PIGEON: Progress Indicator for Subgraph Queries

Xiaojing Xie [#1], Zhe Fan [*2], Byron Choi [*2], Peipei Yi [*2], Sourav S Bhowmick [†3], Shuigeng Zhou [#1]

[#] *Fudan University, China*
[1] {xiexiaojing, sgzhou}@fudan.edu.cn

[*] *Hong Kong Baptist University, Hong Kong*
[2] {zfan, bchoi, csppyi}@comp.hkbu.edu.hk

[†] *Nanyang Technological University, Singapore*
[3] assourav@ntu.edu.sg

***Abstract*—Subgraph queries have been a fundamental query for retrieving patterns from graph data. Due to the well known NP hardness of subgraph queries, those queries may sometimes take a long time to complete. Our recent investigation on real-world datasets revealed that the performance of queries on graphs generally varies greatly. In other words, query clients may occasionally encounter "unexpectedly" long execution from a subgraph query processor. This paper aims to demonstrate a tool that alleviates the problem by monitoring subgraph query progress. Specifically, we present a novel subgraph query progress indicator called PIGEON that exploits *query-time information* to report to users accurate estimated query progress. In the demonstration, users may interact with PIGEON to gain insights on the query evaluation, which include the following: Users are enabled to (i) monitor query progress; (ii) analyze the causes of long query times; and (iii) abort queries that run abnormally long, which may sometimes contain human errors.**

I. INTRODUCTION

Graphs are powerful tools for a wide range of real applications, including chemical and biological databases, social networks and information networks. Subgraph queries have known to be a fundamental and powerful query for retrieving patterns from such networks. Recently, a large body of techniques has been proposed to enhance performances of subgraph queries ($e.g.$, [1]). However, due to the well known NP-hardness of subgraph queries, their optimizations are necessarily heuristic. As a result, not surprisingly, the runtimes of seemingly similar queries may vary significantly.

For example, we experimented the runtime performances of randomly generated 100 queries of size 4 (using DFS) using our implementation of the Ullmann's algorithm [2] on the *Youtube*, *Amazon*, and *DBLP* datasets. Interestingly, the runtimes of the queries that can be completed within ten minutes exhibit huge standard deviations (SD) as reported in Table I, not to mention those did not finish in ten minutes. In particular, for the *Youtube* dataset, the average runtime is 16.2s, whereas the SD is 78.7. However, users may wait for at least 252s ($i.e.$, +3SD) for 5% of the time. Another example is that users may draw some similar queries to explore a network but their runtimes can be completely different (Fig. 1). At first glance, it may seem that this phenomenon is due to our implementation of the Ullmann's algorithm. However, we studied other subgraph algorithms [3] and observe a similar phenomena on other algorithms. For instance, we ran an optimized VF2 implementation on *Youtube* for similar queries of size 4 and the average runtime is 822ms and the SD is 506.

TABLE I
PRELIMINARY EXPERIMENTS ON REAL GRAPHS (QUERY SIZE = 4)

Dataset	Avg. runtimes	stddev	Timeout % (runtime > 10mins)
Youtube	16.2s	78.7	9.1%
DBLP	4.0s	24.7	2.2%
Amazon	25.7s	75.4	10%

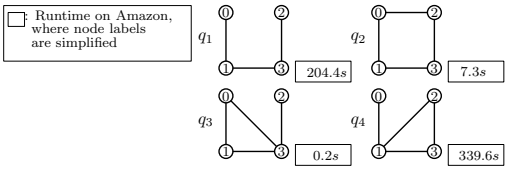

Fig. 1. Example similar subgraph queries and their runtimes (*Amazon*)

Such variability in query runtimes may adversely impact users' experience and frustrate them to use the subgraph query processor. This is more so nowadays as users attention span are arguably shortening. For another application example, the web interface of PubChem [4] informs users to refresh the page few minutes later to check if the subgraph query answers are returned. Further, interactive exploratory searches have a renewed interest in the context of chemical databases [5] and few unexpectedly long-running queries may ruin the exploration. Hence, it is desirable to have an accurate *query progress indicator* so that users are continuously informed about the progress made by the underlying query processor.

