Geo-Social Group Queries with Minimum Acquaintance Constraint

Qijun Zhu, Haibo Hu, Jianliang Xu
Department of Computer Science
Hong Kong Baptist University
Kowloon Tong, Hong Kong
{qjzhu,haibo,xujl}@comp.hkbu.edu.hk

Wang-Chien Lee
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, USA
wlee@cse.psu.edu

ABSTRACT

The prosperity of location-based social networking services enables geo-social group queries for group-based activity planning and marketing. This paper proposes a new family of geo-social group queries with minimum acquaintance constraint (GSGQs), which are more appealing than existing geo-social group queries in terms of producing a cohesive group that guarantees the worst-case acquaintance level. GSGQs, also specified with various spatial constraints, are more complex than conventional spatial queries; particularly, those with a strict kNN spatial constraint are proved to be NP-hard. For efficient processing of general GSGQ queries on large location-based social networks, we devise two social-aware index structures, namely SaR-tree and SaR*-tree. The latter features a novel clustering technique that considers both spatial and social factors. Based on SaR-tree and SaR*-tree, efficient algorithms are developed to process various GSGQs. Extensive experiments on a real-world Gowalla dataset show that our proposed methods substantially outperform the baseline algorithms based on R-tree.

1. INTRODUCTION

With the ever-growing popularity of smartphone devices, the past few years have witnessed a massive boom in location-based social networking services like Foursquare, Yelp, and Facebook Places. In all these services, mobile users are often associated with some locations (e.g., home/office addresses and visiting places). Such location information, bridging the gap between the physical world and the virtual world of social networks, presents new opportunities for group-based activity planning and marketing [22]. Typical examples include recommending a group gathering of nearby friends when a user visits a city, and a restaurant pushing mobile coupons to groups of socially close friends in location-based advertisements. As these group queries consider both spatial and social constraints, we call them *geo-social group queries* in location-based social networks (LBSNs).

While research attention has recently been drawn to geo-social group queries (e.g., [14, 22]), existing works tend to impose some *loose* social constraint in the query. More specifically, in [14] the circle-of-friend query targets at finding a set of k users such that the maximal weighted spatial and social distance among the users is minimized. Since the social constraint is only approximated by the social distance, users in the result group could have diverse social relations. For example, it is possible that most users in the result group are not familiar with each other. As an improvement, the socio-spatial group query proposed in [22] aims to find k spatially close users among which the *average* number of unfamiliar users does not exceed a threshold p. While the use of threshold p effectively reduces the chance of including unfamiliar users in a result group, the intended social atmosphere in the group may not be achieved, as there is no guarantee on the *minimum* number of friends a group member can get. In the worst case, some user may be unfamiliar with all other users (see experiment results in Section 7.2). Moreover, both queries require tailor-made user inputs. For example, [14] imposes a unified metric for social and spatial distances, and the average unfamiliar number in [22] is hard to choose as different users may have different preferences in getting along with unfamiliar people. Finally, these works mainly focused on in-memory processing (e.g., improving the user scanning order and filtering the candidate combinations), and pay little attention to external-memory indexes. Therefore, they cannot adapt to large and real-world LBSNs.

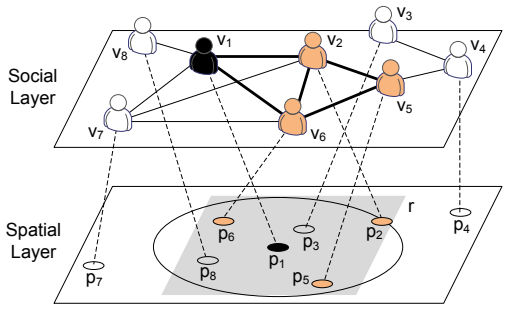


Figure 1: An example of GSGQ$<v_1, 3NN, 2>$. Lines between the users represent acquaintance relations and the points on the spatial layer denote the positions of the users.

In this paper, we propose a new family of Geo-Social Group Queries with *minimum* acquaintance constraint (hereafter called GSGQs for short) on LBSNs. A GSGQ query takes three arguments: (q, Λ, c), where q is the query issuer, Λ is the spatial constraint, and c is the acquaintance constraint. The spatial constraint Λ can be a range query, a k-nearest-neighbor (kNN) query or a relaxed k-nearest-neighbor (rkNN) query, where kNN (resp. rkNN) means the result group has exact (resp. no fewer than) k users and the distance from the farthest member to the query issuer is minimized. The acquaintance constraint imposes a minimum degree c on the familiarity of group members (which may include q), i.e., every user in the group should be familiar with at least c other users. The minimum degree constraint is an important measure of group cohesiveness in social science research [18]. Known as c-core, it has been widely investigated in the research of graph problems [1,

5, 16] and accepted as an important constraint in practical applications [19]. We argue that, compared to the geo-social group queries studied in prior work [14, 22], our GSGQs, with the adoption of a minimum acquaintance constraint, are more appealing to produce a cohesive group that guarantees the worst-case acquaintance level.

Fig. 1 illustrates an example of GSGQ, where the social network is split into a social layer and a spatial layer for clarify of presentation. Suppose user v_1 wants to arrange a friend gathering of some friends nearby. To have a friendly atmosphere in the gathering, she prefers that any one in the group should be familiar with at least two other users. Thus, she may issue a GSGQ $= (q, \Lambda, c)$ with q set as v_1, Λ being 3NN, and $c = 2$. With the objective of minimizing the spatial distance between q and the farthest user in the group, the result group she will obtain is $W = \{v_2, v_5, v_6\}$. Alternatively, to find an acquainted group of friends within a certain range, she may issue a GSGQ $= (q, \Lambda, c)$ with q set as v_1, Λ being r (shaded area in Fig. 1), and $c = 2$. In this example, she will also obtain $W = \{v_2, v_5, v_6\}$.

In this paper, we consider GSGQs under three variants of spatial constraints, including range, relaxed kNN, and strict kNN. These GSGQs are much more complex to process than conventional spatial queries. Particularly, when the spatial constraint is strict kNN, we prove that GSGQs are NP-hard. Due to the additional social constraint, traditional spatial query processing techniques [9, 11, 3, 17] cannot be directly applied to GSGQs. Moreover, our problem is different from other variants of spatial queries, such as spatial-keyword queries [8, 23, 20] and collective spatial keyword queries [4], which only introduce *independent* attributes (e.g., text descriptions) of the objects but not the mutual acquaintance relationship among the users.

