

Subgraph Query Generation with Fairness and Diversity Constraints

Hanchao Ma

Case Western Reserve University

hxm382@case.edu

Sheng Guan

Case Western Reserve University

sxg967@case.edu

Mengying Wang

Case Western Reserve University

mxw767@case.edu

Yen-shuo Chang

Case Western Reserve University

yxc1425@case.edu

Yinghui Wu

Case Western Reserve University

Pacific Northwest National Laboratory

yxw1650@case.edu

Abstract—This paper studies the problem of subgraph query generation with guarantees on both diversity and group fairness. Given a query template (with parameterized search predicates) and a set of node groups in a graph, it is to compute a set of subgraph queries that instantiate the query template, and each query ensures diversified answers that meanwhile covers each group with desired number of nodes. Such need is evident in web and social search with fairness constraints, query optimization, and query benchmarking. We formalize a bi-criteria optimization problem that aims to find a Pareto optimal set of query instances in terms of diversity and fairness measures. We show the problem is in Δ_2^P and verify its hardness (NP-hard and fixed-parameter tractable). We provide (1) two efficient algorithms that can approximate Pareto optimal set with ϵ -dominance relation that yield representative query instances with a bounded size, and (2) an online algorithm that progressively generates and maintains fixed-size ϵ -Pareto set with small delay time. We experimentally verify that our algorithms can efficiently generate queries with desired diversity and coverage properties for targeted groups.

I. INTRODUCTION

Subgraph queries have been routinely used in applications such as social search [27] and knowledge search [36]. Given a graph G and a subgraph query $Q(u_o)$ with a designated output node u_o , it computes a set of nodes (matches) of u_o in G . A number of algorithms [8] have been developed to process subgraph queries in terms of subgraph isomorphism or its approximate variants [36].

The emerging need for data systems that requires both result diversity and fairness [7], [3], [29], [14] poses new challenges to graph querying. In such scenarios, queries are expected to return diversified matches that meanwhile ensure a required coverage of designated groups (node sets) of interests from G . Such groups may refer to the population of vulnerable social groups which are characterized in terms of sensitive attributes (*e.g.*, gender, race, professions) [17], relevant articles yet with diversified labels for Web exploration [2] and recommendation engines [18], or designated columns for query benchmark [5].

Consider the following real-world search scenarios.

Example 1: Talent search. A talent search over a professional network G [18] finds strong candidates with desired skills. Each node in G denotes a user with attributes such as *title*,

skill, *profession*, or an organization with attributes such as the number of employees. Each edge indicates the affiliation (*worksAt*) of a user or recommendation (*recommend*) between users. A recruiter issues a graph search query q_1 (illustrated in Fig. 1) to find directors u_o who have expertise in managing IT business, and moreover, recommended by at least two users from large companies. In our test (Section VI), this query returns a set of qualified candidates $q(G)$, yet with a skewed distribution of 375 male users and 173 female users.

The recruiter pursues a desired gender distribution and diversity of the candidates, and wonders how to revise the search such that (1) the new answer can properly cover $q(G)$ with an equal number of male and female candidates (*e.g.*, both with 200 candidates; a case of “Equal opportunity” [18]); and (2) the candidates are also more diversified in their majors. A more desirable query q_2 can be suggested, which finds 202 male and 198 female candidates that span 10 majors. The difference between q_1 and q_2 suggest that a relaxed condition on recommendation (removing the edge from u_1 to u_3) and a relaxation to recommends from smaller business (reducing ‘1000’ employees to ‘500’ employees) help to achieve the desired answer with proper coverage of the gender groups. \square

It is desirable to suggest such queries with diversified answers that can meanwhile cover a set of designated groups with desired cardinalities. Moreover, there may exist multiple queries with “better” answers in terms of either diversity or equal opportunity.

Example 2: Continuing the talent search, two more queries q_3 and q_4 (Fig. 1) can be found by “perturbing” the ranges of the years of experiences of recommenders and the number of employees they work at. (1) Compared with q_1 , both queries can identify more diversified candidates with less skewed distribution on genders. (2) Compared with q_2 , both finds a more skewed results over gender groups, yet each finds more diversified candidates with more than 30 majors. While all three queries provide more “desired” answers in terms of diversity or promoting equal opportunity, it remains a daunting task for users to inspect all these queries. \square

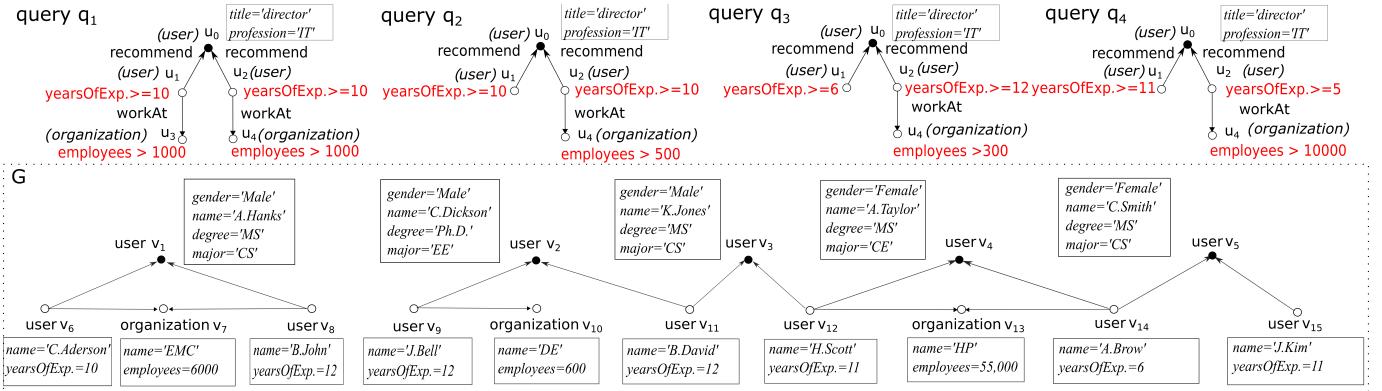


Fig. 1. Subgraph queries with Diversity and Group Fairness: Talent Search

This calls for efficient algorithms that can suggest representative subgraph queries with desirable guarantees on both diversity and coverage, over specified groups of interests. The need is evident in social search and recommendation with group fairness [18], workload generation for query benchmark [5], and query optimization [23].

This paper studies a novel problem called *subgraph query generation with diversity and fairness constraints* (FairSQG), which has a general form below:

- **Input:** graph G , an initial query (template) $Q(u_o)$, and a set of groups \mathcal{P} , where each group $P_i \in \mathcal{P}$ is associated with a coverage constraint c_i ($c_i \leq |P_i|$);
- **Output:** a set of subgraph queries \mathcal{Q} obtained by revising $Q(u_o)$; each query can retrieve a set of diversified matches (“Diversity”) from G that also cover each group P_i with desired cardinality c_i (“Group fairness”).

Several methods have been developed to generate graph queries that lead to desired answers. Notable examples include query suggestion with diversified answers [28], [20], coverage of similar counterparts of “examples” [34], [30], or cardinality constraints on output sizes [24], [4]. These approaches are designed to revise queries towards specific properties rather than ensuring both group fairness and answer diversity, thus cannot be directly applied to our problem.

Contributions & Organization. This paper formally analyzes the subgraph query generation problem with group fairness constraints. We propose both feasible approximation schemes as well as practical exact algorithms for large graphs. We refer to subgraph query simply as “query” in the rest of the paper.

A Bi-objective formalization. We provide a practical formalization of the FairSQG problem (Section III). (1) We introduce a class of *query templates* (denoted as $Q(u_o)$). A template carries variables defined on search predicates and edges yet to be instantiated at processing time. Query generation yields value binding to the variables to produce a set of query instances. (2) We introduce diversity and fairness measures to measure the quality of an instance. Despite the need of optimizing both, they may come in conflict: a single optimal instance may not exist. On the other hand, a complete Pareto set is of substantial size for users to inspect. Instead, we

introduce a bi-objective optimization problem to compute an ϵ -Pareto set of instances based on a query dominance relation. ϵ -Pareto set is a desirable subset approximation of the Pareto optimal set that strikes a balance between dominance relation and the number of instances to be returned.

We show FairSQG is solvable in Δ_2^P (a class of P^{NP} problems with an NPooracle) for templates with fixed variables, and show its hardness varies from PTIME to NP-hard if Q has fixed size and variable sizes, and for templates without range variables. These results verify useful upper and lower bound results for query generation scenarios in practice.

Query generation with quality guarantees (Section IV). Generating a Pareto set of substantial size is often not desirable. We first introduce algorithms that can approximate the exact Pareto set \mathcal{Q} with quality guarantee controlled by an error bound ϵ . The algorithm ensures to find a subset of \mathcal{Q} , denoted as \mathcal{Q}_ϵ , such that for each possible instance of Q_{u_o} , there is an instance in \mathcal{Q}_ϵ that approximately dominant it on both diversity and coverage within a constant factor $\epsilon > 0$. Better still, the algorithm ensures to return a representative query set with a bounded size. We also introduce optimization strategies to reduce the cost of query generation.

Online maintenance of fixed-sized set (Section V). Our analysis shows that one often needs to sacrifice query instance quality (ϵ) in trade for a smaller, representative set to inspect. We follow up with an online algorithm, which progressively constructs and incrementally maintains a ϵ -Pareto set with k instances and an ϵ as small as possible. The online algorithm uses a sliding window strategy to dynamically swap or replace queries, incrementally updates ϵ -Pareto set only when necessary, and incurs a small delay time.

Real-world evaluation and case analysis (Section VI). Using real-life graphs, we verify the effectiveness and efficiency of our algorithms (Section VI). We show that our algorithms can generate subgraph queries with both desired diversity and small errors in covering designated groups. These algorithms are also feasible. For example, it takes 78 seconds to produce instances with desired coverage in real-life graphs with 30 million nodes and edges. We also illustrate that our algorithms can generate favorable queries for different user preferences.

Related Work. We categorize the related work as follows.

Graph query suggestion. Several methods have been studied to suggest subgraph queries towards answers with various desired properties. Graph query by example [20] induces subgraph queries from subgraphs that contain similar nodes to specified examples. Diversified query suggestion [28] expands an initial query with new edges that can lead to relevant and diversified matches. Why-questions [34], [30] suggest queries with both relaxation and refinement operators towards exact or similar matches. These methods cope with a single objective or weighted combination of diversity, similarity or output size constraints on the entire match set. Bi-objective query suggestion in terms of both diversity and group fairness is not addressed by prior methods. Diversified workload generation with fairness has been studied [26], but with a different setting where the goal is to generate a query load with the union of the answers cover a desired fraction of a group.

