, our framework is now "idle" after the construction and processing of e_2 as there are no new vertices or edges to process. 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, based on the DI strategy, e_1 is processed now by leveraging on the idle time. Panel 7 (Figure 3) displays the order of edge processing in DI, highlighting those that are deferred in red.

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.

The Partial-Matched Vertex Sets Generator Module. Once the construction of CAP index is completed, this module is invoked. Reconsider the CAP index in Figure 2. Observe that 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 this vertex set (*i.e.,* $\{v_2, v_5, v_{12}\}$) as a *partial-matched vertex set* (denoted by V_P). Intuitively, it represents vertices of a matching result of the query that satisfies the upper bound constraints. Clearly, there can be many partial-matched vertex sets for a BPH query. Hence, it first traverses $\mathbb{C}$ to extract these vertex sets. Specifically, 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 in M. Then, it displays this set of V_P and details of partial-matched vertices in Panel 5 (Figure 3).

The Results Visualizer Module. This module generates and visualizes the results (result subgraphs) of a BPH query on Panel 6. Since it is cognitively challenging to visualize result subgraphs on a large data graph, it deploys *small region-based* visualization scheme where it iteratively shows a color-coded small subgraph of the underlying data graph that satisfies the user-specified edge bound constraints. Boomer displays at most k neighbours of a vertex (by default, $k = 5$) that matches a query vertex to provide additional contextual information. In particular, it takes a *just-in-time* approach to evaluate the lower bound constraints and to generate the result subgraph involving V_P. Given a V_P, it checks whether there exists a path satisfying the edge bounds by utilizing the PML index. Note that if $e.lower = 1$, it does not need to perform any checking as $e.lower \leq e.upper$. Panel 6 in Figure 3 shows an example of a result subgraph satisfying the query in Panel 2. A user can iterate through different subgraphs for a particular V_P satisfying the path length constraints by clicking on Previous and Next in Panel 6.

The Query Modifier Module. Since a user may modify a BPH query during query formulation, this module handles it by updating the CAP index efficiently [6]. If a deleted edge is not yet processed (*i.e.,* unprocessed edge), no change is required on the CAP index and it simply removes the unprocessed edge. Otherwise, deletion of a processed edge is handled by reconstructing the *affected* region in the index.

For the case of alteration of bounds, there are two scenarios: alteration of *lower* and *upper* bounds. Since the CAP index considers only the upper bound, there is no change in its structure when the lower bound is altered. In the latter case, when e is unprocessed, it updates the bound of the unprocessed edge without modifying the index. On the other hand, if e is processed and the upper bound is modified to be stricter, a vertex may not satisfy the tighter distance constraint anymore. Hence, it needs to examine every relevant pair of vertices to reassess whether the new $e.upper$ is satisfied and remove ones that violate it. Conversely, when the upper bound is loosened, a vertex that satisfies the new $e.upper$ may not be in the CAP index. Hence, such vertices are retrieved and the index is modified in the affected region.

The Query Blender Module. Lastly, this module coordinates the aforementioned modules. In particular, it keeps track of vertices and edges added to a BPH query Q during construction and processes them to construct and maintain the online CAP index by invoking the *CAP Index Constructor* module. When a user clicks the Run icon to execute Q, it waits (if necessary) for completion of the construction of the CAP index and then invokes the *Partial-Matched Vertex Sets Generator* module to generate the partial-matched vertex sets. If a user modifies Q during formulation, it invokes the *Query Modifier* module to update the CAP index.

3 RELATED SYSTEMS & NOVELTY

The demonstrations in [4, 5] focus on blending visual subgraph isomorphism-based queries. BOOMER exposes very *different user interaction experience* compared to these systems. First, in BOOMER, one needs to specify the lower and upper bounds on each edge during query formulation (Panel 2), which is irrelevant in [4, 5]. Second, since the order of execution of edges in [4, 5] are always identical to the query formulation sequence, these systems *do not* expose users to experience deferment-based execution strategy (Panels 2 and 7). Third, the result visualization experience (Panel 6) in BOOMER is different as a user may iteratively visualize matching subgraphs with varying path length constraints.

The internals of BOOMER is also different from [4, 5]. The query processing algorithm of BOOMER can handle both edge-to-edge and edge-to-path mapping and the processing of a constructed edge may be deferred to a more opportune time. Besides, the structure and construction process of the CAP index are different from the online indexes deployed in [4, 5] as the latter do not capture bounded path length information.

4 DEMONSTRATION OVERVIEW

BOOMER is implemented in Java JDK 1.7. Our demonstration will be loaded with real-world networks from different domains and sizes (up to 2M nodes). Example BPH and subgraph matching queries will be presented. Users can also write their own ad-hoc queries through our GUI.

A key objective is to enable the audience to interactively experience the formulation and blending of visual BPH queries. First, one can select a dataset from Panel 1. Next, she can formulate a BPH query (or a subgraph matching query) in Panel 2 by dragging and dropping vertex labels from Panel 3, connecting them, and specifying the lower and upper bounds for each edge in a modal box. Specifically, one can progress from a subgraph matching query to a BPH and vice versa by simply modifying the bounds. One can also modify the query fragment anytime during query formulation.

The blending of a query is visualized in Panels 2 & 7. The query edges in Panel 2 are dynamically colour-coded to differentiate unprocessed (black) and processed (blue or red) edges during blending. Specifically, blue color denotes immediate processing of an edge whereas a red edge is an expensive edge whose processing is deferred (DI strategy) during formulation. One can right-click on an edge to view various details (*e.g.,* status, processing time). Panel 7 shows the processing plan (order of execution of query edges, edge processing time). The colour coding in Panel 2 applies here.

Finally, when one clicks the Run button (Panel 4), BOOMER displays the partial-matched vertex sets (V_Ps) in Panel 5. One can select a V_P to interactively view and explore result matches in Panel 6 as discussed earlier. A user can iterate through different subgraphs satisfying various path length constraints involving a V_P.

A **short video** to illustrate the main features using example use cases is available at https://youtu.be/bdYoXYVVyCA.

Acknowledgments. The first three authors are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. Byron Choi is supported by HKRGC GRF 12201119 and 12201518.

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] W. Fan, J. Li, S. Ma, H. Wang, Y. Wu. Graph Homomorphism Revisited for Graph Matching. *In PVLDB*, 2010.
[3] W. Fan, J. Li, et al. Graph Pattern Matching: from Intractable to Polynomial Time. *In PVLDB*, 2010.
[4] H. Hung, S. S. Bhowmick, et al. QUBLE: Blending Visual Subgraph Query Formulation with Query Processing on Large Networks. *In SIGMOD*, 2013.
[5] C. Jin, et al. GBLENDER: Visual Subgraph Query Formulation Meets Query Processing. *In SIGMOD*, 2011.
[6] Y. Song, H. E. Chua, et al. BOOMER: Blending Visual Formulation and Processing of P-Homomorphic Queries on Large Networks. *In SIGMOD*, 2018.