Any-k Algorithms for Exploratory Analysis with Conjunctive Queries*

Xiaofeng Yang, Mirek Riedewald, Rundong Li, Wolfgang Gatterbauer†
College of Computer and Information Science, Northeastern University, Boston, MA, USA
{xiaofeng,mirek,rundong,wolfgang}@ccs.neu.edu

ABSTRACT

We recently proposed the notion of *any-k queries*, together with the KARPET algorithm, for tree-pattern search in labeled graphs. Any-k extends top-k by not requiring a pre-specified value of k. Instead, an any-k algorithm returns as many of the top-ranked results as possible, for a given time budget. Given additional time, it produces the next-highest ranked results quickly as well. It can be stopped anytime, but may have to continue until all results are returned. In the latter case, any-k takes times similar to an algorithm that first produces all results and then sorts them. We summarize KARPET and argue that it can be extended to support any-k exploratory search for arbitrary conjunctive queries.

ACM Reference Format:
Xiaofeng Yang, Mirek Riedewald, Rundong Li, Wolfgang Gatterbauer. 2018. Any-k Algorithms for Exploratory Analysis with Conjunctive Queries. In *ExploreDB 2018 : 5th International Workshop on Exploratory Search in Databases and the Web , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3214708.3214711

1 INTRODUCTION

Top-k queries are well-suited for exploratory analysis: by exploiting that the user is interested only in the top-ranked results, query cost can be significantly reduced [2–5]. Unfortunately, it is difficult to set the value for k in practice. Especially for exploratory search, when users try to get a better understanding of the data, they cannot determine in advance when they will have seen enough results.

To address this challenge, we proposed the notion of *any-k queries* in the context of pattern search in labeled graphs [6]. Intuitively, any-k is an anytime ranking algorithm that:

(1) returns the top-ranked result as quickly as possible;
(2) then returns the second-ranked result next, followed by the third-ranked, and so on, until the user terminates the process;
(3) if not stopped, returns all results in a time comparable to an approach that first produces all results and then ranks them.

In other words, the ranked enumeration can be stopped *anytime* and should then return as many top results as possible. Notice the difference to top-k which requires k to specified upfront. While *progressive* top-k algorithms exist, we are not aware of any that (1) can be applied to graph-pattern search and general conjunctive queries and (2) provide strong guarantees for the time-to-first result, time-to-next result, and full result enumeration.

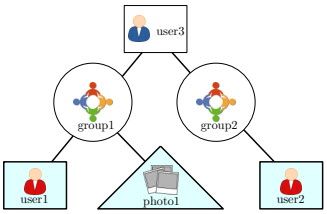

Figure 1: Example query on a photo-sharing network

We first summarize our recent results for any-k pattern search in labeled graphs, then outline how to extend them to conjunctive queries (CQs). CQs are common in databases and data warehousing. Intuitively, a CQ over a set of relations computes a subset of the Cartesian product that satisfies a conjunction of conditions on relation attributes. In Datalog notation, we can express such queries as rules like $Q(x,y,z,u,v) :- R(x,y,z), S(y,u), T(z,v)$. In SQL, assuming schema $R(A,B,C)$, $S(B,D)$, and $T(C,E)$, this query is `SELECT * FROM R, S, T WHERE R.B=S.B AND R.C=T.C`.

2 CURRENT RESULT: ANY-K FOR PATTERN SEARCH IN LABELED GRAPHS

We developed our any-k algorithm, KARPET, in the context of pattern search in labeled graphs and illustrate it here with an example. Refer to Yang et al. [6] for technical details.

Example 2.1 (Photo-sharing network). Consider a photo-sharing social network with three vertex type labels: user, photo, and group. Users are connected to the photos they upload, and photos are connected to groups when they are posted there. Finally, users can connect to groups by joining them. To maintain a vibrant community and alert users about potentially interesting photos, the social network might run queries of the type shown in Figure 1: given *photo1* and two users, *user1* and *user2*, find alternative groups (matching nodes for *group2*) to post the photo in order to reach *user2* without spamming her directly. This is achieved by identifying a user belonging to both groups (*user3*), who can post the photo in the other group. There might be hundreds of matching triples (*group1*, *user3*, *group2*), and there would be many more if *user2* was not given in advance. Under these circumstances, the goal often is not to find *all* results, but only the *most important* ones. Importance can be determined based on node and edge weights. Then the query should return the lightest (or heaviest) pattern instances. For example, the weight of a group may be based on its number of members, the weight of a user on how active s/he is, and the

*Summary paper
†Authors listed in inverse alphabetical order. Be nice to the letter Z!