On the other hand, most previous works on group queries in social networks use sequential scan in query processing. That is, they enumerate every possible combination of a user group and optimize the processing through some pruning heuristics. Although [22] proposed an SR-tree to cluster the users of each leaf node, this index can only reduce the computation cost of query processing, but not the disk access cost. Besides, its filtering techniques only work for average-degree social constraint, and are not suitable for GSGQs with minimum-degree social constraint. In this paper, we propose two novel social-aware spatial indexing structures, namely, SaR-tree and SaR*-tree, for efficient processing of general GSGQ queries on external storage. The main idea is to project the social relations of an LBSN on the spatial layer and then index both social and spatial relations in a uniform tree structure to facilitate GSGQ processing. Furthermore, we optimize the in-memory processing of GSGQs with a strict kNN constraint by devising powerful pruning strategies. To sum up, the main contributions of this paper are as follows:

- We propose a new family of geo-social group queries with minimum acquaintance constraint (GSGQs), which guarantees the worst-case acquaintance level. We prove that the GSGQs with a strict kNN spatial constraint are NP-hard.
- We design new social-aware index structures, namely SaR-tree and SaR*-tree, for GSGQs. To optimize the I/O access and processing cost, a novel clustering technique that considers both spatial and social factors is proposed in the SaR*-tree. The update procedures of both indexes are also presented.
- Based on the SaR-tree and SaR*-tree, efficient algorithms are developed to process various GSGQs. Moreover, in-memory optimizations are proposed for GSGQs with a strict kNN constraint.
- We conduct extensive experiments to demonstrate the performance of our proposed indexes and algorithms.

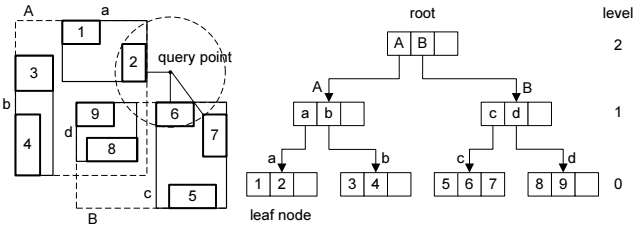


Figure 2: An example of R-tree.

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 introduces some core concepts in the social constraint and formalizes the problems of GSGQs. Section 4 presents the designs of basic SaR-tree and optimized SaR*-tree. Section 5 details the processing methods for various GSGQs based on SaR-trees. Section 6 describes the update algorithms of SaR-trees. Section 7 evaluates the performance of our proposals. Finally, Section 8 concludes the paper and discusses future directions.

2. RELATED WORKS

2.1 Spatial Query Processing

Most spatial databases use R-tree or its extensions [11, 3] as an access method to answer spatial queries (e.g., range, kNN, and spatial join queries). Fig. 2 shows nine objects in a two-dimensional space and how they are aggregated into Minimum Bounding Rectangles (MBRs) recursively to build up the corresponding R-tree. An R-tree node is composed of a number of entries, each covering a set of objects and using an MBR to bound them. A query is processed by traversing the R-tree from the root node all the way down to leaf nodes for qualified objects. During this process, a priority queue H can be used to maintain the entries to be explored. A generic query evaluation procedure for a query Q can be summarized as follows: (1) push the root entries into H; (2) pop up the top entry e from H; (3) if e is a leaf entry, check if the corresponding object is a result object; otherwise push all qualified child entries of e into H; (4) repeat (2) and (3) until H is empty or a termination condition of Q is satisfied.

Some variants of spatial queries have been studied with the consideration of certain grouping semantics. The *group nearest-neighbor* query [17] extends the concept of the nearest neighbor query by considering a group of query points. It targets at finding a set of data points with the smallest sum of distances to all the query points. Based on R-tree, [17] proposes various pruning heuristics to efficiently process group nearest-neighbor queries. The *spatial-keyword* query is another well-known extension of spatial queries that exploits both locations and textual descriptions of the objects. Most solutions for this query, e.g., BR*-tree [23], IR^2-tree [8], and IR-tree [20], rely on combining the inverted index, which was designed for keyword search, with a conventional R-tree. The *collective spatial keyword* query [4] further considers the problem of retrieving a group of spatial web objects such that the group's keywords cover the query's keywords and the objects, with the shortest inter-object distances, are nearest to the query location. Based on IR-tree, [4] proposes dynamic programming algorithms for exact query processing and greedy algorithms for approximate query processing. It is noteworthy that, while these works deal with some grouping semantics, they do not consider acquaintance relations in social networks.

2.2 Social Network Analysis and Query Processing

There have been a lot of works on community discovery in social networks. A typical approach is to optimize the modularity measure [10]. Since communities are usually cohesive subgraphs

formed by the users with the acquaintance relationship, some graph structures such as clique [12], k-core [18], and k-plex [1, 15, 16] have been well studied under this topic. However, most of these works only provide theoretical solutions with asymptotic complexity, with a few exceptions such as the external-memory top-down algorithm for core decomposition [5].

As for query processing in social networks, [7] addresses the problem of finding a subgraph that connects a set of query nodes in a graph. [19] studies a query-dependent variant of the community discovery problem, which finds a dense subgraph that contains the query nodes. Based on a measure of graph density, an optimal greedy algorithm is proposed. The authors of [19] also prove that finding communities of size no larger than a specified upper bound is NP-hard. Besides, [21] proposes a social-temporal group query with acquaintance constraint in social networks. The aim is to find the activity time and attendees with the minimum total social distance to the initiator. As this problem is NP-hard, heuristic-based algorithms have been proposed to reduce the run-time complexity. However, all these works do not consider the spatial dimension of the users and thus cannot be applied to location-based social networks.