Set selection with group fairness. Subset selection with diversity and fairness constraints has been studied [35], [29]. Given a universal set and a set of groups (subsets), it computes a diverse subset that can cover each group with individual cardinality constraints. Approximation algorithms have been studied to generate subsets for max-sum and max-min diversification [29]. These methods study set coverage properties and cannot be directly used to suggest graph queries. Our formal analysis establishes the hardness of subgraph query suggestion with group fairness, which is a more general counterpart.

Skyline search. Multiobjective search such as skyline queries [10], [11] has been extensively studied. Our problem can be considered as computing a representative bi-objective skyline front in subgraph query space with diversity and coverage preferences. Existing skyline algorithms are optimized to compute Pareto optimal sets and their variants over a set of tuples. Our problem is more involved due to the expensive query processing cost in terms of subgraph isomorphism (an NP-hard problem). It is also not desirable to return a large Pareto set [10] for practical query generation scenarios. ϵ -Pareto set [31], [22] has been studied as a desirable approximation for Pareto optimal set. We advocate feasible query generation with such approximation, and introduce practical algorithms that can (1) efficiently generate and maintain ϵ -Pareto instance sets for large graphs, and (2) strike a balance between the solution quality and size.

Query workload generation. Query generation with output size constraints and distribution properties have been investigated for graph benchmarking and query optimization. [4] generates regular path queries from a given schema that can output answers with required cardinalities when projected to pre-defined attributes. [23] generates SPARQL queries that can cover the answer of given queries with cheaper plans for multi-query optimization. In contrast to these works, (1) We study query generation under fairness constraints for arbitrary groups with guaranteed coverage requirement; and (2) We consider queries defined in subgraph isomorphism with search

predicates, a more involved query class compared with regular path queries. Our algorithms can be readily applied to generate queries for benchmarking graph databases.

II. GRAPH, QUERY TEMPLATES AND INSTANCES

Graphs. We consider directed graphs $G = (V, E, L, T)$, where (1) V is a finite set of nodes, (2) $E \subseteq V \times V$ is a set of edges, (3) each node $v \in V$ (resp. edge $e \in E$) carries a label $L(v)$ (resp. $L(e)$); and (4) each node v carries a tuple $T(v) = \langle (A_1, a_1), \dots, (A_n, a_n) \rangle$, where each A_i ($i \in [1, n]$) is a distinct node attribute with a value a_i .

We denote the finite set of all the node attributes in G as \mathcal{A} . The *active domain* $\text{adom}(A)$ of an attribute $A \in \mathcal{A}$ refers to the set of values of $v.A$ as the node v ranges over V .

Query Template. A *query template* (or simply “template”) $Q(u_o)$ is a connected graph (V_Q, E_Q, L_Q, T_Q) , where V_Q (resp. $E_Q \subseteq V_Q \times V_Q$) is a set of query nodes (resp. query edges). Each query node $u \in V_Q$ (resp. query edge $e \in E_Q$) has a label $L_Q(u)$ (resp. $L_Q(e)$). Specifically, there is a designated *output node* $u_o \in V_Q$.

Variables. A template allows “placeholders” in search predicates that can be bound to specific values when executed. It extends parameterized queries [9] for graph query generation. We consider two types of variables in a template $Q(u_o)$. (a) For each node $u \in V_Q$, $T_Q(u)$ is a set of literals. A literal l is in the form of $u.A \text{ op } x_l$, where op is from $\{\text{>}, \text{>}=, \text{=}=\text{, <}=, \text{<}\}$, and x_l is a *range variable* that can be assigned to a constant. (b) For each edge $e \in E_Q$, $T_Q(e)$ is a Boolean *edge variable* x_e (either ‘0’ or ‘1’). The set of all the variables in $Q(u_o)$ is denoted as $\bar{X} = \bar{X}_L \cup \bar{X}_E$.

Query Instances. Given a template $Q(u_o)$, an *instantiation* of $Q(u_o)$ is a function I , such that for each variable $x \in \bar{X}$, $I(x)$ is either a constant or a wildcard ‘_’. A *query instance* (or simply “instance”) $q(u_o)$ of $Q(u_o)$ induced by an instantiation I is a connected graph (V_q, E_q, L_q, T_q) with the same V_Q , output node u_o and L_Q , and moreover,

- o for each literal $l \in T_Q(u)$ in $Q(u_o)$, if $I(x_l)$ is a constant, then there is a literal $l = u.A \text{ op } I(x_l)$ in $T_q(u)$; and
- o there is an edge $e \in E_Q$ if and only if (a) $I(x_e) = '1'$, and (b) e is in the same connected component of u_o .

An instance q of $Q(u_o)$ contains no variables but only literals and the edges in the connected component where u_o resides, induced by the constant binding from I . We denote the set of all the possible instances of $Q(u_o)$ as $\mathcal{I}(Q)$.

Matches. Given an instance $q(u_o)$ and a graph G , a matching from $q(u_o)$ to G is a function $h \subseteq V_q \times V$, where (1) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$, and for each literal $u.A \text{ op } c$ in L_q , $h(u).A \text{ op } c$; and (2) for each edge $e = (u, u')$ in $q(u_o)$, $h(e) = (h(u), h(u'))$ is an edge in G , and $L_Q(e) = L(h(e))$.

The *matches* of a query node u of $q(u_o)$ in G , denoted as $q(u, G)$, refers to the set of all the nodes in G that can match node u via a matching $h(u)$ from q to G . The *result* of q in G , denoted as $q(G)$, refers to the match set $q(u_o, G)$.

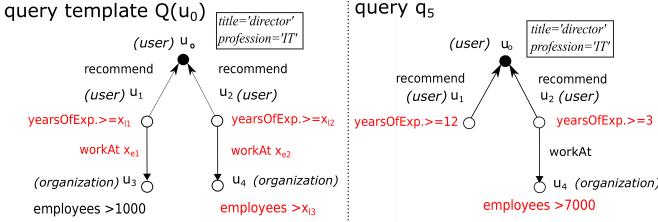


Fig. 2. Query Template and Query Instance

Example 3: A template $Q(u_o)$ that searches for directors in a graph G [18] is illustrated in Fig. 1. (1) $Q(u_o)$ has five variables $\bar{X} = \{x_{l1}, x_{l2}, x_{l3}, x_{e1}, x_{e2}\}$, with three range variables in literals $u_1.\text{yearsOfExp.} \geq x_{l1}$, $u_2.\text{yearsOfExp.} \geq x_{l2}$, and $u_4.\text{employees} \geq x_{l3}$ respectively, and edge variables x_{e1}, x_{e2} . (2) A corresponding instantiation $\{10, 10, 1000, '1', '1'\}$ (resp. $\{10, 10, 500, '0', '1'\}$, $\{6, 12, 300, '0', '1'\}$, $\{11, 5, 1000, '0', '1'\}$) of \bar{X} induces an instance q_1 (resp. q_2 and q_3 and q_4) of $Q(u_o)$. (3) Given G , $q_1(G) = \{v_1\}$, $q_2(G) = \{v_1, v_2, v_3\}$ and $q_3(G) = \{v_1, v_2\}$, $q_4(G) = \{v_3, v_4, v_5\}$. \square

Remarks. The instances are well-defined for a “partial” instantiation in which some variables are assigned a wildcard ‘_’ (“don’t care”). For such case, q is induced by removing corresponding parameterized predicates or edges to ensure valid $q(G)$. A user-defined “initial” query (e.g., q_1 in Example 1) can be captured by a template with a partial instantiation.

III. QUERY GENERATION PROBLEM

Given a template $Q(u_o)$, graph G and m disjoint node groups \mathcal{P} in G , where each group $P_i \in \mathcal{P}$ has a cardinality constraint $c_i \in [0, |P_i|]$, the query generation problem aims to compute a set of instances $\mathcal{Q} \subseteq \mathcal{I}(Q)$ of Q with maximized diversity and required coverage properties.

A. Quality Measures

We consider two functions to quantify the “goodness” of instances in terms of diversity and fairness.

Diversification. We consider Max-sum diversity as a natural objective for result diversification [19]. Given an instance q and G , the diversity of q is defined as:

$$\delta(q, G) = (1 - \lambda) \sum_{v \in q(G)} r(u_o, v) + \frac{2\lambda}{|V_{u_o}| - 1} \sum_{v, v' \in q(G)} d(v, v')$$

where (1) $\lambda \in [0, 1]$ is a constant to balance relevance and diversity; (2) the function $r(u_o, v) \in [0, 1]$ (resp. $d(v, v') \in [0, 1]$) computes a relevance score between u_o and a match v (resp. difference between two matches v and v'). In practice, $d(v, v')$ can be the edit distance between tuples $T(v)$ and $T(v')$ [21], and $r(u_o, v)$ can be an entity linkage score or impact of v in social networks [15].

Here V_{u_o} is the set $\{v | L(v) = L(u_o), v \in V\}$, i.e., the nodes in G with the same label of u_o . Given G , the pairwise dissimilarity is normalized with a constant $\frac{|V_{u_o}| - 1}{2}$, as there

are at most $\frac{|V_{u_o}|(|V_{u_o}| - 1)}{2}$ pairs but $|V_{u_o}|$ relevance numbers. That is, $\delta(q, G) \in [0, |V_{u_o}|]$.

Coverage. Ideally, an instance should satisfy the coverage requirement c_i posed on each group $P_i \in \mathcal{P}$, and cover exactly c_i nodes in each $P_i \in \mathcal{P}$. Given \mathcal{P} , G and template Q , an instance is *feasible*, if for each $P_i \in \mathcal{P}$, $|q(G) \cap P_i| \geq c_i$.

We next introduce a function to quantify the quality in terms of desired coverage as:

$$f(q, \mathcal{P}) = C - \sum_{P_i \in \mathcal{P}} (|q(G) \cap P_i| - c_i)$$

where the constant $C = \sum_1^{|\mathcal{P}|} c_i$. The function penalizes the total accumulated errors between desired coverage and actual counterpart by $q(G)$ over each group in \mathcal{P} . The larger $f(q, \mathcal{P})$ is, the better ($f(q, \mathcal{P}) \in [0, C]$).

Example 4: Continue with the talent search queries in G as illustrated in Fig. 1. In a simple scenario where we want to cover exactly 2 male and 2 female over the qualified candidates, one may verify the following: $\delta(q_1, G) = 0$ and $f(q_1, \mathcal{P}) = 1$; $\delta(q_2, G) = 1$ and $f(q_2, \mathcal{P}) = 1$; and $\delta(q_3, G) = 0.75$ and $f(q_3, \mathcal{P}) = 2$, and $\delta(q_4, G) = 0.5$ and $f(q_3, \mathcal{P}) = 3$. \square

In the rest of the paper, we only consider feasible instances. We shall denote $\delta(q, G)$ and $f(q, \mathcal{P})$ simply as $\delta(q)$ and $f(q)$, respectively, when G and \mathcal{P} are specified in the context.