ExploreDB 2018 , June 15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5847-7/18/06…$15.00
https://doi.org/10.1145/3214708.3214711

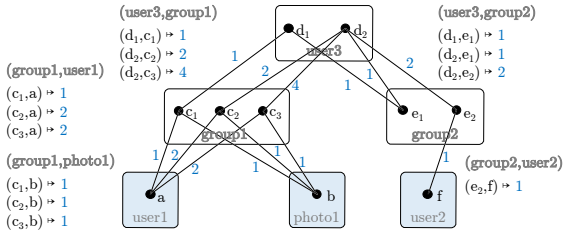

Figure 2: Candidate instances for matching the example query in Figure 1. Edge sets are named based on the corresponding pairs of adjacent nodes in the query pattern.

weight of a link on the timestamp when it was established (to give preference to long-term relationships or more recent photo posts), or the sum of the PageRanks of its endpoints.

Figure 2 shows an example graph for the photo-sharing network. KARPET processes *acyclic* pattern search queries like in the above example by combining three conceptually separate steps into a two-phase algorithm; the implementation can be downloaded from [1]. Here we concentrate on the first two steps only: (1) The search space of possible homomorphic graph patterns is pruned to the provably smallest representation of the original graph. This uses insights from the well-known Yannakakis algorithm [7] for evaluating answers to acyclic conjunctive queries to create this representation in just one bottom-up and a subsequent top-down sweep through the query tree. (2) Our novel any-k algorithm for enumerating homomorphic tree patterns uses dynamic programming to perform a bottom-up cost calculation, followed by a top-down guided search.

KARPET combines the two steps into two phases: 1) a bottom-up sweep from leaves to the root of the query tree, and 2) a top-down depth-first traversal from root to leaves. The first phase prunes some of the spurious candidates and creates a *candidate graph* with *minimum subtree weights*. The second phase prunes the remaining spurious candidates and performs a search guided by the subtree weights. Here the term *spurious candidate* refers to a node or edge of the input graph that does not appear in any of the subgraph-homomorphism query results.

Bottom-Up Phase. The bottom-up phase processes a query node only after all its children have been processed, constructing a candidate graph consisting of two index structures: (1) $\texttt{CandNode}(u)$ returns for query node u a hash index that maps a node candidate c of u to a list of minimum subtrees, each corresponding to a child node u'. For each minimum subtree, its weight and root node are both stored. (2) $\texttt{CandEdge}(u, u')$ returns for each query edge between a node u and its child u' a hash index that maps a candidate node c of u to all adjacent candidates c' of u'.

We illustrate the algorithm with Figures 3a, 3b, and 3c. It first inserts candidate nodes for each query leaf node u into the corresponding candidates $\texttt{CandNode}(u)$, setting their weights to zero. In Figure 3a there is a single candidate per leaf, but in practice it can be a larger subset of graph nodes, depending on the node constraints. Then, for each query node u, the algorithm (*i*) finds possible candidate nodes, (*ii*) prunes them, and (*iii*) calculates the minimum subtree weights.