Few progress indicators have been proposed in the literature. Chaudhuri et al. [6] propose a progress indicator for SQL queries. It exploits an execution plan for a given query and estimates the progress based on their `GetNext` model. However, there is no such model for subgraph queries. Recently, Kristi et al. [7] propose a time-remaining indicator for declarative queries in the MapReduce environment. A critical-path-based progress-estimation approach is proposed to estimate the MapReduce jobs expressed in DAGs. In the context of graph databases, existing research prototypes for subgraph queries do not give progress estimates to users ($e.g.$, our recent work on visual subgraph queries [8], [9]). To the best of our knowledge, there have not been tools that continually report query progress for subgraph queries.

Demonstration attendees may observe at least three technical challenges for estimating the progress. (i) Real data graphs often contain diverse structures ($e.g.$, [10]). Thereby, the work done in evaluating subgraph queries – in particular, enumerating the isomorphic mappings between the query graph and the data subgraphs – differ greatly among subgraphs. Therefore,

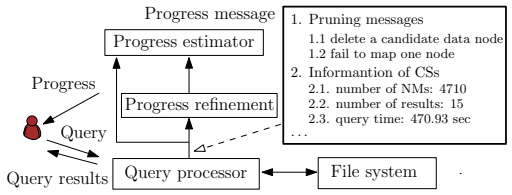

Fig. 2. System overview of PIGEON

it is a daunting task to derive a tight bound and/or cost model of the total work to be done in static time. (ii) There have been a wide variety of querying techniques in the literature for subgraph queries. Intuitively, the progress indicator may capture the core exponential computation embedded in these techniques. Further, the progress estimation may reflect the effectiveness of optimization techniques that prunes unnecessary computation. (iii) The progress estimation should be accurate and have little effect on query performance.

In this demonstration, we take the first step to provide a light-weight and accurate progress indicator called PIGEON (**P**rogress **I**ndicator for Sub**G**raph Qu**E**ry Evaluati**ON**) for subgraph queries. We present our system with the seminal Ullmann's algorithm, which consists of a mapping enumeration step of subgraph queries that captures the computation that runs in exponential time to the sizes of data graphs/queries in worst case. The structure of the Ullmann's algorithm is simple and its enumeration (and its variants) gives rise in all subgraph query algorithms. Our main idea is to extend the algorithm to continuously generate *runtime* information for progress estimation. We define the unit of works of the enumeration called *node mapping* (NM). Based on these, the progress indicator computes the estimated total amount of work and obtains the real work done so far. To tamper the variations discussed, we implement an estimation refinement module to refine estimation as the query is being run.

II. PROTOTYPE OVERVIEW

The system architecture of PIGEON is sketched in Figure 2. The graphs of this demonstration are stored as adjacent lists on a file system. The query user interface is a simple text editor. The query processor is our implementation of the Ullmann's algorithm, extended with a writer of progress messages (the bold rectangle). The prototype does not assume any specific indexes. We remark that we have not made technical assumptions on these modules, which aim to maintain a high portability. The most technically-involved are the notion of *progress*, the *progress messages*, the *progress estimation* module and the *progress refinement* module, whose main ideas are highlighted in Sections III and IV.

III. PROGRESS OF SUBGRAPH QUERIES

Graph and subgraph query. A graph $G = (V, E, \Sigma, l)$ is a 4-ary tuple, where V, E, Σ and l are the set of nodes, edges, labels and the function that returns the label of a node/edge. Given a query graph q, a data graph G and a subgraph isomorphism algorithm $\mathcal{A}$, $\mathcal{A}(q, G)$ is to apply the algorithm on q and G and output the *subgraph isomorphism relations* from q to subgraphs G. Given two graphs $G = (V, E, \Sigma, l)$ and $G' = (V', E', \Sigma', l')$, a *subgraph isomorphic relation* or *valid subgraph mapping* from G to G' is an injective relation S: $V(G) \rightarrow V(G')$ such that

- $\forall v \in V(G), S(v) \in V(G'), l(v) = l'(S(v))$; and
- $\forall (v, v') \in E(G)$, $(S(v), S(v')) \in E(G')$, $l(v, v') = l'(S(v), S(v'))$.