B. Query Generation Problem

Pareto Optimality. Given G , \mathcal{P} and a template Q , an “optimal” instance q^* should maximize both diversity and relative coverage, (i.e., a Pareto optimal instance):

$$q^* = \arg \max_{q \in \mathcal{I}(Q(u_o))} \delta(q); \quad q^* = \arg \max_{q \in \mathcal{I}(Q(u_o))} f(q)$$

While desirable, such a solution may not always exist, as diversity (which favors instances with diversified matches) and group fairness (which requires desired coverage) may be in conflict. A proper option is to compute a Pareto set. Given two instances q and q' in $\mathcal{I}(Q)$, we say q dominants q' , if either (1) $\delta(q) \geq \delta(q')$ and $f(q) > f(q')$, or (2) $\delta(q) > \delta(q')$ and $f(q) \geq f(q')$. A set $\mathcal{Q}^* \subseteq \mathcal{I}(Q)$ is a *Pareto instance set* if (1) there is no pair of instances (q, q') from \mathcal{Q}^* , such that q dominants q' , and (2) for any instance $q'' \in \mathcal{I}(Q)$, there exists an instance $q \in \mathcal{Q}^*$ that dominants q'' .

A Pareto instance set \mathcal{Q}^* provides solutions for users with different preferences on diversity and fairness in query generation. We show it is well defined for query generation.

Lemma 1: Given a template $Q(u_o)$, graph G and groups \mathcal{P} , there exists a unique, finite and maximum Pareto set \mathcal{Q}^* . \square

Nevertheless, the exact \mathcal{Q}^* is often of substantial size in practice. For a small template of 3 edges and 3 variables, a complete Pareto set in real-world graphs may already contain 100 instances (see Section VI), a daunting task for users to inspect. We next introduce a notion of ϵ -Pareto instance

set, which approximates \mathcal{Q}^* with quality guarantee, and is of polynomially bounded size.

ϵ -Pareto instance set. Given Q , G , and \mathcal{P} , we say an instance q ϵ -dominants q' for some $\epsilon > 0$, denoted as $q \succ_\epsilon q'$, if and only if $(1 + \epsilon)\delta(q) \geq \delta(q')$, and $(1 + \epsilon)f(q) \geq f(q')$.

We show the following properties of ϵ -dominance relation.

Lemma 2: *Given template Q and G , the ϵ -dominance defined on $\mathcal{I}(Q)$ is a preorder.* \square

Lemma 2 can be verified by observing that ϵ -dominance is reflexive and transitive on query instances.

A set of instances $\mathcal{Q}_\epsilon^* \subseteq \mathcal{I}(Q(u_o))$ is an ϵ -Pareto instance set, if (a) $\mathcal{Q}_\epsilon^* \subseteq \mathcal{Q}^*$, and (b) for any instance $q' \in \mathcal{I}(Q)$, there exists an instance $q \in \mathcal{Q}_\epsilon^*$, such that $q \succeq_\epsilon q'$.

An ϵ -Pareto instance set is desirable: it not only ϵ -dominants all the instances in $\mathcal{I}(Q)$, with a configurable quality controlled by ϵ , but also contains instances from the Pareto instance set \mathcal{Q}^* . That is, it approximates \mathcal{Q}^* with a subset of representative but less number of instances. On the other hand, there exist multiple such ϵ -Pareto instance sets.

We next state the FairSQG problem.

Problem statement. A *configuration* of query generation is a tuple $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$, which contains a graph G , template $Q(u_o)$, disjoint groups \mathcal{P} with coverage constraints, and a constant $\epsilon > 0$. Given a configuration $(G, Q(u_o), \mathcal{P}, \epsilon)$, the FairSQG problem is to compute an ϵ -Pareto instance set \mathcal{Q}_ϵ .

Example 5: Given the $\mathcal{I}(Q)$ that contains q_1, q_2, q_3 and q_4 , the Pareto set of $I(Q)$ is $\{q_2, q_3, q_4\}$ since q_1, q_2 and q_3 all dominant q_1 . Here, we set $\epsilon = 0.3$. Given the δ and f values of these query instances (See 4). We can easily compute the “boxing” coordinates of $\mathcal{I}(Q)$. From q_2 to q_4 , “boxing” coordinates are $\{2.0, 2.0\}(2.0, 4.0)(1.0, 5.0)\}$, respectively. Then, the ϵ -Pareto instance set of $\mathcal{I}(Q)$ is $\{q_2, q_3, q_4\}$. We can see that q_3 and q_4 can not dominate each other, however, q_3 dominants q_2 with the “boxing” coordinates. Thus, q_2 will be removed from Pareto Set of $\mathcal{I}(Q)$ to form the ϵ -Pareto set which is $\{q_3, q_4\}$. \square

Although desirable, FairSQG remains nontrivial. We provide the following upper and lower bound analysis.

Theorem 3: *Given a configuration $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$, the FairSQG problem (1) is in Δ_2^P when Q has a fixed number of variables $|\hat{X}|$, (2) remains NP-hard when Q has no range variables, and (3) is fixed-parameter tractable, for Q with fixed size (number of edges) and fixed $|\hat{X}|$.* \square

Proof sketch: The decision problem of FairSQG is to determine whether there is a non-empty ϵ -Pareto instance set \mathcal{Q}_ϵ . (1) FairSQG is solvable in Δ_2^P for fixed template Q . Here Δ_2^P is the class of problems in P^{NP} . A Δ_2^P algorithm first enumerates $\mathcal{I}(Q)$. For each instance, it consults an NPoracle to verify $f(q)$ and $\delta(q)$, and computes an ϵ -Pareto set with a pairwise comparison (e.g., nested loop). As

$|\mathcal{I}(Q)| \leq 2^{|X_e|} |\text{adom}_m|^{|X_L|}$ ($|X_e|$ and $|X_L|$ are constants), the verification is in PTIME. Here adom_m refers to the largest active domain in G . (2) The NP-hardness can be verified from the hardness of deciding subgraph isomorphism, even when X_e is fixed (thus in total $2^{|X_e|}$ instances).

To see (3), we observe that the Δ_2^P algorithm in (1) takes polynomial time when both $|\hat{X}|$ and $|Q|$ are fixed, given that it takes PTIME to verify the coverage and diversity for $\mathcal{I}(Q)$ with polynomially bounded size. Detailed proofs are in [1]. \square

The above analysis provides a naive algorithm (denoted as EnumQGen): enumerates up to $2^{|X_e|} |\text{adom}_m|^{|X_L|}$ instances, verifies each instance to find feasible ones, and invokes a nested loop comparison to generate ϵ -Pareto instance set. This is infeasible when G is large. We next show that an ϵ -Pareto instance set \mathcal{Q}_ϵ^* with a *bounded size* can be efficiently computed (Section IV) and dynamically maintained (Section V), where the size bound is only determined by ϵ and the range of diversity and coverage. This enables flexible query generation that strikes a balance between quality and instance sizes.

IV. APPROXIMATING PARETO INSTANCE SETS

We present a generic query generation algorithm, denoted as QGen, for FairSQG without enumeration. We start with the auxiliary structures it maintains.

Auxiliary structures. To characterize the search space, we start with a notion of *refinement* relation defined on $\mathcal{I}(Q)$.

Instance Refinement. Given a template Q and instantiations I and I' of Q , I' refines I at a variable x (denoted as $I' \succeq_I$) if it binds a constant to x that makes the predicate parameterized by x no less selective than the counterpart from I . Specifically, (1) for a literal l in the form of $u.A > x_l$ or $u.A \geq x_l$ (resp. $u.A < x_l$ or $u.A \leq x_l$), $I'(x_l)$ refines $I(x_l)$ if $I'(x_l) \geq I(x_l)$ (resp. $I'(x_l) \leq I(x_l)$) (“refines a selection condition”); (2) for edge variable x_e , $I'(x_e)$ refines $I(x_e)$ if $I'(x_e) = 1$ and $I(x_e) = 0$ (“adds a query edge”); (3) $I' \succeq_x I$ if $I(x) = '_-$.

We say I' refines I (denoted as $I' \succeq I$) if for every variable x in Q , $I' \succeq_x I'$. Given two instances q' and q induced by I' and I respectively, q' refines q (denoted as $q' \succeq_{\mathcal{I}} q$) if $I' \succeq I$.

We observe the following result.

Lemma 4: *Given a configuration $\mathcal{C} = (G, Q, \mathcal{P}, \epsilon)$, (1) the refinement relation is a preorder; and (2) for any instances q and q' in $\mathcal{I}(Q)$, if $q' \succeq_{\mathcal{I}} q$, then (a) $\delta(q) \geq \delta(q')$; and (b) $f(q) \leq f(q')$ when both q and q' are feasible.* \square

Proof sketch: The above results can be verified by observing that (a) refinement relation is reflexive and transitive, and (b) $q'(G) \subseteq q(G)$ if $q' \succeq_{\mathcal{I}} q$, i.e., any match of u_o in q' remains to be a match of u_o in q if q' refines q at some variables. \square

These results justify the following. Lemma 4 (1) provides a convenient lattice encoding of the search space induced by the refinement preorder. Lemma 4 (2) verifies useful monotonicity properties on diversity and coverage measures that shall be exploited for effective pruning.

Instance Lattice. Following Lemma 4, the algorithm QGen maintains a lattice encoding of the instance space $\mathbb{L} = (\mathcal{I}(Q), \prec_{\mathcal{I}})$ induced by refinement. (1) It initializes $(\mathcal{I}(Q), \prec_{\mathcal{I}})$ with a single root q_r (upper bound) induced by the “most relaxed” instantiation, and a single lower bound q_b induced by the “most refined” instantiation. (2) Each node q in $(\mathcal{I}(Q), \prec_{\mathcal{I}})$ is an instance. For each node q , QGen maintains

- o (a) two Boolean flags: ‘verified’ to record if q is verified, and ‘feasible’ to indicate if q is a feasible instance;
- o (b) $q.(G)$, the (estimated) query answer; and
- o (c) a coordinate $(\delta_\epsilon(q), f_\epsilon(q))$, and a “boxing” coordinate $Box(q) = (\delta_\epsilon(q), f_\epsilon(q))$. The coordinate value $\delta_\epsilon(q)$ (resp. $f_\epsilon(q)$) is defined as $\frac{\log(1+\delta(q))}{\log(1+\epsilon)}$ (resp. $\frac{\log(1+f(q))}{\log(1+\epsilon)}$). $Box(q)$ specifies a box region in the bi-objective (2-dimensional) space of instance q to decide the ϵ -dominance relation (see “Updater” below).

(3) There is an edge (q, q') with a label x if (a) $q' \succeq q$, and (b) q' differs from q in the value of only one variable x , and q' refines q by modifying the value of x to its closest counterpart in the corresponding active domain (e.g., changing 0 to 1 if x is an edge variable). Intuitively, an edge indicates a stepwise refinement action of q by adjusting the value of x only.