2.3 Geo-Social Query Processing

Efficient processing of queries that consider both spatial and social relations is essential for LBSNs. [6] directly combines spatial and social networks and proposes graph-based query processing techniques. More recently, [14] proposes a circle-of-friend query to find minimal-diameter social groups. By transforming the relations in social networks into social distances among users, an integrated distance combining both spatial and social distances is proposed. [22] considers a special socio-spatial group query with the requirement of minimizing the total spatial distance. Accordingly, in-memory pruning and searching schemes are proposed in [22]. All these works only impose a *loose* social constraint in the query. As for the processing techniques, the methods of these works enumerate all possible combinations guided by some searching and pruning schemes. Although a tree structure named SR-tree is introduced in [22], it is used to reduce the checking states during the enumeration. Thus, indexes or external-memory schemes tailored for geo-social query processing in large-scale LBSNs are still lacking in this area.

3. PRELIMINARIES AND PROBLEM STATEMENT

Aiming to find a cohesive group of acquaintances, GSGQs use c-core [18] as the basis of social constraint to restrict the result group. In this section, we first introduce the definition and the properties of c-core, based on which the GSGQ problems are then formalized.

3.1 C-Core

c-core is a degree-based relaxation of clique [18]. Consider an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Given a vertex $v \in V$, we define the set of neighbors of v as $N_G(v) = \{u \in V \mid uv \in E\}$ and the degree of v as $deg_G(v) = |N_G(v)|$. Accordingly, the maximum and minimum degrees of G are represented as $\Delta(G) = max_{v \in V} deg_G(v)$ and $\delta(G) = min_{v \in V} deg_G(v)$, respectively. Let $G[W]$ denote a subgraph induced by $W \subseteq V$. The following is a formal definition of a c-core [18].

DEFINITION 1. ***(c-core)*** *A subgraph $G[W]$ is a* c-core *(or* a core of order c*) if $\delta(G[W]) \geq c$.*

In the sequel, the term c-core refers to both the set W and the subgraph $G[W]$. The *core number* of a vertex v, denoted by c_v, is the highest order of a core that contains this vertex.

A greedy algorithm can be used for *core decomposition*, i.e., finding the core numbers for all vertices in G. The basic idea is to iteratively remove the vertex with the minimum degree in the remaining subgraph, together with all the edges adjacent to it, and determine the core number of that vertex accordingly. The most costly step of this algorithm is sorting the vertices according to their degrees at each iteration. As shown in [2], a bin-sort can be used with $O(|V| + |E|)$ time complexity. Thus, for a given c, we can find the maximum c-core of G in $O(|V| + |E|)$ time.

3.2 Problem Statement

Consider an LBSN $G = (V, E)$, where the set of vertices V denotes the users and the set of edges E denotes the acquaintance relations among the users in V. For any two users $v, u \in V$, there exists an edge $vu \in E$ if and only if v is acquainted with u. Moreover, for any user $v \in V$, its location p_v is also stored in G. Given two users v and u, let $d(v, u)$ denote the spatial distance between v and u, and the largest distance from v to a set of users W is denoted by $d_{max}(v, W) = max_{u \in W} d(v, u)$.

As formally defined below, a GSGQ finds a group of users that satisfies the given spatial and social constraints. Without loss of generality, we assume that the query issuer $q \in V$.

DEFINITION 2. ***(Geo-Social Group Query with Minimum Acquaintance Constraint (GSGQ))*** *Given an LBSN $G = (V, E)$, a GSGQ is represented as $Q_{gs} = (v, \Lambda, c)$, where $v \in V$ is the query issuer, Λ is a type of spatial query denoting the spatial constraint, and c is the minimum degree of result group, denoting the social acquaintance constraint. GSGQ finds a user result set W which satisfies Λ and the condition that the induced subgraph $G[W \cup \{v\}]$ is a c-core, or formally, $\delta(G[W \cup \{v\}]) \geq c$.*

As for the spatial constraint, this paper mainly focuses on three query types: range query, relaxed k-nearest-neighbor (rkNN) query, and strict k-nearest-neighbor (kNN) query. Accordingly, they correspond to three types of GSGQs:

- GSGQ with range constraint, denoted as $GSGQ_{range}$. A $GSGQ_{range}$ is represented as $Q_{gs} = (v, range, c)$, where $p_v \in range$. It targets at finding the largest c-core $W \cup \{v\}$ located inside $range$.
- GSGQ with relaxed kNN constraint, denoted as $GSGQ_{rkNN}$. A $GSGQ_{rkNN}$ is represented as $Q_{gs} = (v, rkNN, c)$. It targets at finding a c-core $W \cup \{v\}$ of size no less than $k + 1$ with the minimum $d_{max}(v, W)$.
- GSGQ with strict kNN constraint, denoted as $GSGQ_{kNN}$. A $GSGQ_{kNN}$ is represented as $Q_{gs} = (v, kNN, c)$. It is a strict form of $GSGQ_{rkNN}$, which requires that the c-core $W \cup \{v\}$ has an exact size of $k + 1$.

For these GSGQs, we prove the following theorems on their complexities. Due to the space limit, detailed proofs of the theorems are omitted in this paper and available at http://www.comp.hkbu.edu.hk/~haibo/papers/GSGQ_VLDB_full.pdf.

THEOREM 1. *$GSGQ_{range}$ and $GSGQ_{rkNN}$ can be solved in polynomial time.*

THEOREM 2. *$GSGQ_{kNN}$ is NP-hard.*

3.3 R-tree based Query Processing

We consider the GSGQ problems for large-scale LBSNs where the user information is stored on external disk storage. A baseline approach of processing GSGQs on an R-tree index of user locations is as follows. For a $GSGQ_{range}$ $Q_{gs} = (v, range, c)$, we first find all users located inside $range$ via R-tree, then compute the c-core W' of the subgraph formed by these users. If v exists in W', then

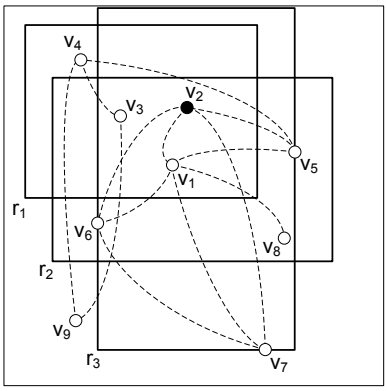
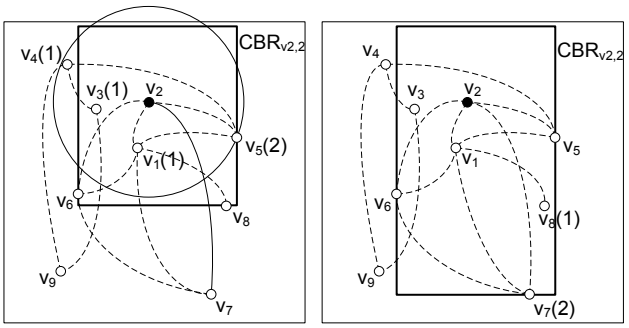


Figure 3: An example of CBR. The LBSN is shown on the spatial layer. The points represent the users as well as their positions, while the dashed lines denote the acquaintance relations among users.