For simplicity, we may often call S a *subgraph mapping* and we call a binary tuple $(v, u) \in S$ a *node mapping* (NM).

A *subgraph query* q on a graph G is to determine all subgraph mappings from q to G. Subgraph query algorithms ($e.g.$, [2], [1]) typically generate the subgraph mapping tuple by tuple. When the mapping is being constructed, these algorithms often localize their processing in *candidate subgraphs* instead of the original graph. Such subgraphs can be formalized as follows. Let $\mathcal{A}$ be a subgraph isomorphism algorithm and $\mathcal{A}(q, G)$ be the query answers. A *candidate subgraph* (CS) of G is a subgraph of G such that $\mathcal{A}(q, \mathsf{CS}(G)) \subseteq \mathcal{A}(q, G)$. To obtain all answers, $\mathcal{A}$ is run on all candidate graphs: $\mathcal{A}(q, G) = \bigcup_{j=1}^{n} \mathcal{A}(q, \mathsf{CS}_j)$, n is the number of CSs on G. That is, since it is well known that the worst case complexity of subgraph queries is exponential to the graph size, it is more efficient to enumerate the mappings from smaller subgraphs. It is worth noting that our techniques do not make any technical assumptions on the algorithms for generating the CSs.

Query progress. We now present the unit of work of subgraph isomorphism algorithms. At the core of those algorithms is an enumeration of possibly exponentially-many isomorphic relations between the query and the data graph. The finest unit of work is to determine a node mapping ($i.e.$, NM), which is a binary tuple containing *a query node and the graph node it is mapped to*[1].

A valid subgraph mapping always contains $|V(q)|$ node mappings. In other words, the total work to be done for enumerating a subgraph mapping is $|V(q)|$. An invalid mapping is smaller than $|V(q)|$. When a partial mapping S' is being processed, the work completed so far is $|S'|$. Then, the *query progress* is defined as the *ratio* of (a) the total number of completed NMs (denoted as K) to (b) the estimated total NMs of all candidate subgraphs (denoted as N) [2].

IV. THE PROGRESS ESTIMATION MODULE

The section presents the main techniques implemented in the *progress estimation* module (sketched in Figure 3). Given a query q, the candidate subgraphs (CSs) can be easily located from the data graph. Each CS contains one or more *possible* subgraph mappings. We employ a generic procedure for locating all CSs and omit specific optimization that minimizes the CSs. Specifically, we first (randomly) choose a start node v_s of q and then locate from the data graph G the candidates that

[1]There are algorithm specific details such as function calls, backtracking and maintenance of data structures that are not included. Instead, we assume that such computation costs are uniformly distributed among the tasks of determining node mappings.

[2]It might appear that when very few mappings were valid, they ed to discrete progresses, with abrupt progress increments. We remark that in such cases, the queries finish almost instantly and the users may not monitor their progresses.

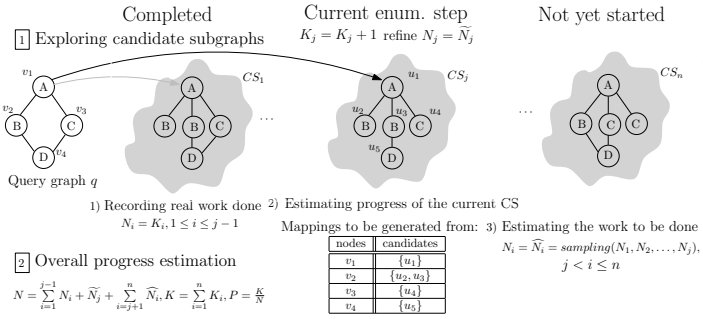

Fig. 3. Progress estimation in PIGEON.

can be mapped to v_s. We then perform a breadth first search on q and each of the candidates simultaneously to obtain a CS from G, until all the edges of q are visited.