Example 6: A fraction of the lattice structure \mathbb{L} that contains $\{q_r, q_1, \dots, q_4, q_5\}$ is shown on the left-hand side of Fig. 4. Algorithm QGen maintains the auxiliary information of e.g., q_3 once it is verified, including the coordinates $\delta(q_3), f(q_3)$, and the boxing coordinates $Box(q_3)$. As q_4 refines q_3 at variable x_1 at the node u_1 , (q_3, q_4) is a direct edge in \mathbb{L} . \square

Generic Algorithm. Given a configuration $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$, algorithm QGen maintains an instance set \mathcal{Q} and iteratively refines \mathcal{Q} towards an ϵ -Pareto instance set of $\mathcal{I}(Q)$. At each iteration i , it refines the solution \mathcal{Q}_i from the last iteration by interacting two procedures.

(1) Spawner. A spawner (Spawn) constructs new instances to be verified that may contribute to the current instance set \mathcal{Q}_i with new ϵ -dominance relation in diversity and coverage. In each iteration, Spawn (a) refines the current configuration given the quality of \mathcal{Q}_i to reduce unnecessary generation, (b) constructs a *front set* of instances \mathcal{Q}_F (thus a fraction of lattice $(\mathcal{I}(Q), \prec_{\mathcal{I}})$) on-the-fly, and (c) prunes unpromising instances that are already ϵ -dominated by \mathcal{Q}_i whenever possible. The spawner performs no actual query processing and verification.

(2) Updater. An updater (Update) refines \mathcal{Q}_i with the front set \mathcal{Q}_F from Spawn towards a better solution \mathcal{Q}_{i+1} . Our idea extends [22] to maintain “boxes” of instances that discretize the bi-objective (2-dimensional) space of answer diversity and coverage of groups. Each box is represented by a single instance q and specified by its boxing coordinates $(\delta_\epsilon(q), f_\epsilon(q))$. To verify ϵ -dominance, it then suffices to verify the dominance of boxing coordinates at both box level and instance level.

We present our main result below.

Theorem 5: Given a configuration $(G, Q(u_o), \mathcal{P}, \epsilon)$, there

Algorithm RfQGen

Input: configuration $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$;
Output: an ϵ -Pareto instance set \mathcal{Q}_ϵ^* .

1. set $\mathcal{Q}_\epsilon^* := \emptyset$; initializes $\mathbb{L} := \{q_r\}$;
2. BFExplore $(\mathcal{C}, q_r, \mathbb{L}, \mathcal{Q}_\epsilon^*)$;
3. **return** \mathcal{Q}_ϵ^* ;

Procedure BFExplore($\mathcal{C}, q_r, \mathbb{L}, \mathcal{Q}_\epsilon^*$)

1. **if** q_r .verified **then return** ;
2. incVerify (q_r, \mathbb{L}, G) ; q_r .verified:= true;
3. **if** $!q_r$.feasible **then return** ;
4. Update $(q_r, \mathcal{Q}_\epsilon^*)$;
5. set $\mathcal{Q}_F :=$ Spawn (q_r, \mathcal{C}) ;
6. **for each** $q' \in \mathcal{Q}_F$ **do**
7. BFExplore $(\mathcal{C}, q', \mathbb{L}, \mathcal{Q}_\epsilon^*)$;

Fig. 3. Algorithm RfQGen

exists an algorithm that (1) correctly maintains an ϵ -Pareto instance set \mathcal{Q}_ϵ^i over all generated instances upon any time i the updater is invoked; (2) ensures a size-bounded \mathcal{Q}_i , where $|\mathcal{Q}_\epsilon^i| \leq \frac{\log |V|}{\log(1+\epsilon)}$, and (3) take $O(|\text{adom}_m|^{|\hat{X}|}(\frac{\log |V|}{\log(1+\epsilon)} + T_q))$ time to compute \mathcal{Q}_ϵ^* , where adom_m refers to the largest active domain, and T_q is the cost of verifying a single instance. \square

We next present two efficient algorithms as constructive proof of Theorem 5. Each implements QGen with different exploration strategies of the instance lattice \mathbb{L} , and convergences to instances with high diversity, or a more balanced distribution of diversity and coverage, for different user preferences. Both have provable guarantees in Theorem 5.

A. Query Generation by Refinement

Our first algorithm, denoted as RfQGen, uses a “refine as always” strategy to compute \mathcal{Q}_ϵ^* . Given a configuration $(G, Q(u_o), \mathcal{P}, \epsilon)$, it starts from the root q_r of the instance lattice \mathbb{L} (which carries search predicates with most “relaxed” conditions), and performs a depth-first exploration of \mathbb{L} . The algorithm uses Lemma 4 (2) to achieve early pruning of infeasible instances, and reduce unnecessary updates.

Algorithm. Algorithm RfQGen is shown in Fig. 3. It initializes set \mathcal{Q}_ϵ^* , and the lattice \mathbb{L} with a single root q_r . It then invokes a recursive procedure RfExplore to perform depth-first exploration, which interacts spawn and update process and generates a front set \mathcal{Q}_F to be explored at each level of \mathbb{L} . RfQGen early terminates if no new instance can be spawned (as RfExplore backtracks), and returns set \mathcal{Q}_ϵ^* .

Procedure BFExplore. The recursive procedure RfExplore starts by verifying an unvisited instance q from the current front set \mathcal{Q}_F . (1) It invokes a procedure incVerify (line 2; not shown) to incrementally update the match set $q.(G)$ [16], along with the coordinates $(\delta(q), f(q))$ and boxing coordinates $(\delta_\epsilon(q), f_\epsilon(q))$. Following Lemma 4, incVerify only determines which matches should be excluded from the counterparts of the verified “parent” of q in \mathbb{L} . (2) It then invokes a procedure Update (line 4) to maintain \mathcal{Q}_F^* given a feasible instance q . (3) For a feasible instance q , it invokes a procedure Spawn

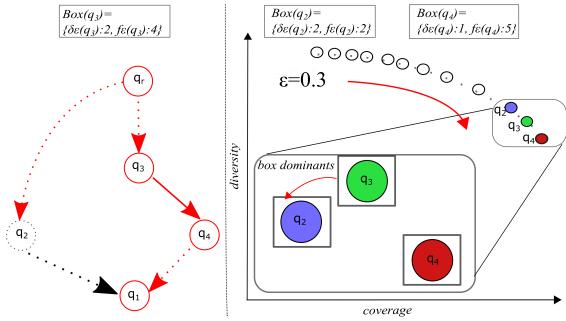


Fig. 4. Instance Lattice and “Refinement as always” exploration

(line 5) to generate the frontier set \mathcal{Q}_F of refined instances (thus spawns a set of children of q in \mathbb{L}), by modifying one variable at a time using the next closest active domain value. It then starts a next-level exploration for each instance in \mathcal{Q}_F . BFExplore backtracks whenever q is not feasible (line 3), as no refined counterparts are feasible (Lemma 4).

Procedure Update. Given a feasible instance q and current ϵ -Pareto instance set \mathcal{Q}_ϵ^* , procedure Update maintains ϵ -dominance by verifying the dominance of Boxing coordinates $Box(q)$, with a case analysis below.

(Case 1) *Replacing boxes* (lines 1-5). This case verifies a box-level dominance relation. Using the boxing coordinates $Box(q) = (\delta_\epsilon(q), f_\epsilon(q))$ of q , it verifies if q introduces a box that also already dominants a set of boxes in the bi-objective space of diversity and coverage. If so, it removes all the representative instances of those boxes from \mathcal{Q}_ϵ^* , and adds q .

(Case 2) *Replacing instances* (lines 6-7). If q falls into a box which is represented by another instance q' , Update simply keeps the one that can dominate the other.

(Case 3) *Adding a non-dominated box* (lines 8-9). If no box can dominate $Box(q)$, Update simply adds q to \mathcal{Q}_ϵ^* (which represents a new box). Here we use $Box(q') \succeq Box(q)$ to denote that $Box(q') \succ Box(q)$ or $Box(q') = Box(q)$.

Example 7: Fig 4 illustrates a case of the running of “update”. Starting from the root of the lattice q_r , RfQGen spawns and verifies instances following the refinement preorder. (1) In the first iteration, Update simply add q_2 to \mathcal{Q}_ϵ^* . (2) Once q_3 is verified, Update removes q_2 under Case (1), as $Box(q_3) \succeq Box(q_2)$ (“Replacing boxes”). (3) In the next iteration, Update keeps q_4 in \mathcal{Q}_ϵ^* , since q_4 and q_3 cannot dominant each other at the box level. (4) Update finally rejects q_1 , since q_3 and q_4 both dominant q_1 . RfQGen then returns \mathcal{Q}_ϵ^* as $\{q_3, q_4\}$. \square

Procedure Spawn. To further reduce generation and verification cost, procedure Spawn uses the following strategy to actively refine the values. Each variable can take, and “simplifies” template $Q(u_o)$ when possible. The refined templates are restored when BFExplore backtracks to ensure the correctness.

Template refinement. Given a verified instance q , Spawn dynamically tracks the subgraph induced by d -hop neighbors of $q(G)$ (d is the diameter of $Q(u_o)$), denoted as G_q^d .

(1) For each literal $u.A$ op x of $Q(u_o)$, it refines the values

Procedure Update ($q, \mathcal{Q}_\epsilon^*$)

1. set $\mathcal{Q}_B := \emptyset$;
/* verify “box-level” dominance */
 2. **for each** $q' \in \mathcal{Q}_\epsilon^*$ **do**
 3. **if** $Box(q') \prec Box(q)$ **then** $\mathcal{Q}_B := \mathcal{Q}_B \cup \{q'\}$;
 4. **if** $\mathcal{Q}_B \neq \emptyset$ **then**
 5. $\mathcal{Q}_\epsilon^* := (\mathcal{Q}_\epsilon^* \setminus \mathcal{Q}_B) \cup \{q\}$;
/* verify “instance-level” dominance */
 6. **else if** there is an instance $q' \in \mathcal{Q}_\epsilon^*$ and $Box(q') = Box(q)$ **then**
 7. **if** $q' \prec q$ **then** $\mathcal{Q}_\epsilon^* := (\mathcal{Q}_\epsilon^* \setminus \{q'\}) \cup \{q\}$;
/* adding a new instance and a non-dominated box */
 8. **else if** there is no instance $q' \in \mathcal{Q}_\epsilon^*$ such that $Box(q') \succeq Box(q)$ **then**
 9. $\mathcal{Q}_\epsilon^* := \mathcal{Q}_\epsilon^* \cup \{q\}$;
 10. **return** \mathcal{Q}_ϵ^* ;
-

Fig. 5. Algorithm Update

x can take to $\{T(v.A)\} \subseteq \text{adom}(A)$, where v ranges over the nodes in G_q^d and $L(v) = L(u)$.

(2) For each edge variable x_e on edge $e = (u, u')$ in $Q(u_o)$, it “fixes” x_e to be 0 if there is no path from any match of u_o in G_q^d with an edge (v, v') such that $L_Q(e) = L((v, v'))$. Moreover, if e is a bridge of $Q(u_o)$, i.e., removing e leads to two connected components in $Q(u_o)$, Spawn removes e and the entire connected component that does not contain u_o .