(a) Initialization (b) Expansion

Figure 4: An exemplary procedure of computing $CBR_{v_2,2}$ in an LBSN. The number after a user v_i denotes the core number of v_2 in the subgraph determined by v_i. For a), the subgraph is formed by the users inside $\odot_{v_2,v_i}$; for b), the subgraph is formed by the users inside $CBR_{v_2,2}$ when moving its bottom edge outward to go through v_i.

$W = W' - \{v\}$ is the final result; otherwise, there is no result for Q_{gs}. Since the user filtering step can be done in $O(|V|)$ time and the core decomposition step can be done in $O(|V| + |E|)$ time, the complexity of this method is $O(|V| + |E|)$.

For a $GSGQ_{rkNN}$ $Q_{gs} = (v, rkNN, c)$, we access the users in ascending order of their spatial distances to v. A priority queue H and a candidate result set $\widetilde{W}$ are employed to facilitate query processing. At the beginning, $\widetilde{W}$ is initialized as $\{v\}$ and all the root entries of the R-tree are put into H. Each time the top entry e of H is popped up and processed. If e is a non-leaf entry, its child entries are accessed and put into H; otherwise, e is a leaf entry, i.e., a user, so e is added into $\widetilde{W}$. When the size of $\widetilde{W}$ exceeds k, we compute the c-core W' of the subgraph formed by the users in $\widetilde{W}$. If $|W'| \geq k + 1$ and $v \in W'$, $W = W' - \{v\}$ is the result; otherwise, the above procedure is continued until the result is found. Since each round of c-core detection can be done in $O(|V| + |E|)$ time, the complexity of this method is $O(|V|(|V| + |E|))$.

For a $GSGQ_{kNN}$ $Q_{gs} = (v, kNN, c)$, the processing is similar to $GSGQ_{rkNN}$. The major difference is how to find the result from $\widetilde{W}$. Since the query returns exact k users, all possible user sets of size $k + 1$ and containing v are checked to see if it is a c-core. If such a user set W' exists, then $W = W' - \{v\}$ is the result.

Obviously, these approaches are not suitable for GSGQs with a large c, because a significant number of users will need to be involved to find a c-core, thus leading to inefficiency.

4. SOCIAL-AWARE R-TREES

Since a GSGQ involves both spatial and social constraints, to expedite its processing, both spatial locations and social relations of the users should be indexed simultaneously. Unfortunately, R-tree only indexes spatial locations of the users and is thus inefficient. In this section, we design novel Social-aware R-trees (SaR-trees) to form the basis of our query processing solutions. In what follows, we first introduce the concept of Core Bounding Rectangle (CBR) and then present the details of SaR-tree, followed by a variant SaR*-tree.

4.1 Core Bounding Rectangle (CBR)

The social constraint of a GSGQ requires the result group to be a c-core. Unfortunately, pure social measures such as core number and centrality cannot adequately facilitate GSGQ processing which also features a spatial constraint. To devise effective spatial-dependant social measures to filter users in query processing, in this paper, we develop the concept of *Core Bounding Rectangle (CBR)* by projecting the minimum degree constraint on the spatial layer. Simply put, the CBR of a user v is a rectangle containing v, inside which any user group with v does not satisfy the minimum degree constraint. In other words, it is a localized social measure to a user. As a GSGQ mainly requests the nearby users, the locality of CBR becomes very valuable for processing GSGQs. The formal description of a CBR of user v for a minimum degree constraint c, denoted by $CBR_{v,c}$, is given in Definition 3.

DEFINITION 3. ***(Core Bounding Rectangle (CBR))*** *Consider a user $v \in G$. Given a minimum degree constraint c, $CBR_{v,c}$ is a rectangle which contains v and inside which any user group with v (excluding the users on the bounding edges) cannot be a c-core. Formally, $CBR_{v,c}$ satisfies $p_v \in CBR_{v,c}$ and $\forall W = \{v\} \cup \{u | u \in V, p_u \in CBR_{v,c}\}$ $\delta(G[W]) < c$.*

An example is shown in Fig. 3. According to the acquaintance relations of user v_2, rectangular area r_1 is a $CBR_{v_2,2}$, because any user group inside r_1 that contains v_2 cannot be a 2-core. On the contrary, r_2 is not a $CBR_{v_2,2}$, because some user groups inside r_2 that contain v_2, e.g., $\{v_2, v_1, v_6\}$, are 2-cores. Note that $CBR_{v,c}$ is not unique for a given v and c. For example, r_3 is another $CBR_{v_2,2}$ for user v_2. From Definition 3, we can quickly exclude a user v from the result group by checking $CBR_{v,c}$ during query processing. For example, if the query range of a $GSGQ_{range}$ is covered by $CBR_{v,c}$, then v can be safely pruned from the result. This property makes CBR a powerful pruning mechanism.

Computing CBR of a User. In an LBSN G, given a user v and minimum degree constraint c, a simple method to compute $CBR_{v,c}$ is to search neighboring users in ascending order of distance until there is a user u such that the core number of v in the subgraph formed by the users inside $\odot_{v,u}$ (i.e., the circle centered at v with radius $d(v, u)$) is no less than c, i.e., all user groups located within $\odot_{v,u}$ are not qualified as a c-core. $CBR_{v,c}$ can then be easily derived from $\odot_{v,u}$. An example is shown in Fig. 4(a), where a $CBR_{v_2,2}$ is constructed based on users v_5, v_6, and v_8. This generated CBR satisfies Definition 3 since the users inside it (i.e., v_1, v_2, v_3) cannot form 2-core groups. However, it is not a maximal one, thus limiting its pruning power in GSGQ processing. We improve this initial $CBR_{v,c}$ by recursively expanding it from each bounding edge until no edge can be further moved outward (see Fig. 4(b)) .