In more detail: (*i*) for each query edge (u, u') to a child u', it first finds all candidate edges (c, c'), storing the map $\texttt{CandEdge} : (u, u') \mapsto [c \mapsto c']$. (*ii*) Then, the algorithm only keeps the list of candidates for each query node that are reachable from candidate instances in *all* leaves of the query node: In Figure 3c, the list of candidates for query node *group1* is $\{c_1, c_2, c_3\}$. Notice how spurious candidates not reachable from the leaves, e.g., e_1 in *group2*, are not even accessed (compare with Figure 2). Similarly, while d_1 in *user3* is reachable from the left, it is not reachable from the right subtree and is thus automatically pruned as well. (*iii*) Then, the algorithm finds for each reachable node, the min weight along each query edge (u, u') starting at c. For example, in Figure 3c, the left weight 5 for c_2 is computed as the minimum of weights for following (d_2, c_2), which is 5 as the sum of the weight of edge (d_2, c_2) (= 2) plus the weight of c_2 (= 2+1); or for following (d_2, c_3), which is 7 as the sum of the weight of edge (d_2, c_2) (= 4) plus the weight of c_3 (= 2+1). The latter is obtained from CandNode by looking up the entry for query node *group1* and candidate node c_3. The two newly created indices speed up finding adjacent edges in a subtree of the query pattern during top-down traversal.

Top-Down Phase. The second part of our algorithm performs top-down search, starting at the root node and proceeding downward to the leaves. This is essential for two reasons: First, the pre-computed subtree weights guide the search to the lightest patterns before exploring the heavier ones. Second, the top-down traversal implicitly prunes *all* remaining spurious candidates for sub-graph homomorphism. Again, pruning actually happens implicitly by not reaching those candidates. To see the latter, consider *group1* candidate c_1 in Figure 3c. It is spurious, but could not be removed by the bottom-up sweep. However, it will never be accessed during top-down traversal, because d_1 was never recorded in CandNode.

Initially, all candidates in the query root node are inserted into a priority queue pq, with their priorities set to the sum of the candidate's weights. In Figure 3c, there is a single candidate, d_2, of weight 5 + 3 = 8. Then the algorithm repeatedly pops the top element from pq and expands the partial pattern using pre-order traversal. The priority value of each expanded partial match is defined as the sum of the pattern's edge weights plus the sum of the weights of the unexplored subtrees. In the example, partial match (d_2, c_2) is inserted into pq with priority 8 = 2 (edge weight) + (2+1) (weights of c_2) + 3 (weight of right subtree of d_2). Similarly, partial match (d_2, c_3) is inserted with priority 4+(2+1)+3 = 10. (Those values are updated incrementally during traversal.) Then (d_2, c_2) is popped next, and expanded to partial match (d_2, c_2, a) with priority 8. This pattern is expanded next to (d_2, c_2, a, b), (d_2, c_2, a, b, e_2), and finally (d_2, c_2, a, b, e_2, f)—all with the same priority of 8. The latter is output as the minimal-weight solution. Only then will partial match (d_2, c_3) with the higher priority value 10 be expanded analogously.

Summary of Algorithm Properties. The cost of the bottom-up sweep and weight computation is linear in the product of graph size and query size. For the top-down traversal, we prove that for each *final result tuple*, there is *at most* one push and *at most* one pop operation on priority queue pq. This establishes an upper bound on space complexity equal to full result size R, i.e., orders of magnitude smaller than the combinatorial space of possible partial

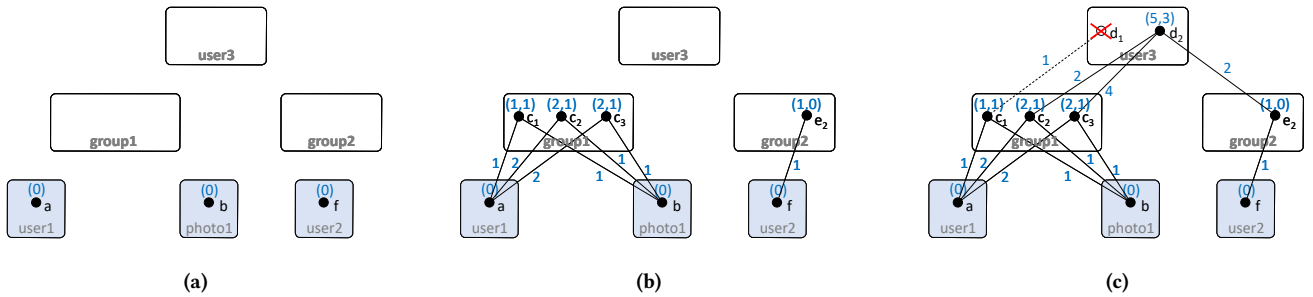

Figure 3: Minimal subtree weight computations: (a) after traversing all leaves, (b) after traversing middle level, (c) after finishing at the root. Numbers above node candidates indicate minimum sub-tree weights stored in CandNode; numbers on edges indicate edge weights.