At query time, the CSs can be exhaustively categorized into the following (shown in grey at the top of Figure 3): 1) completed its processing; 2) currently being processed; and 3) not yet started its processing.

The number K of completed NMs. The number of completed NMs by CS_j (denoted as K_j) are computed as follows:

1) For a completed CS_j, K_j is known, which equals to the number of NMs determined from CS_j;
2) For the currently being processed CS_j, the completed number of NMs K_j is updated as $K_j = K_j + 1$ whenever an NM is determined; and
3) For a CS_j that is not yet started, K_j is obviously 0.

The estimated number N of NMs. The estimated number of NMs by CS_j (denoted as N_j) are computed as follows:

1) For a completed CS_j, N_j is known, where, $N_j = K_j$;
2) For the currently being processed CS_j, suppose v_i is the i-th node in q being mapped. Denote the node set that v_i is mapped as $\mathsf{CS}_j(v_i)$, *e.g.*, in Figure 3, $\mathsf{CS}_j(v_2) = \{u_2, u_3\}$. The estimated number of NMs related to v_i is denoted as $C_{j,i}$, where
$$C_{j,i} = \prod_{k=1}^{i} |\mathsf{CS}_j(v_k)|. \quad (1)$$
For instance, $C_{j,2} = |\{u_1\}| \times |\{u_2, u_3\}| = 2$. Thus, the estimated number of NMs of CS_j for all query nodes is:
$$\widetilde{N_j} = \sum_{i=1}^{|V(q)|} C_{j,i}; \quad (2)$$
3) For a CS_j that is not yet started, we estimate N_j by the average N value of the completed CSs, denoted as $\widehat{N_j}$.

Putting these together, the progress P is estimated as follows:
$$P = \frac{K}{N} \quad (3)$$
where $K = \sum_{j=1}^{n} K_j$ is the total number of completed NMs, and N is the estimated total NMs, computed as:
$$N = \sum_{i=1}^{j-1} N_i + \widetilde{N_j} + \sum_{i=j+1}^{n} \widehat{N_i}, \quad (4)$$
where CS_j is the CS being processed.

V. The Progress Refinement Module

As NMs are being enumerated, runtime information are used by the *progress refinement module* to continuously adjust the estimated number of NMs $\widetilde{N_j}$ on processed CS_j. In this section, we summarize the techniques of the module.

Again, consider the candidate subgraph CS_j being processed. This module determines if v_i can be mapped to u, $u \in CS_j(v_i)$. There are three possible cases:

1) v_i can be mapped to u;
2) v_i cannot be mapped to u under the current partial subgraph mapping while u may still be mapped to v_i under other mappings in the future; or
3) v_i cannot be mapped to u under the current partial mapping and u is impossible to be mapped to v_i in any possible mappings in the future.

For the first case, the algorithm adds 1 to K_j and continue to map to v_{i+1}. The second and third cases can be a result of pruning optimizations, which lead to less work to be done. For the second case, the algorithm backtracks and maps v_i with other data nodes. Then, we refine $\widetilde{N_j}$ to reflect that v_{i+1},... $v_{|V(q)|}$ do not need to be enumerated at all.

1) Update $C_{j,k}$, $i \le k \le |V(q)|$ as:
$$C_{j,k} = C_{j,k} - D_k, \quad (5)$$
where D_k is the number of NMs skipped after the backtracking, defined as:
$$D_k = \begin{cases} 1 & k = i \\ D_{k-1} \times |CS_j(v_k)| & i < k \le |V(q)| \end{cases} \quad (6)$$
In particular, as v_i cannot map to u, we first substract 1 from $C_{j,i}$. For $C_{j,k}$ where $i < k \le |V(q)|$, D_k is subtracted from $C_{j,k}$ since the algorithm skips those NMs from v_k; and
2) Recompute $\widetilde{N_j}$ according to Eq. (2).