Algorithm 1 details the procedure of computing $CBR_{v,c}$. In Line 1, we first sort the users of V in ascending order of their distances to v. In Lines 2-5, we find the nearest user u such that $c_v \geq c$

in the subgraph formed by the users in V with equal or shorter distances to v. In Line 6, we initialize $CBR_{v,c}$ based on u such that $CBR_{v,c}$ does not contain any user outside $\odot_{v,u}$. In Lines 7-9, we expand $CBR_{v,c}$ by moving each bounding edge l of $CBR_{v,c}$ outward, if $c_v < c$ in the subgraph formed by the users inside $CBR_{v,c}$ and on l. Obviously, the rectangle generated by Algorithm 1 is a maximal $CBR_{v,c}$, i.e., it is a $CBR_{v,c}$ and cannot be fully covered by any other $CBR_{v,c}$. This property guarantees its pruning power for GSGQ processing. Fig. 4 provides an exemplary procedure for computing $CBR_{v_2,2}$ when applying Algorithm 1 on an LBSN.

Algorithm 1 Computing CBR of a User

Input: LBSN $G = (V, E)$, user v, constraint c
Output: $CBR_{v,c}$

CompCBR(G, v, c)
1: Sort users of V in ascending order of distances to v;
2: **for** each user u in V **do**
3: Compute c_v in the subgraph formed by the users before (and including) u;
4: **if** $c_v \geq c$ **then**
5: Break;
6: **end if**
7: **end for**
8: Build an initial $CBR_{v,c}$ which goes through u and does not contain any user outside $\odot_{v,u}$;
9: **while** existing a bounding edge l of $CBR_{v,c}$ s.t. $c_v < c$ in the subgraph formed by the users inside $CBR_{v,c}$ and on l **do**
10: Move l outward until $c_v \geq c$ in the subgraph formed by the users inside $CBR_{v,c}$ and on l;
11: **end while**
12: return $CBR_{v,c}$;

To save the computing and storage cost, we only maintain a limited number of CBRs for user v — $CBR_{v,2^0}$, $CBR_{v,2^1}$, $\cdots$, $CBR_{v,2^{\lfloor \log_2 c_v \rfloor}}$ — where c_v is the core number of v in G. We choose CBRs with respect to exponential minimum degree constraints because, as shown in Section 7, the pruning power of CBRs with a large c is significant. Therefore, we do not need dense CBRs for large c.

Complexity Analysis. Let $n = |V|$ and $m = |E|$. In Algorithm 1, the sorting step, i.e., Line 1, requires $O(nlogn)$ time complexity. Since the core number of a user in graph G can be computed in $O(n + m)$ time, initializing $CBR_{v,c}$ in Lines 2-6 requires $O((n + m)n)$ time complexity. During CBR expansion in Lines 7-8, the movement of a bounding edge requires $O((n+m)n)$ time complexity. In total, the time complexity of Algorithm 1 is $O((n + m)n)$. By applying a binary search to find a proper u in CBR initialization and expansion, the time complexity can be reduced to $O((n + m)logn)$. Usually, $m > n$ in an LBSN, so the time complexity of Algorithm 1 is $O(mlogn)$.

4.2 SaR-tree

We now present the basic SaR-tree. It is a variant of R-tree in which each entry further maintains some aggregate social-relation information for the users covered by this entry. Fig. 5 exemplifies an SaR-tree. Different from a conventional R-tree, each entry of an SaR-tree stores two pieces of information, i.e., a set of CBRs (detailed below) and an MBR, to describe the group of users it covers. An example of the former, $CBRs_b$ is shown in the figure. It comprises the core number c_b and two CBRs $\{CBR_{b,1}, CBR_{b,2}\}$ for entry b. The core number of an entry is the maximum core number of the users it covers, which bounds the number of CBRs of this entry. Perceptually, a CBR in the SaR-tree bounds a group of users from the social perspective while an MBR bounds the users from the spatial perspective. As such, SaR-tree gains the power for both

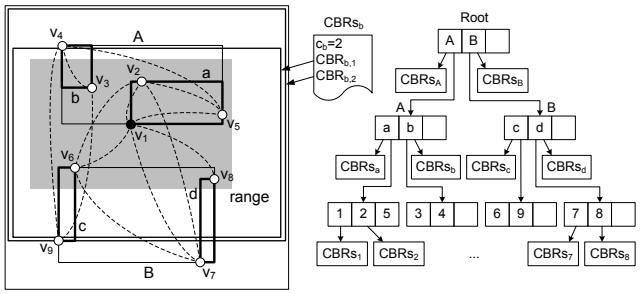


Figure 5: The design of SaR-tree. $CBRs_e$ denotes the set of CBRs for an entry e.

social-based and spatial-based pruning during GSGQ processing, as will be explained in the next section.

CBR of an Entry. To define the CBRs for each SaR-tree entry, we extend the concept of CBR defined for each individual user (in the previous subsection). Let MBR_e and V_e denote the MBR and the set of users covered by an entry e, respectively. A CBR of e is a rectangle which intersects MBR_e and inside which any user group containing any user from V_e cannot satisfy the minimum degree constraint. The formal definition of a CBR of entry e with respect to a minimum degree constraint c, denoted by $CBR_{e,c}$, is given as follows:

DEFINITION 4. ***(CBR of an Entry)*** *Consider an entry e with MBR MBR_e and user set V_e. Given a minimum degree constraint c, $CBR_{e,c}$ is a rectangle which intersects MBR_e and inside which any user group containing any user from V_e (not including the users on the bounding edges) cannot be a c-core.*

Note that $CBR_{e,c}$ is required to intersect MBR_e to guarantee its locality. Fig. 5 shows two examples of CBRs for an entry b, where $V_b = \{v_3, v_4\}$. We can see that any user group inside $CBR_{b,2}$ and containing v_3 or v_4 (not including v_9 on the bounding edges) cannot be a 2-core. Thus, during GSGQ processing, we may safely prune entry e, for example, if the query range of a $GSGQ_{range}$ (with a minimum degree constraint of 2) is fully covered by $CBR_{b,2}$. Since $CBR_{e,c}$ is determined by the set of users in V_e, we use $CBR_{V_e,c}$ and $CBR_{e,c}$ interchangeably.