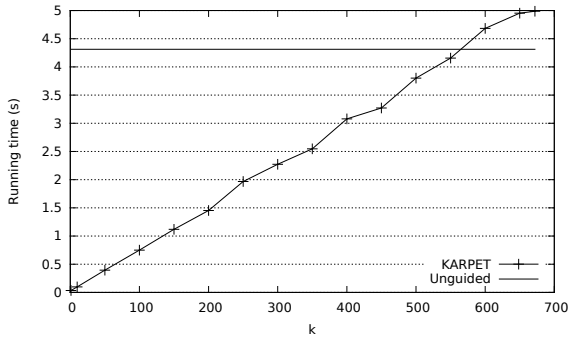

Figure 4: Running time for increasing k until all results are returned. The "Unguided" algorithm is optimized for full enumeration.

pattern matches. We also prove for subgraph homomorphism that the initial latency of the top-down phase to return the top result, and also the wait time between results, is $O(\text{outDegree} + \log R)$. Here outDegree $\leq R$ is the greater of (1) the number of candidates in the root node and (2) the maximum cardinality of the set of adjacent children for any node candidate in CandEdge.

Figure 4 shows a representative observation for a 5-edge query pattern on a real graph with millions of edges. As desired, KARPET starts returning results almost immediately, while the full enumeration incurs a long delay for producing the entire result at once. Our approach pays a small penalty for the any-k property, even when all results are requested.

3 FUTURE WORK: ANY-K FOR ARBITRARY CONJUNCTIVE QUERIES

We argue that the notion of an any-k algorithm can be extended for exploratory search with any given conjunctive query. Notice that KARPET can be applied with minimal changes to support any *acyclic* conjunctive query over *binary* relations, i.e., relations with exactly two attributes. Intuitively, we fold the content of CandNode into CandEdge to create a hash index similar to CandEdge on the join attributes for each input relation.

N-ary relations. It should be straightforward to extend the algorithm to N-ary relations. Whenever two relations join on multiple attributes, the corresponding index hashes on all these attributes.

Queries with cycles. For acyclic queries, our two-phase algorithm relies on global consistency properties of the Yannakakis algorithm. With cycles, we need to consider different tree decompositions.

Parallelization. We can reduce time for returning the first result, and also result-to-result, through parallelization. This can be achieved by extending index CandNode so that it maps a candidate node not only to the subtree weight, but also to the child node candidate that determined this minimal weight. Then the main thread keeps expanding this "winning" candidate, while all other branches are pushed to pq by another thread. This requires careful synchronization whenever a new element is popped off the queue.

Optimality results. We are interested in developing optimality results for any-k queries where ranking between results is determined by any monotone function over the answer tuples. Here we compare the time it takes the any-k algorithm to return the top-1 answer to the best known worst-case time complexity of the corresponding boolean query, i.e., to answer the question if the query has any answers. Similarly, enumerating all answers should match the complexity of the best known algorithm for full enumeration, i.e., an algorithm designed for efficiently computing the entire result set.

Acknowledgments. This work was supported in part by the National Institutes of Health (NIH) under award number R01 NS091421 and by the National Science Foundation (NSF) under award number CAREER III-1762268. The content is solely the responsibility of the authors and does not necessarily represent the official views of NIH or NSF. We would also like to thank the reviewers for their constructive feedback.

REFERENCES

[1] Any-k: anytime top-k pattern retrieval in labeled graphs (code) https://github.com/northeastern-datalab/any-k-karpet, 2018.
[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pp. 495–506, 2007.
[3] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
[4] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.
[5] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pp. 281–290, 2001.
[6] X. Yang, D. Ajwani, W. Gatterbauer, P. K. Nicholson, M. Riedewald, and A. Sala. Any-k: Anytime top-k tree pattern retrieval in labeled graphs. In *WWW*, pp. 489–498, 2018. Full version available on https://arxiv.org/abs/1802.06060.
[7] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pp. 82–94, 1981.