Example 8: Given a configuration $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$ where \mathcal{P} is defined on gender group of users in G , and $\epsilon = 0.3$. Algorithm RfQGen starts with the root q_r in the lattice \mathbb{L} , as illustrated in Fig. 4. (1) Following a depth first strategy, BFExplore reaches q_3 and invokes Spawn to refine q_3 to q_4 . In particular, Spawn selects variable X_{l1} at node u_1 . While the active domain of “yearsOfExp” suggests three values $\{10, 11, 12, 20\}$, Spawn recognizes that it suffices to explore only $\{10, 11, 12\}$ with the next available value, given that no neighbors of the current match has “yearsOfExp” more than 20. It then generates q_4 and adds it to the front set for further exploration. (2) As the exploration reaches q_2 , it finds a ϵ -Pareto set $\{q_3, q_4\}$ and returns the set (See Example 9). \square

Correctness. Algorithm RfQGen correctly maintains an ϵ -Pareto instance set \mathcal{Q}_ϵ^i over the generated instances $\mathcal{I}^i(Q)$ upon Update is invoked at time i . To see this, assume \mathcal{Q}_ϵ^i is not an ϵ -Pareto instance set. Then either (a) there exists an instance $q \in \mathcal{I}^i(Q) \setminus \mathcal{Q}_\epsilon^i$ that is not ϵ -dominated by any instance in \mathcal{Q}_ϵ^i , or (b) $q \in \mathcal{Q}_\epsilon^i$ but not in the Pareto set of $\mathcal{I}^i(Q)$. For case (a), Update only removes q if there is another verified instance $q' \in \mathcal{I}^i(Q)$ that either dominants q (line 7), or ϵ -dominants q (line 3). In either case, it contradicts the assumption. Similarly, case (b) indicates that there exists at least an instance $q' \in \mathcal{I}^i(Q)$ that $q' \succeq q$. Thus q' is verified at some time and either remains in \mathcal{Q}_ϵ^i or leaves a box $Box(q'')$, where $q'' \succeq q'$. In either case, q should be excluded by Update from \mathcal{Q}_ϵ^i (at line 7 or line 3), which contradicts that $q \in \mathcal{Q}_\epsilon^i$.

Size bound. We next show that at any time i , $|\mathcal{Q}_\epsilon^i| \leq \frac{\log |V|}{\log(1+\epsilon)}$. To see this, observe that (a) Update ensures that each box

Algorithm BiQGen

Input: configuration $\mathcal{C} = (G, Q(u_o), \mathcal{P}, \epsilon)$;
Output: an ϵ -Pareto instance set \mathcal{Q}_ϵ^* .

1. queue $S_f := \{q_r\}$; queue $S_b := \{q_b\}$; set $\mathcal{Q}_\epsilon^* := \emptyset$;
2. set $\mathcal{Q}_f := \emptyset$; set $\mathcal{Q}_b := \emptyset$; set SBounds:= \emptyset ;
3. **while** $S_f \neq \emptyset$ or $S_b \neq \emptyset$ **do**
- /* forward exploration */
4. **if** $S_f \neq \emptyset$ **then** instance $q := \mathcal{Q}_f.\text{dequeue}()$;
5. **if** $q.\text{verified}$ or $!q.\text{feasible}$ **then** continue;
6. **if** SPrune(q , SBounds) **then** continue;
7. incVerify (q); $q.\text{verified} := \text{true}$;
8. **if** $q.\text{feasible}$ **then** Update ($q, \mathcal{Q}_\epsilon^*$);
9. $\mathcal{Q}_F^f := \text{SpawnF } (q, \mathcal{C})$; $S_f.\text{enqueue}(\mathcal{Q}_F^f)$;
- /* backward exploration */
10. **if** $S_b \neq \emptyset$ **then** instance $q' := \mathcal{Q}_b.\text{dequeue}()$;
11. **if** $q'.\text{verified}$ or $!q'.\text{feasible}$ **then** continue;
12. **if** SPrune(q' , SBounds) **then** continue;
13. incVerify (q'); $q'.\text{verified} := \text{true}$;
14. **if** $q'.\text{feasible}$ **then** Update ($q', \mathcal{Q}_\epsilon^*$);
15. $\mathcal{Q}_F^b := \text{SpawnB } (q', \mathcal{C})$; $S_b.\text{enqueue}(\mathcal{Q}_F^b)$;
- /* update “Sandwich” bounds with feasible pair (q, q') */
16. **if** $q' \succ_{\mathcal{I}} q$ and ($\text{Box}(q').\delta = \text{Box}(q).\delta$ or $\text{Box}(q').f = \text{Box}(q).\delta$)
17. update SBounds with (q, q') ;
18. **return** \mathcal{Q}_ϵ^* ;

Fig. 6. Algorithm BiQGen

is represented by a single instance; (b) there are in total $\frac{\log(|V|)}{\log(1+\epsilon)} \frac{C}{\log(1+\epsilon)}$ boxes in the bi-objective 2D space, thus at most $\frac{\log(|V|)}{\log(1+\epsilon)}$ instances having same coverage that ϵ -dominant the rest (as $\delta(q) \in [0, |V|]$), or $\frac{\log C}{\log(1+\epsilon)}$ instances that ϵ -dominate the rest having same diversity (as $f(q) \in [0, C]$). As $C \leq |V|$ for disjoint groups, the size bound follows.

Time Cost. Following Theorem 3, Spawn generates up to $2^{|X_e|} |\text{adom}_m|^{|X_L|}$ instances. Thus it takes in $O(|\text{adom}_m|^{\hat{X}|})$ runs of BFExplore. For each instance q , it takes Update $O(\frac{\log(|V|)}{\log(1+\epsilon)} + T_q)$ time to verify q (in time T_q) and update \mathcal{Q}_ϵ^q . Thus the total time cost is in $O(|\text{adom}_m|^{\hat{X}|} (\frac{\log(|V|)}{\log(1+\epsilon)} + T_q))$ time. We found that adom_m and $|\hat{X}|$ are usually small in real-world graphs. Moreover, the early pruning of infeasible instances significantly reduce on average 40% of generated ones, compared with the naive algorithm EnumQGen (Section VI).

The above analysis completes the proof of Theorem 5.

B. Bi-directional Query Generation

Algorithm RfQGen achieves early convergences to an ϵ -Pareto instance set \mathcal{Q}_ϵ^* , where a majority of instances may have high answer diversity (Lemma 2 (2a)).

We next present a second algorithm, denoted as BiQGen. It adopts a bi-directional strategy that explores \mathbb{L} from both ends. The forward exploration inspects instances with non-increasing diversity, and the backward exploration keeps “relaxing” instances towards early convergence to instances with high coverage. Following Lemma 2 (2), the computation has more chance to generate \mathcal{Q}_ϵ^* with a more “balanced” distribution on instances with high diversity and those with desired coverage, as also verified by our experiments (Section VI).

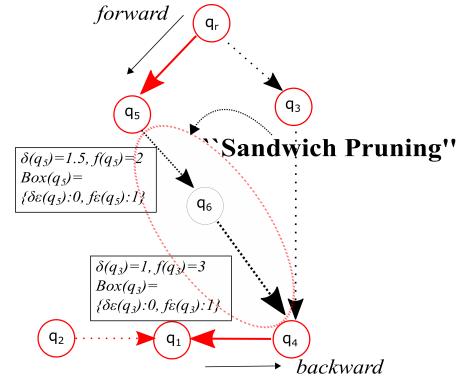


Fig. 7. Bi-directional Query Generation

Algorithm. The algorithm BiQGen (shown in Fig. 6) uses the same procedure Update as in RfQGen to maintain \mathcal{Q}_ϵ^* at any time, but specifies two separate spawners: SpawnF, same as Spawn in RfQGen, and SpawnB, a reversed “relaxation” counterpart that yields new instances by relaxing the search predicates. The procedure SpawnF and SpawnB are invoked to generate a front set \mathcal{Q}_F^f (line 9) and \mathcal{Q}_F^b (line 15) in a “forward” refinement-based exploration from q_r , and a “backward” relaxation-based exploration from q_b , respectively. It uses two queues S_f and S_b to control the iterative forward and backward exploration (line 2). It iteratively performs forward (lines 3-9) and backward exploration (lines 10-15), and terminates if no new instances can be generated (line 3).

“Sandwich” pruning. The bidirectional strategy enables an effective pruning strategy that exploits the monotonicity properties of diversity and coverage in Lemma 4(2).

Lemma 6: For any feasible instances $q \in \mathcal{Q}_F^f$ and $q' \in \mathcal{Q}_F^b$, if $q' \succeq_{\mathcal{I}} q$, and (a) $\text{Box}(q).\delta = \text{Box}(q').\delta$ or (b) $\text{Box}(q).f = \text{Box}(q').f$, then for any instance $q'' \in \mathcal{I}(Q)$ where $q \prec_{\mathcal{I}} q'' \prec_{\mathcal{I}} q'$, $q'' \notin \mathcal{Q}_\epsilon^*$. \square

Proof sketch: Consider an instance $q'' \in \mathcal{I}(Q)$ where $q \prec_{\mathcal{I}} q'' \prec_{\mathcal{I}} q'$. Following Lemma 4, $\delta(q') \leq \delta(q'') \leq \delta(q)$, and $f(q') \geq f(q'') \geq f(q)$. For Case (a), if $\text{Box}(q).\delta = \text{Box}(q').\delta$, then $\text{Box}(q'').\delta = \text{Box}(q').\delta$ and $f(q') \geq f(q'')$. Thus $q' \succ q''$ or $q' \succ_\epsilon q''$. Similarly, for Case (b), if $\text{Box}(q).f = \text{Box}(q').f$, then $\text{Box}(q'').f = \text{Box}(q').f$. Thus $q \succ q''$ or $q \succ_\epsilon q''$. In either case, Update rejects q'' (line 5 or line 7). \square

During the bi-directional exploration, BiQGen keep tracks of the occurrences of “sandwich” pairs (q, q') that satisfy the condition in Lemma 6 in a set SBounds. Upon a new pair (q, q') is identified (line 16), it updates SBounds by (a) replacing any pair (q_1, q_2) with (q, q_2) (resp. (q_1, q')) if $q \prec_{\mathcal{I}} q_1$ (resp. $q_2 \prec_{\mathcal{I}} q'$), or (q, q') if $q \prec_{\mathcal{I}} q_1$ and $q_2 \prec_{\mathcal{I}} q'$; or (b) adding (q, q') (line 17). This in turn allows more instances to be pruned (by a procedure SPrune; lines 6 and 12), for both forward and backward exploration.