To efficiently generate the CBRs of the entries in SaR-tree, we adopt a bottom-up approach in our implementation. Obviously, the CBR of a leaf entry e is just the CBR of the user it covers. For a non-leaf entry e, let $e_1, e_2, \cdots, e_m$ be the child entries of e. Then, the CBR of e can be computed by recursively applying the following function on $CBR_{e_1}, \ldots, CBR_{e_m}$:

$$CBR_{\{e_1,\ldots,e_{i+1}\},c} = \begin{cases} CBR_{\{e_1,\ldots,e_i\},c}, \\ \quad MBR_{e_{i+1}} \cap CBR_{\{e_1,\ldots,e_i\},c} = \phi \\ CBR_{\{e_1,\ldots,e_i\},c} \cap CBR_{e_{i+1},c}, \\ \quad \text{otherwise} \end{cases}$$

Finally, $CBR_{e,c} = CBR_{\{e_1,\ldots,e_m\},c}$. It is easy to verify that the CBRs of the entries generated by the above approach satisfy Definition 4.

For an entry e, similar to a user, we only store the CBRs of e with respect to minimum degree constraints $2^0, 2^1, \cdots, 2^{\lfloor \log_2 c_e \rfloor}$, where $c_e = max_{v \in V_e} c_v$ is the core number of e. Let c_G denote the maximum core number of the users in G and s denote the minimum fanout of an SaR-tree. The total number of CBRs in an SaR-tree can be estimated as,

$$n_{CBR} \leq \sum_{v \in V} (\lfloor \log_2 c_v \rfloor + 1) + \frac{2n(\lfloor \log_2 c_G \rfloor + 1)}{s}$$
$$\leq n(\lfloor \log_2 \frac{\sum_{v \in V} c_v}{n} \rfloor + \frac{2(\lfloor \log_2 c_G \rfloor + 1)}{s} + 1).$$

Since c_G and $\frac{\sum_{v \in V} c_v}{n}$ are quite small in a typical LBSN (e.g., they are 43 and 4.5 for the Gowalla dataset used in our experiments), the storage cost of CBRs is comparable to G (e.g., around $2.3n$ in our experiments).

Based on the concept of CBRs, SaR-tree can be directly built on top of R-tree. That is, we first construct a standard R-tree based on the locations of the users and then embed the CBRs into each entry. In this way, SaR-tree indexes both spatial locations and social relations of the users. Note that the users in SaR-tree are organized merely based on their locations — they are spatially close, but may not be well clustered in terms of their social relations. This unfortunately weakens the pruning power of SaR-tree in processing GSGQs. To overcome this weakness, we propose a variant in the next subsection.

4.3 SaR*-tree

SaR*-tree uses the same index structure as SaR-tree but adopts a new user organizing method that integrates both spatial and social relations. Specifically, instead of minimizing the areas of MBRs, SaR*-tree defines a new metric that combines CBRs and MBRs for a group of users V to measure their combined social and spatial closenesses:

$$I(V) = ||MBR_V|| \cdot \sum_c (|| \cup_{v \in V} CBR_{v,c} - CBR_{V,c}||) \quad (1)$$

where $||\cdot||$ is the area of an MBR or CBR. Obviously, a small $I(V)$ indicates that the users of V have both close locations and similar CBRs.

SaR*-tree is constructed by iteratively inserting users. During this construction, CBRs and MBRs are generated at the same time and used for further user insertion. Moreover, if a node N of an SaR*-tree overflows, it will be split. The details about these two main operations in SaR*-tree construction, i.e., *user insertion* and *node split*, are described below.

- *User insertion.* When a user v is inserted into an SaR*-tree, for a node N with entries $e_1, e_2, \cdots, e_m$, we will select the entry e_i with the minimal $I(V_{e_i} \cup \{v\})$ to insert v.
- *Node split.* When a node N of an SaR*-tree overflows, we split N into two sets of entries N_1 and N_2 with the minimal $I(\cup_{e_i \in N_1} V_{e_i}) + I(\cup_{e_j \in N_2} V_{e_j})$. Then, the parent node of n use two entries to point to n_1 and n_2, respectively. This splitting may propagate upwards until the root.

5. GSGQ PROCESSING

In this section, we present the detailed processing algorithms based on SaR-trees for various GSGQs. As mentioned in Section 3, we mainly focus on three types of GSGQs, namely, $GSGQ_{range}$, $GSGQ_{rkNN}$, and $GSGQ_{kNN}$. We will show that the CBRs of SaR-trees can be used in different ways for processing these queries.

5.1 GSGQ with Range Constraint

When processing a $GSGQ_{range}$ $Q_{gs} = (v, range, c)$, each entry of the SaR-tree or SaR*-tree that may cover result users will be visited and possibly further explored. Compared to traditional R-trees, which only provide spatial information via MBRs, an SaR-tree or SaR*-tree provides much greater pruning power due to the social information in CBRs. Consider an exemplary GSGQ $Q_{gs} = (v_1, range, 2)$ in Fig. 5, where the shaded area is the query range. When entry b (which covers users v_3 and v_4) is visited, b needs further exploration if we only consider MBR_b like in regular R-tree. However, with $CBR_{b,2}$, we can easily decide that any user group inside the query range and containing any user in V_b (i.e., v_3 or v_4), cannot be a 2-core, because the query range is covered by $CBR_{b,2}$. Since V_b does not contain any result user, we can simply prune entry b from further processing, as formally proved in Theorem 3.

Considering SaR-trees only maintain the CBRs with respect to exponential minimum degree constraints, given a minimum degree c, we use $CBR_{v,2^{\lfloor \log_2 c \rfloor}}$ to represent $CBR_{v,c}$ in $GSGQ_{range}$ processing. Similar ideas are also applied in $GSGQ_{rkNN}$ and $GSGQ_{kNN}$ processing.