For the third case, we have known that all NMs involving u are not enumerated and this is reflected to $\widetilde{N_j}$ as follows:

1) Update $\mathsf{CS}_j(v_i)$ as:
$$\mathsf{CS}_j(v_i) = \mathsf{CS}_j(v_i) \backslash \{u\} \quad (7)$$
2) Update $C_{j,k}$, where $i \le k \le |V(q)|$ using Eq. (5). Suppose that v_{i-1} maps to the m-th node in $\mathsf{CS}_j(v_{i-1})$, where $1 \le m \le |\mathsf{CS}_j(v_{i-1})|$. D_k is computed as:
$$D_k = \begin{cases} 1 & k = 1 \\ (|\mathsf{CS}_j(v_{k-1})| - m + 1) \times C_{j,k-2} & k = i \neq 1 \\ D_{k-1} \times |\mathsf{CS}_j(v_k)| & i < k \le |V(q)| \end{cases} \quad (8)$$
For $k = 1$ and $i < k \le |V(q)|$, the refinement is the same to Eq. (6). For $k = i \neq 1$, since u is no longer valid when the algorithm backtracks to the remaining nodes in $\mathsf{CS}_j(v_{i-1})$, those corresponding NMs skipped are subtracted. Hence, we substract D_i from $C_{j,i}$, where $D_i = (|\mathsf{CS}_j(v_{i-1})| - m + 1) \times C_{j,i-2}$.
3) Recompute $\widetilde{N_j}$ according to Eq. (2).

VI. Related work & Novelty

In addition to the related work on progress indication discussed in Section I, there is considerable work on selectivity estimation of graph/twig queries on graph/tree data, *e.g.*, [11], among many others. Selectivities are estimated at static time, typically for query optimizers. In comparison, progress is estimated and refined during query time for users. In addition,

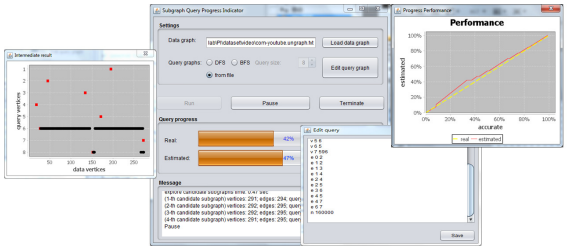

Fig. 4. [Best viewed in color]. The GUI of PIGEON.

there is a host of work on distributed graph processing frameworks ($e.g.$, Pregel [12]). Such frameworks do not yet provide a tool for progress indication of *individual tasks*. There is also a related continual effort on making graph databases more usable and efficient by providing structural search with GUIs [8], [9]. These techniques are orthogonal to PIGEON as they address interleaving visual query formulation and query performance to improve the *system response time*. Such interleaving action does not report any monitoring of query performance during visual query formulation. In summary, to the best of our knowledge, PIGEON is the first system to demonstrate query progress for subgraph queries.

VII. DEMONSTRATION DESCRIPTION

Prototype. We have implemented a demonstration prototype of PIGEON using JAVA. The GUI of PIGEON is shown in Fig. 4. The demonstration consists of datasets that are popular in the literature of graph databases such as *PubChem*, *AIDS*, *Youtube* and other networks from *Stanford Network Analysis Project* (SNAP). A set of example queries will be presented. Attendees may also generate massive random queries using DFS or BFS methods. Moreover, attendees may edit and run their ad-hoc queries. A video of this demonstration can be found in http://youtu.be/rOb5dw26Yvg.

Interactive experience of the progress indicator. The attendees may observe how real progress and estimated progress match (and sometimes, differ) during the query runtime, as shown in the middle of Fig. 4. For demonstration purposes, we obtain the real progress by running the queries prior to the progress estimation. Then, attendees may click the RUN button with the progress indicator. Attendees may pause the query execution at anytime to display the partial query-data graph mapping, as shown by the red dots on the left hand side of Fig. 4. The software also visualizes the matrix that encodes all possible mappings to be enumerated, indicated by the black dots of the same window. From the demonstration, attendees may locate subgraphs that may cause long query runtimes, where the region of the matrix that contains many black dots; inaccurate progress estimation by pausing the query; and/or the significant runtime discrepancies of similar queries by comparing the black dots in their matrices. Further, attendees can observe interactively the refinement of estimated progress during runtime. That is, attendees can experience how is a *dynamic* estimation superior to a static estimation.