These pruning strategies are fast and effective. Checking the refinement preorder $\prec_{\mathcal{I}}$ takes $O(|\hat{X}|)$ time per instance. We found on average 60% of the generated instances from EnumQGen are pruned by BiQGen (see Section VI).

Example 9: Given $\mathcal{I}(Q)$ that contains $\{q_r, q_1, q_2, q_3, q_4\}$, BiQGen starts a forward search from q_r as in RfQGen, and a backward search from q_1 . (1) In the first round of bidirectional search, SpawnF refines q_r to q_5 , and SpawnB relaxes q_1 to q_4 . Upon the verification of q_4 and q_5 , BiQGen finds that $Box(q_5).f = Box(q_4).f = 4$. It then creates a pair (q_5, q_4) and adds it to SBounds. (2) In the next round, the backward search reaches q_2 . Meanwhile, as $q_5 \prec_{\mathcal{I}} q_6 \prec_{\mathcal{I}} q_4$, q_6 is skipped in the forward search without further exploration. \square

Analysis. The correctness of BiQGen follows from the correctness analysis of Update, SpawnF and SpawnB (following the analysis of Spawn) and the invariant that forward and backward exploration verifies and safely prunes $\mathcal{I}(Q(u_o))$ (Lemma 4 and Lemma 6). The size bound and time cost follows from similar analysis as in RfQGen.

V. ONLINE QUERY GENERATION

Another need of query generation is to produce workloads with arbitrary size k with diversity and coverage requirements over interested groups for query benchmarking [32], [4], [5], [6]. We consider the following problem. Given a configuration $\mathcal{C}=(G, Q(u_o), \mathcal{P}, k)$, maintain an ϵ -Pareto instance set $\mathcal{Q}_{(\epsilon,k)}^t$ over a large set of instances (due to e.g., active domains and G), such that at any time t , (a) $|\mathcal{Q}_{(\epsilon,k)}^t| = k$, and (b) ϵ is as small as possible. This is nontrivial: as more instances arrives, one needs to compromise with a larger ϵ (thus worse approximation) for smaller k (Theorem 5).

We extend QGen to an online algorithm, denoted as OnlineQGen, to maintains an ϵ -Pareto instance set with a fixed size k and a small ϵ at any time. To cope with large instance space, it treats $\mathcal{I}(Q)$ as a stream of instances from a query generator. Unlike RfQGen and BiQGen, it does not assume an ordered processing (e.g., “refinement”). Instead, (1) it uses a sliding window W_Q with a bounded size w to cache a certain number of instances that can help reduce ϵ , while keeping k fixed; and (2) it *incrementalizes* the maintenance of $\mathcal{Q}_{(\epsilon,k)}^t$ upon the arrival of a new instance, and only perform necessary maintenance when Update causes size growth (in particular, Case (3) in Update).

Online Algorithm. The algorithm, as shown in Fig. 8, takes as input a stream of instances from an arbitrary generator that instantiates $Q(u_o)$, and a small initial constant $\epsilon_m > 0$. It starts by populating $\mathcal{Q}_{(\epsilon,k)}^t$ with Update upon newly arrived instances (in arbitrary order), until $|\mathcal{Q}_{(\epsilon,k)}^t| = k$ (lines 7-10). If an instance q is rejected by Update, OnlineQGen includes it to W_Q (line 9) for future consideration (up to at most w timestamps before it “expires”; lines 5-6). This is to “temporally” keep the instances that may be accepted by Update again, thus reduce $|\mathcal{Q}_{(\epsilon,k)}^t|$ (hence ϵ remains unchanged).

When $|\mathcal{Q}_{(\epsilon,k)}^t| = k$ and a new instance q can be added, OnlineQGen incrementalizes Update by individually checking (a) if adding q increases $|\mathcal{Q}_{(\epsilon,k)}^t|$ (Case (3)) or not (Case (1) and (2); see Update in Section IV). if the latter, it simply adds q (lines 12-13); otherwise, it first finds the nearest neighbor of

Algorithm OnlineQGen

```

Input: a configuration  $(G, Q(u_o), \mathcal{P}, k)$ ,  $\epsilon_m$ ;
       a stream of instances  $\mathcal{I}(Q)$ , a cache size  $w$ ;
Output: a size- $k$   $\epsilon$ -Pareto instance set  $\mathcal{Q}_{(\epsilon,k)}^t$  at any time  $t$ .
1.   set  $\mathcal{Q}_{(\epsilon,k)} := \emptyset$ ; set  $W_Q := \emptyset$ ; integer  $t := 0$ ;  $\epsilon := \epsilon_m$ ;
2.   while  $\mathcal{I}(Q)$ .hasNext() do
3.     instance  $q := \mathcal{I}(Q)$ .getNext();
4.     verify  $q$ ;  $q.ts := i$ ;  $i := i + 1$ ;
/* remove “expired” query instances */
5.     for each  $q' \in W_Q$  do
6.       if  $q'.ts < i - w + 1$  then  $W_Q := W_Q \setminus \{q'\}$ ;
7.       if  $|\mathcal{Q}_{(\epsilon,k)}| < k$  then
8.         Update ( $q, \mathcal{Q}_{(\epsilon,k)}$ );
/* cache an  $\epsilon$ -dominated instance for future update */
9.       if  $q \notin \mathcal{Q}_{(\epsilon,k)}$  then  $W_Q := W_Q \cup \{q\}$ ;
10.      else continue ;
11.      if  $|\mathcal{Q}_{(\epsilon,k)}| = k$  then
12.        if Update accepts  $q$  with Case (1) or (2) then
13.           $\mathcal{Q}_{(\epsilon,k)} := \mathcal{Q}_{(\epsilon,k)} \cup \{q\}$ ; Continue;
14.        if Update accepts  $q$  with Case (3) then
/* replace an instance with  $q$  */
15.           $q' := \text{NearestNeighbor}(q, \mathcal{Q}_{(\epsilon,k)})$ ;
16.           $\epsilon := \text{dist}((q.\delta, q.f), (q'.\delta, q'.f))$ ;
17.           $\mathcal{Q}_{(\epsilon,k)} := \mathcal{Q}_{(\epsilon,k)} \setminus \{q'\}$ ;
/* check if a cached instance can be added without impact */
18.          if there is a  $q_b \in W_Q$  such that Update accepts  $q_b$ 
           in Case (1) or Case (2) then
19.             $\mathcal{Q}_{(\epsilon,k)} := \mathcal{Q}_{(\epsilon,k)} \cup \{q_b\}$ ;
20.             $\mathcal{Q}_{(\epsilon,k)} := \mathcal{Q}_{(\epsilon,k)} \cup \{q\}$ ;
21.          return  $\mathcal{Q}_{(\epsilon,k)}$  upon request;
22.          return  $\mathcal{Q}_{(\epsilon,k)}$ ;
```

Fig. 8. Algorithm OnlineQGen

q in $\mathcal{Q}_{(\epsilon,k)}^t$ to be replaced by q . To this end, it enlarges ϵ as the Euclidean distance of the coordinates of q and q' , in order to include q and q' in a larger box (line 16). It also verifies if a cached instance can be added without increasing $|\mathcal{Q}_{(\epsilon,k)}^t|$ (lines 18-20). It returns a size- k ϵ -Pareto set $\mathcal{Q}_{(\epsilon,k)}^t$ upon request (line 21) or no new instance is generated (line 22).

Analysis. OnlineQGen correctly maintains an ϵ -Pareto instance set with a fixed size k for the “seen” fraction of $\mathcal{I}(Q)$ at time t . We first observe the following property.

Lemma 7: *If $q \prec_{\epsilon} q'$, then $q \prec_{\epsilon'} q'$ for any $\epsilon' > \epsilon$.* \square

The correctness follows from the following invariant: at any time t , (1) either Update correctly rejects an instance that is already ϵ -dominated by an instance in $|\mathcal{Q}_{(\epsilon,k)}^t|$, including those “expired” in W_Q ; or (2) ϵ is adjusted to a larger counterpart to reduce the size of $|\mathcal{Q}_{(\epsilon,k)}^t|$ to k (Theorem 5), and preserves any previous ϵ -dominance relation (Lemma 7).

Delay time. OnlineQGen efficiently maintains ϵ -Pareto instance set with a delay time in $O(T_q + w + k)$ time, where T_q is the cost of verifying a single instance. This verifies practical application of OnlineQGen in query generation and selection with desired diversity and coverage for large workload.

VI. EXPERIMENTS

Based on real-world graphs, we experimentally verify the effectiveness and efficiency of our algorithms for FairSQG.

Experiment Setting. We used the following setting.

Datasets and Groups. We use three real-life data graphs.

- (1) DBP is a movie knowledge graph induced from DBpedia [25] with $1M$ entities (e.g., movies, directors, actors) and $3.18M$ relations (e.g., directed, collaboration). Each node has attributes such as title, genre, and years, and on average 10 attributes. We induce movie groups based on their genres or countries, and create up to 4 groups.
- (2) LKI [37] contains $3M$ nodes (e.g., users, organizations) with labels (professions, skills) and $26M$ edges (e.g., co-review). Each node has attributes such as “Major” with on average 7 attributes. We induce gender groups \mathcal{P} with synthetic genders generated by gender inference tools [13].
- (3) Cite [33] contains $4.9M$ nodes (e.g., papers, authors) with labels (e.g., topics) and $46M$ edges (e.g., citations, authorship). Each node has attributes such as “numberOfCitations”. The nodes have on average 6 attributes. We induce groups \mathcal{P} of papers having different topics.

Queries and Templates. We developed a generator to produce query templates with practical search conditions, controlled by number of variables $|\hat{X}|$ (specifically, the number of edge variables and range variables), query size $|Q(u_o)|$ (in terms of the number of edges) and topologies.

For each dataset, we generated a set of $Q(u_o)$ and \mathcal{P} and ensure the existence of feasible query instances. The largest set of instances $\mathcal{I}(Q)$ for DBP, LKI, and Cite are 1000, 1400 and 800, respectively. We quantify the diversity of two nodes with the normalized edit distances of their matching attributes.

Algorithms. We implemented the following algorithms in Java: (1) EnumQGen, which enumerates and verifies the instances in $\mathcal{I}(Q)$, and performs a simple nested loop to compute the ϵ -Pareto optimal instance set. (2) RfQGen, with “refine as always” strategy; (3) BiQGen, the algorithm that adopts bi-directional search with “Sandwich” pruning; and (4) the online query generation algorithm OnlineQGen, which maintains an ϵ -Pareto instance set with a fixed size k and small ϵ . (5) To verify the quality of query generation, we also implemented Kungs, an algorithm that enumerates and verifies the instances in $\mathcal{I}(Q)$, and invokes Kung’s algorithm [12] to compute the Pareto optimal non-dominated set. All our source codes, datasets, and test cases are available online¹.

Experimental results. We next present our findings.

Exp-1: Effectiveness. We first evaluated the effectiveness of our algorithms. As the diversity and coverage of queries vary over different graphs, we quantify the effectiveness with two established relative indicators for Pareto set approximations: R -indicator, and ϵ -indicator[38], for a fair comparison.