THEOREM 3. *For a $GSGQ_{range}$ $Q_{gs} = (v, range, c)$ where $p_v \in range$, any user in V_e of entry e does not belong to the result group if $range \subset CBR_{e,c}$ and $range$ does not contain any bounding edge of $CBR_{e,c}$.*

Algorithm 2 Processing $GSGQ_{range}$

Input: LBSN $G = (V, E)$, $Q_{gs} = (v, range, c)$
Output: Result of Q_{gs}
ProGSGQRange(G, Q_{gs})
1: Get $c' = 2^{\lfloor \log_2 c \rfloor}$;
2: **if** $c_v < c$ or $range \subset CBR_{v,c'}$ **then**
3: return ϕ;
4: **end if**
5: Initialize H with the root entries of index tree;
6: **while** H has non-leaf entries **do**
7: Pop the first non-leaf entry e from H;
8: **for** each child entry e' of e **do**
9: **if** $range \cap MBR_{e'} \neq \phi$ and $c_{e'} \geq c$ and $range \not\subset CBR_{e',c'}$ **then**
10: Put e' into H;
11: **end if**
12: **end for**
13: **end while**
14: Get the users $\widetilde{W}$ corresponding to the entries of H;
15: Compute the maximum c-core W' of $G[\widetilde{W}]$;
16: **if** $v \in W'$ **then**
17: return $W = W' - \{v\}$;
18: **else**
19: return ϕ;
20: **end if**

Algorithm 2 details the procedure of processing a $GSGQ_{range}$ based on an SaR-tree or SaR*-tree. At the beginning, we access the CBR of user v. If $c_v < c$ or $range \subset CBR_{v,2^{\lfloor \log_2 c \rfloor}}$, it means the core number of v is smaller than c in the subgraph formed by the users inside $range$. Thus, we cannot find any c-core containing v inside $range$ and there is no answer to Q_{gs} (Lines 1-3). Otherwise, we move on to find all candidate users $\widetilde{W}$ via the proposed pruning schemes (Lines 4-10). Then, we compute the maximum c-core W' of $G[\widetilde{W}]$ by applying the core-decomposition algorithm (Line 11). If $v \in W'$, $W = W' - \{v\}$ is the answer; otherwise, there is no answer to Q_{gs}.

We again use the example in Fig. 5 to illustrate the pruning power of the proposed algorithm for processing $GSGQ_{range}$. When applying the baseline algorithm based on R-tree, 7 index nodes, i.e., $root$, A, B, a, b, c and d, and 5 users, i.e., v_2, v_3, v_5, v_6 and v_8, need to be accessed. In contrast, in the proposed algorithm, by using both MBRs and CBRs, there is no need to access index node b (as well as its covered user v_3) and user v_8 since $range \subset CBR_{b,2}$ and $range \subset CBR_{v_8,2}$. As a result, only 6 index nodes and 3 users are accessed, achieving a great saving on I/O cost.

5.2 GSGQ with Relaxed kNN Constraint

To process a $GSGQ_{rkNN}$ $Q_{gs} = (v, rkNN, c)$ on an SaR-tree or SaR*-tree, we maintain a priority queue H of entries. H uses $d_e = max\{d(v, MBR_e), d_{in}(v, CBR_{e,c})\}$ of an entry e as the sorting key in the queue. Here $d_{in}(v, CBR_{e,c})$ is the minimum distance from v (which is inside $CBR_{e,c}$) to reach any bounding edge of $CBR_{e,c}$. Formally, let $L_{CBR_{e,c}}$ denote the set of bounding

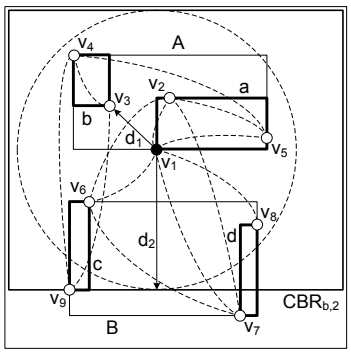


Figure 6: An example of processing a $GSGQ_{rkNN}$ $Q_{gs} = (v_1, r3NN, 2)$.

edges of $CBR_{e,c}$ and $d(v, l)$ denote the distance from v to edge l. Then,

$$d_{in}(v, CBR_{e,c}) = \begin{cases} min_{l \in L_{CBR_{e,c}}} d(v, l), & v \in CBR_{e,c} \\ 0, & \text{otherwise} \end{cases}$$

In implementation, $d_{in}(v, CBR_{e,c})$ is computed on $CBR_{e,2^{\lfloor \log_2 c \rfloor}}$. The rationale of using d_e as the sorting key in the query processing is as follows. By the definition of d_{in}, any user group inside the area $\odot(v, d_{in}(v, CBR_{e,c}))$ and containing any user in V_e cannot be a c-core. In other words, if some users covered by entry e belong to the result group, the maximum distance of the result group to v is expected to be at least d_e, as formally proved in Theorem 4 below. Therefore, we shall explore other entries with keys smaller than d_e before e. Fig. 6 shows an example to demonstrate this rationale. Suppose user v_1 issues a $GSGQ_{rkNN}$ $Q_{gs} = (v_1, r3NN, 2)$. When entry b covering users v_3 and v_4 is visited, we have $d_1 = d(v_1, MBR_b)$ and $d_2 = d_{in}(v_1, CBR_{b,2})$. Then, the key of b is set to be $d_b = max\{d_1, d_2\} = d_2$. We can see that if v_3 or v_4 belongs to the result group, it should also contains v_9 to make the whole group a 2-core, which makes the maximum distance to v_1 larger than d_b. Thus, we can access entry c before b, although c is spatially farther away from v_1 than b. As a result, if a result group W can be obtained from entry c, since $d_{max}(v_1, W) < d_b$, there is no need to visit entry b any longer, thereby saving the access cost.