Experience of the impact of graph query runtimes with high deviations. Despite an ample body of research on optimization techniques on graph queries, our preliminary study finds that the runtimes of some techniques varied greatly ($e.g.$, [13]), where the average runtime reflects a partial picture of the performance. Attendees may examine a set of sample queries that are structurally similar whose runtimes differ greatly. Attendees may modify the queries to observe how small changes in subgraph queries may cause great impact on query runtimes and how ad-hoc such changes are.

Applications. Finally, we present two scenarios in the demo. *Scenario 1.* Consider a chemist who explores a database of molecules. He/she may start with a small substructure ($e.g.$, a benzene ring) that he/she is familiar and refines it to form more specific queries. The initial query may run long and its progress may be small. The progress reminds the chemist that in order to complete the query, much computation is needed. One reason is that the query is too common and many CSs are identified and enumerated. Hence, he/she may submit a more specific initial query.

Scenario 2. The *Amazon* network contains co-purchase product data. In Fig. 1, q_1 shows a purchase pattern of products 1-4. With the help of the progress indicator, an attendee is informed that q_1 will not finish interactively (say, in 10s). By clicking the *Pause* button, the user may know there are again too many CSs. He/she may modify q_1 to q_2 and q_3 and retrieve some results. However, the results are relatively few. He/she may further formulate q_4. This shows how attendees may interact with PIGEON to analyze the network.

ACKNOWLEDGMENT

X. Xie and S. Zhou were partially supported by the Research Innovation Program of Shanghai Municipal Education Committee under grant no. 13ZZ003. Z. Fan, P. Yi and B. Choi were partially supported by the HKBU grants FRG2/12-13/079 and FRG2/13-14/073. S. S Bhowmick was supported by the Singapore-MOE AcRF Tier-1 Grant RG24/12.

REFERENCES

[1] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD*, 2013, pp. 337–348.
[2] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.
[3] W. S. Han, J. Lee, M. D. Pham, and J. X. Yu, "iGraph: a framework for comparisons of disk-based graph indexing techniques," *PVLDB*, pp. 449–459, 2010.
[4] NCBI, "PubChem," *http://pubchem.ncbi.nlm.nih.gov*, 2012.
[5] S. Wetzel, K. Klein, S. Renner, D. Rauh, T. I. Oprea, P. Mutzel, and H. Waldmann, "Interactive exploration of chemical space with scaffold hunter," *Nature Chemical Biology*, vol. 5, no. 8, pp. 581–583, 2009.
[6] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating progress of execution for sql queries," in *SIGMOD*, 2004, pp. 803–814.
[7] K. Morton, M. Balazinska, and D. Grossman, "Paratimer: a progress indicator for mapreduce DAGs," in *SIGMOD*, 2010, pp. 507–518.
[8] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou, "Quble: towards blending interactive visual subgraph search queries on large networks," *VLDBJ*, vol. 23, no. 3, pp. 401–426, 2014.
[9] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou, "Prague: towards blending practical visual subgraph query formulation and query processing," in *ICDE*, 2012, pp. 222–233.
[10] L. Zhu, B. Choi, B. He, J. X. Yu, and W. K. Ng, "A uniform framework for ad-hoc indexes to answer reachability queries on large graphs," in *DASFAA*, 2009, pp. 138–152.
[11] Y. Peng, B. Choi, and J. Xu, "Selectivity estimation of twig queries on cyclic graphs," in *ICDE*, 2011, pp. 960–971.
[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.
[13] L. Song, Y. Peng, B. Choi, J. Xu, and B. He, "Spectral decomposition for optimal graph index prediction," pp. 187–200, 2013.