ϵ -Indicator (I_ϵ). Given a set of tuples \mathcal{Q} , the ϵ -indicator [38] finds the minimum ϵ , denoted by ϵ_m , for which \mathcal{Q} is an ϵ_m -Pareto set. Given ϵ -Pareto instance set \mathcal{Q}_ϵ^* that conform to a given constant ϵ , we define a *normalized ϵ -Indicator* (denoted

as I_ϵ), which is computed as $I_\epsilon(\mathcal{Q}_\epsilon^*) = 1 - \frac{\epsilon_m}{\epsilon}$, where ϵ_m refers to the minimum constant such that for any instance $q \in \mathcal{I}(Q)$, there still exist an instance q' and $q' \succeq_{\epsilon_m} q'$, i.e., \mathcal{Q}_ϵ^* remains to be an ϵ_m -Pareto instance set. The larger $I_\epsilon(\mathcal{Q}_\epsilon^*)$ is, the better. For the complete Pareto optimal set \mathcal{Q}^* , $I(\mathcal{Q}^*) = 1$.

R -indicator (I_R). For a set of tuples \mathcal{Q} , an R -indicator takes into consideration users’ preference, and maps \mathcal{Q} to a score by aggregating the weighted attribute values [38]. We define a simple R -indicator with a preference factor $\lambda_R \in (0, 1)$, denoted as I_R , which is defined as $I_R(\mathcal{Q}_\epsilon^*) = \frac{(1-\lambda_R)\delta^* + \lambda_R f^*}{2}$, where δ^* (resp. f^*) refers to the maximum diversity (resp. coverage) of an instance in \mathcal{Q}_ϵ^* (normalized to be in $[0, 1]$). Here we use λ_R to “reward” the quality of \mathcal{Q}_ϵ^* in terms of coverage: a higher λ_R indicates the user’s preference that favors queries with better coverage property; accordingly, a higher I_R under fixed λ_R suggests a query set \mathcal{Q}_ϵ^* that contain queries with more desired group coverage.

Overall Effectiveness (ϵ -indicator). We compare the effectiveness of Kungs, EnumQGen, RfQGen and BiQGen over the three real life datasets (Fig. 9(a)). We set $|Q| = 3$ with 3 variables (1 edge variables, and 2 range variables), $|\mathcal{P}| = 2$, $\epsilon = 0.01$, and $C=200$. We use an “Equal opportunity” scenario and set $c = 100$ for both groups. (1) Kungs always can achieve scores as 1 over all the graphs as it computes the exact Pareto-optimal sets. (2) Over all the datasets, EnumGen, RfQGen and BiQGen achieve I_ϵ at least 0.6, which indicates that they constantly achieve an “actual” approximation of Pareto optimal set with an ϵ_m constantly smaller than $0.4 \cdot \epsilon$ for a predefined ϵ . (3) RfQGen and BiQGen approximate the complete Pareto set almost equally good as EnumGen, which enumerates all query instances. On the other hand, RfQGen and BiQGen on average inspects 40% and 60% less instances compared with EnumGen and Kungs.

On average, RfQGen and BiQGen effectively approximate Pareto optimal sets with a representative subset of 10% of their sizes. For example, while the complete Pareto set contains 60 instances over LKI, BiQGen computes an ϵ -Pareto instance set with only 10 instances.

Varying ϵ (ϵ -indicator). Fixing $|Q| = 4$, $|\hat{X}| = 3$ (with 1 range variables and 2 edge variables), and $C = 200$, we varied ϵ from 0.1 to 1 and evaluate its impact over LKI. Fig. 9(b) verifies the following. (1) EnumGen, RfQGen and BiQGen approximate Pareto optimal set with larger ϵ_m (all bounded by ϵ). This is due to the trade-off between the enforced tolerance ϵ and the output size. The larger ϵ is, the fewer the boxes and representative ϵ -dominance instances are verified by RfQGen and BiQGen. Thus it is more difficult to use less amount of representative instances to approximate the Pareto set. (2) In all cases, RfQGen and BiQGen are able to approximate Pareto optimal set with a small ϵ_m up to 0.4 of predefined ϵ in all cases ($I_R \leq 0.6$), and achieve same performance as EnumGen. These suggest our methods can generate good approximation of the Pareto set over various settings of ϵ .

Varying $|X_L|$ (ϵ -indicator). We use DBP to evaluate the im-

¹<https://github.com/PanCakeMan/QueryGen>

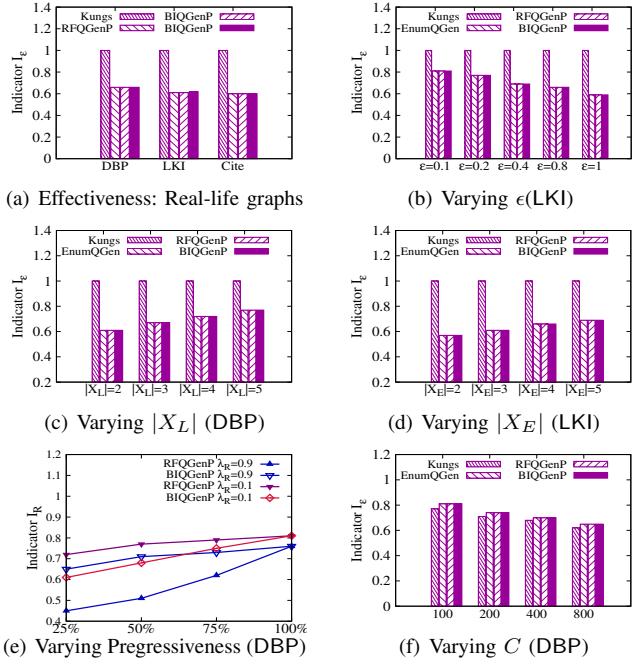


Fig. 9. Effectiveness of Subgraph Query Generation

pact of range variables given that the nodes have more attributes on average. Fixing $|Q| = 4$, $|\mathcal{P}| = 200$, $\epsilon = 0.01$, we varied the number of range variables from 2 to 5, and evaluated the impact to the effectiveness of Kungs, EnumQGen, RfQGen and BiQGen. As shown in Fig. 9(c), EnumQGen, RfQGen and BiQGen approximate the Pareto set better for larger $|X_L|$. Interestingly, on one hand, the larger $|X_L|$ is, the more ϵ -dominating query instances are verified to approach Pareto optimal set; on the other hand, the increased query complexity lead to less number of matches, which in turn reduces feasible instances and the sizes of Pareto instance sets, thus make it “easier” to approximate Pareto set with fewer instances.

Varying $|X_e|$ (ϵ -indicator). We use LKI to evaluate the impact of edge variables, given its dense social structures. Fixing $|Q| = 5$, $|\mathcal{P}| = 200$, $\epsilon = 0.01$, we varied the number of range variables from 2 to 5, and evaluated its impact to the effectiveness of Kungs, EnumQGen, RfQGen and BiQGen. We observe a consistent trend (Fig. 9(d)) for the algorithms over larger $|X_e|$ as for their counterparts in Fig. 9(c). Similarly, more edge variables allow the spawning of more query instances and help us identify more dominating instances, approximating Pareto set with fewer ones.

The above results verify that our methods suggest better approximation for higher template complexity (in terms of the the number of range and edge variables), due to the reduction of size of feasible instances and Pareto optimal set, and a larger query instance space that can be efficiently explored by RfQGen and BiQGen.

“Any time” quality with user preference (R-indicator). In this test, we evaluate the convergence property of RfQGen and BiQGen in response to different user preferences (controlled

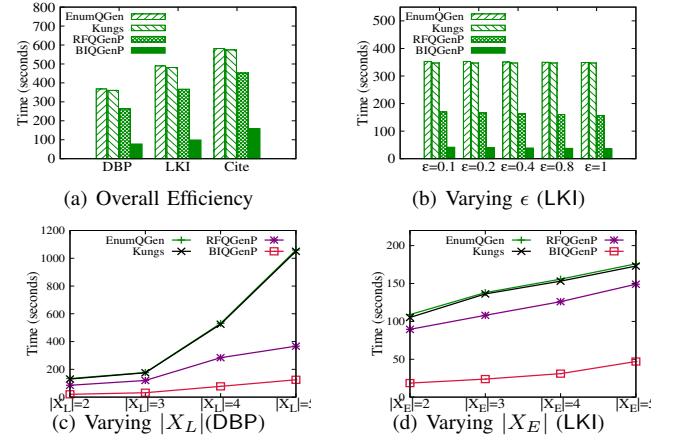


Fig. 10. Efficiency of Subgraph Query Generation

by λ_R). Fixing $|Q| = 4$, $|\mathcal{P}| = 200$, $|\bar{X}| = 3$, and $\epsilon = 0.01$, we report I_R when different fraction of $\mathcal{I}(Q)$ are explored over DBP, with $\lambda_R = 0.9$ (favoring high coverage) and $\lambda_R = 0.1$ (favoring answer diversity). We observe the following. (1) RfQGen converges faster to a set of instances with high answer diversity than BiQGen when $\lambda_R = 0.1$, as the refinement strategy probes feasible instances (2) On the other hand, BiQGen promotes the discovery of instances with desired group coverage when $\lambda_R = 0.9$, due to the bi-directional search brings more feasible queries with higher coverage from the backward exploration. (3) Consistently, BiQGen and RfQGen converges to query sets with higher diversity and coverage, respectively.

Varying C (R-indicator). Fixing $|Q| = 4$, $|\mathcal{P}| = 3$, $|\bar{X}| = 3$ and $\lambda_R = 0.5$ which represents an equal preference over diversity and coverage. We evaluate the impact of coverage requirement. We follow equal opportunity and evenly distribute C to each group, and report I_R over DBP. As more nodes are required to be covered, less instances become feasible. This reduces the chance for EnumGen, RfQGen and BiQGen to identify ϵ -dominating instances that can approximate Pareto set.

These result verify the application of our methods for generating favorable queries for different user preferences.

Exp-2: Efficiency. We next evaluate the efficiency of our query generation algorithms.

Efficiency over real-life graphs. using the same setting as in Fig. 9(a), we report the efficiency of Kungs, EnumQGen, RfQGen and BiQGen, over the real datasets in Fig. 10(a). (1) BiQGen achieves the best performance for all the datasets. On average, it outperforms EnumQGen and RfQGen by 4.4 and 2.5 times, respectively, due to the bi-directional search, and the pruning from both forward and backward exploration. (2) Query generation with diversity and coverage is feasible for large graphs. For example, it takes BiQGen (resp. RfQGen) 78s (resp. 367s) over LKI with 3M nodes and 26M edges.

Varying ϵ . Using the same setting as in Fig. 9(b), we report the efficiency of the algorithms in Fig. 10(b). (1) EnumGen and Kungs are not sensitive due to that their main bottleneck is

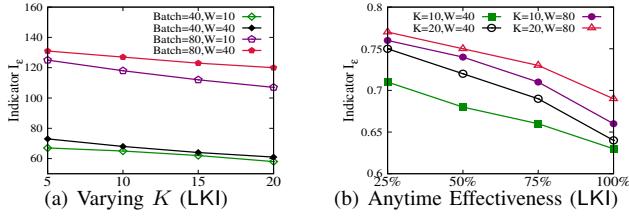


Fig. 11. Performance of OnlineQGen

the enumeration and verification of all instances. (2) BiQGen achieves the best performance among all the algorithms due to effective pruning. BiQGen (resp. RfQGen) outperforms EnumGen by 6 times (resp. 2.2 times) on average. While not very sensitive to ϵ , BiQGen and RfQGen take slightly less time over large ϵ due to that more instances are ϵ -dominated and captured by Update, thus are early pruned.

Varying $|X_L|$ and $|X_e|$. Following the setting as in Fig. 9(c), we report the efficiency of the algorithms in Fig. 10(c). BiQGen achieves the best performance among all the algorithms, and is the least sensitive compared with others. BiQGen (resp. RfQGen) outperforms EnumQGen (resp. QGenEq_n) by 7.5 times (resp. 5.6 times) on average over DBP.

Using the same setting as in Fig. 10(d) over LKI, Fig. 10(d) verifies that BiQGen achieves the best performance among all the algorithms. BiQGen (resp. RfQGen) outperforms EnumQGen by 3 times (resp. 2.1 times) on average due to the pruning over LKI. RfQGen and BiQGen are less sensitive to $|X_e|$. This is because an increase of edge variables (and by enforcing them to ‘1’) significantly reduces feasible instances and are effectively captured by spawn procedure during template refinement (Section IV).

Exp-3: Online Generation. We next evaluate the performance of OnlineQGen over LKI. We simulate instance streams by randomly instantiating fixed query templates.

Delay time: Varying k and Batch sizes. Fig. 11(a) reports the delay time of OnlineQGen to process a batch of instances (with size 40 or 80) from the input stream. We varied k from 5 to 20 and set window size w as 10 and 40, respectively. While OnlineQGen takes around 1 second per instance to maintain $\mathcal{Q}(\epsilon, k)$ of size k , on average it takes 63 seconds for the batch with size 40 and 121 seconds for the batch with size 80. It takes less time for larger k and smaller w . Indeed, the cached instances and incremental update reduces the chance of k to be enlarged; on the other hand, the larger w is, the more unexpired instances in the cache need to be verified.

Anytime Effectiveness (ϵ -indicator). Keeping the setting in Fig. 11(a), we evaluate the anytime effectiveness of OnlineQGen by setting $k = 10, 20$ and $w = 40, 80$, respectively. Fig. 11(b) verifies the following. (1) I_e decreases as the OnlineQGen evaluates more instances from the stream. This is consistent with our observation in Fig. 9(b) even when k is not fixed. Indeed, OnlineQGen compromises ϵ (a case in Fig. 9(b)) in trade for smaller k as more instances arrives from the stream. On the other hand, it retains an $I_R \geq 0.63$

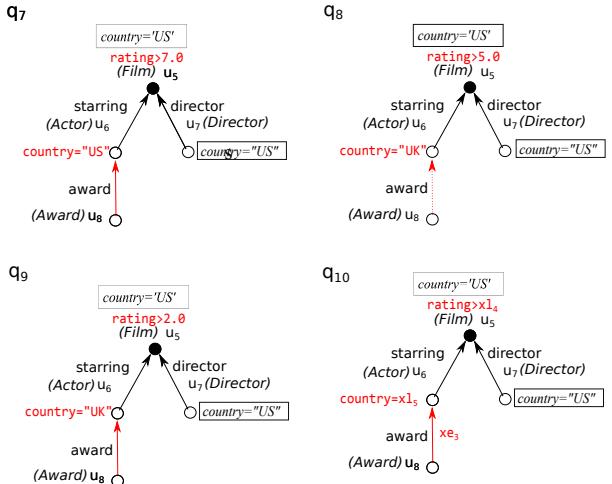


Fig. 12. Case study: Query Generation

at any time. (2) OnlineQGen effectively exploits larger w over larger k to achieve higher I_e . The incremental updating and caching strategy reduces the unnecessary growth of ϵ as well as k , keeping both smaller to maintain high-quality queries.

Exp-4: Case Study. We conducted case analysis to evaluate how our algorithms adapt to users’ preference. The query template (shown as q_{10}) contains parameterized ratings, country and award information, and three query instances are shown in Fig. 12 for movie search (DBP). An initial query (not shown) searches for high rating, award winning US movies ($\text{rating} > 7$) with US actors returns a skewed distribution of 350 romance movies and 120 horror movies. Upon enforcing an equal coverage over movie genres, BiQGen prefers q_7 and q_8 , achieving more desired coverage over movie genres. For example, q_8 refines the results to 112 romance movies and 103 horror movies. On the other hand, RfQGen returns q_7 and q_9 , where q_9 has more skewed results compared with q_7 , but a more diversified set of movies.

VII. CONCLUSIONS

We have introduced and studied the subgraph query generation with both diversity and group fairness constraints. We have introduced a formalization that aims to compute bi-objective Pareto optimal set in terms of answer diversity and coverage of groups of interests. We have provided the upper and lower bound analysis of the problem. We have provided two feasible algorithms that approximate the Pareto-optimal query instance set with ϵ -Pareto instance set, with effective pruning strategies, and an online algorithm that maintains the ϵ -Pareto instance set with a fixed size and high quality (small ϵ). As verified analytically and experimentally, our methods are feasible for large graphs, and can achieve desirable diversity and coverage properties over targeted groups. A future topic is to study parallel query generation over large graphs with diversity and group fairness. Another future topic is to incorporate more fairness measures, such as top-k fairness.

REFERENCES

- [1] Full version. <https://github.com/PanCakeMan/SQGen/blob/main/full.pdf>.
- [2] Z. Abbassi, V. Mirrokni, and M. Thakur. Diversity maximization under matroid constraints. In *KDD*, 2013.
- [3] A. Asudeh and H. Jagadish. Fairly evaluating and scoring items in a data set. *Proceedings of the VLDB Endowment*, 13(12):3445–3448, 2020.
- [4] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. Controlling diversity in benchmarking graph databases. *arXiv preprint arXiv:1511.08386*, 11, 2015.
- [5] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2016.
- [6] A. Bonifati, G. Fletcher, J. Hidders, and A. Iosup. A survey of benchmarks for graph-processing systems. In *Graph Data Management*, pages 163–186. 2018.
- [7] A. Bonifati, I. Holubová, A. Prat-Pérez, and S. Sakr. Graph generators: State of the art and open challenges. *ACM Computing Surveys (CSUR)*, 53(2):1–30, 2020.
- [8] S. Bouhenni, S. Yahiaoui, N. Nouali-Taboudjemat, and H. Kheddouci. A survey on distributed graph pattern matching in massive graphs. *CSUR*, 54(2):1–35, 2021.
- [9] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, 2010.
- [10] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *ACM SIGMOD Record*, 42(3):6–18, 2013.
- [11] P. Ciaccia and D. Martinenghi. Reconciling skyline and ranking queries. *Proceedings of the VLDB Endowment*, 10(11):1454–1465, 2017.
- [12] L. Ding, S. Zeng, and L. Kang. A fast algorithm on finding the non-dominated set in multi-objective optimization. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, 2003.
- [13] Y. Dong, Y. Yang, J. Tang, Y. Yang, and N. V. Chawla. Inferring user demographics and social strategies in mobile social networks. In *KDD*, 2014.
- [14] T. Draws, N. Tintarev, and U. Gadiraju. Assessing viewpoint diversity in search results using ranking fairness metrics. *ACM SIGKDD Explorations Newsletter*, 23(1):50–58, 2021.
- [15] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *VLDB*, 2013.
- [16] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):1–47, 2013.
- [17] Y. Ge, S. Zhao, H. Zhou, C. Pei, F. Sun, W. Ou, and Y. Zhang. Understanding echo chambers in e-commerce recommender systems. In *SIGIR*, pages 2261–2270, 2020.
- [18] S. C. Geyik, S. Ambler, and K. Kenthapadi. Fairness-aware ranking in search & recommendation systems with application to linkedin talent search. In *KDD*, 2019.
- [19] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, 2009.
- [20] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2797–2811, 2015.
- [21] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.
- [22] M. Laumanns, L. Thiele, E. Zitzler, and K. Deb. Archiving with guaranteed convergence and diversity in multi-objective optimization. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 439–447, 2002.
- [23] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *ICDE*, pages 666–677, 2012.
- [24] M. Lissandrini, D. Mottin, T. Palpanas, and Y. Velegrakis. Graph-query suggestions for knowledge graph exploration. In *The Web Conference*, 2020.
- [25] J. Lu, J. Chen, and C. Zhang. Helsinki Multi-Model Data Repository. <https://www2.helsinki.fi/en/researchgroups/unified-database-management-systems-udbms/>, 2018.
- [26] H. Ma, S. Guan, and Y. Wu. Diversified subgraph query generation with group fairness. In *WSDM*, 2022.
- [27] T. Ma, S. Yu, J. Cao, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.
- [28] D. Mottin, F. Bonchi, and F. Gullo. Graph query reformulation with diversity. In *KDD*, 2015.
- [29] Z. Moumolidou, A. McGregor, and A. Meliou. Diverse data selection under fairness constraints. In *ICDT*, 2021.
- [30] M. H. Namaki, Q. Song, Y. Wu, and S. Yang. Answering why-questions by exemplars in attributed graphs. In *SIGMOD*, 2019.
- [31] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *FOCS*, pages 86–92, 2000.
- [32] M. Poess and J. M. Stephens Jr. Generating thousand benchmark queries in seconds. In *VLDB*, pages 1045–1053, 2004.
- [33] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. P. Hsu, and K. Wang. An overview of microsoft academic service (mas) and applications. *WWW*, 2015.
- [34] Q. Song, M. H. Namaki, and Y. Wu. Answering why-questions for subgraph queries in multi-attributed graphs. In *ICDE*, 2019.
- [35] J. Stoyanovich, K. Yang, and H. Jagadish. Online set selection with fairness and diversity constraints. In *Proceedings of the EDBT Conference*, 2018.
- [36] Y. Wang, Y. Li, J. Fan, C. Ye, and M. Chai. A survey of typical attributed graph queries. *World Wide Web*, 24(1):297–346, 2021.
- [37] Y. Zhang, J. Tang, Z. Yang, J. Pei, and P. S. Yu. Cosnet: Connecting heterogeneous social networks with local and global consistency. In *KDD*, 2015.
- [38] E. Zitzler, J. Knowles, and L. Thiele. *Quality Assessment of Pareto Set Approximations*. 2008.