THEOREM 4. *Given a user v and a minimum degree constraint c, if a user set W makes $G[W \cup \{v\}]$ a c-core, then $d_{max}(v, W) \geq d_e$ for any entry e which covers a user $u \in W$.*

Algorithm 3 presents the details of processing a $GSGQ_{rkNN}$ based on an SaR-tree or SaR*-tree. A set $\widetilde{W}$ is used to store the currently visited users and initialized as $\{v\}$. The entries in H are visited in ascending order of d_e. If a visited entry e is not a leaf entry, it will be further explored and its child entries with $c_{e'} \geq c$ are inserted into H (Lines 7-10); otherwise, we get its corresponding user u (Line 12) and proceed with the following steps. If $c_u < c$, it means u cannot be a result user. Thus, we simply ignore it and continue checking the next entry of H. On the other hand, if $c_u \geq c$, u is added into the candidate set $\widetilde{W}$ (Lines 13-14). Then, we compute the maximum c-core, denoted as W', in the subgraph formed by $\widetilde{W}$ (Line 15). If $|W'| \geq k+1$ and $v \in W'$, $W' - \{v\}$ is the result (Line 16-17); otherwise, the above procedure is continued until the result is found or shown to be non-existent. Theorem 5 proves the correctness of Algorithm 3 and its superiority to the baseline accessing model.

THEOREM 5. *For a $GSGQ_{rkNN}$ $Q_{gs} = (v, rkNN, c)$, Algorithm 3 generates the result of Q_{gs}. Moreover, it incurs equal or less cost than that of the baseline accessing model based on $d(v, MBR_e)$.*

Algorithm 3 Processing $GSGQ_{rkNN}$

Input: LBSN $G = (V, E)$, $Q_{gs} = (v, rkNN, c)$
Output: Result of Q_{gs}

ProGSGQrKNN(G, Q_{gs})
1: **if** $c_v < c$ **then**
2: return ϕ;
3: **end if**
4: $\widetilde{W} = \{v\}$;
5: Initialize H with the root entries of index tree;
6: **while** $H \neq \phi$ **do**
7: Pop the first entry e from H;
8: **if** e is not a leaf entry **then**
9: **for** each child entry e' of e **do**
10: **if** $c_{e'} \geq c$ **then**
11: Compute $d_{e'}$ and put e' into H;
12: **end if**
13: **end for**
14: **else**
15: Get the corresponding user u of e;
16: **if** $c_u \geq c$ **then**
17: $\widetilde{W} = \widetilde{W} \cup \{u\}$;
18: Compute the maximum c-core W' in $\widetilde{W}$;
19: **if** $|W'| \geq k+1$ and $v \in W'$ **then**
20: return $W' - \{v\}$;
21: **end if**
22: **end if**
23: **end if**
24: **end while**
25: return ϕ;

Recall the example in Fig. 6. When applying Algorithm 3 to process $Q_{gs} = (v_1, r3NN, 2)$, the access order of the users is v_2, v_6, v_5, v_3, v_4, v_9 and v_7. The result can be obtained by accessing the first 3 users. During this procedure, 5 index nodes, i.e., $root$, A, B, a and c, and 3 users are accessed and processed. In contrast, the baseline algorithm based on R-tree accesses the users in the order of v_2, v_3, v_6, v_5, v_4, v_8, v_9, and v_7. Then, 6 index nodes and 4 users are accessed and processed. Obviously, by reorganizing the access order of entries, Algorithm 3 processes $GSGQ_{rknn}$ more efficiently.

5.3 GSGQ with Strict kNN Constraint

For a $GSGQ_{kNN}$ $Q_{gs} = (v, kNN, c)$, we adopt the same processing framework as in Algorithm 3. However, when a valid W' is found for $GSGQ_{rkNN}$ at Line 16, more steps will be needed to obtain the result of $GSGQ_{kNN}$. Let W' be the maximum c-core formed by the set of currently visited users $\widetilde{W}$. Only if $|W'| \geq k+1$ and $v \in W'$, it is possible to find a c-core of size $k+1$ in $\widetilde{W}$ that contains v. Moreover, such a c-core must be a subset of W'. Thus, we invoke a function *FindExactkNN* to check all user sets of size $k+1$ that contain v in W'. If such a user set W'' is found, $W'' - \{v\}$ is the result of Q_{gs}; otherwise, the above procedure is repeated when Algorithm 3 continues to find the next candidate W'.

In-Memory Optimizations. The above processing framework provides optimized node access on SaR-trees for $GSGQ_{kNN}$. However, due to the NP-hardness of $GSGQ_{kNN}$, the in-memory processing function *FindExactkNN* also has a great impact on the performance of the algorithm. A naive idea of checking all possible combinations of the user sets is obviously inefficient. In this subsection, we single out this problem to optimize the *FindExactkNN* function by designing two pruning strategies.

Algorithm 4 details the optimized *FindExactkNN*, which employs a *branch-and-bound* method and expands the source user set S from the candidate user set U. At the beginning, S and U are initialized as $\{v, u\}$ and $W' - \{v, u\}$ (u denotes the newly accessed user in Algorithm 3), respectively. Note that if a result W'' exists in W', W'' must contain u, because it has been proved that $W' - \{u\}$ does not contain a result. During the processing, two major pruning strategies, namely, *core-decomposition based pruning* (Lines 6-11, 16-20) and *k-plex based pruning* (Lines 5, 12), are applied.

1) Core-Decomposition based Pruning: Based on the definition of c-core, we can observe that if the current source user set S' can be expanded to a c-core of size $k+1$, it must be contained by the maximum c-core of $U' \cup S'$, where U' denotes the set of remaining candidate users. Therefore, we conduct a core-decomposition on $U' \cup S'$ before further exploration. If a user of S' has a core number smaller than c, S' cannot be expanded to a result from the candidate user set U' and thus we can safely stop further exploration. In addition, if the maximum c-core in $U' \cup S'$ contains S' and has size $k+1$, it is the result of $GSGQ_{kNN}$ and the whole processing terminates. Otherwise, further exploration on the maximum c-core of $U' \cup S'$ is required. Finally, if S' cannot be expanded to a c-core of size $k+1$, we roll back to explore S and the remaining U. Similarly, we compute the maximum c-core W' of $S \cup U$. If $|W'| \geq k+1$ and $S \subseteq W'$, S could be expanded to the result from $U = W' - S$ and further exploration is applied; otherwise, no result can be found.

2) k-plex based Pruning: One major challenge of the c-core problem is that it does not preserve locality, that is, if W is a c-core, adding or dropping some users from W no longer retains it as a c-core. As a workaround, we transfer the problem to a dual $\bar{c}$-plex problem [1] (which preserves the locality property) by adding some constraint. Simply speaking, a $\bar{c}$-plex $W \subseteq V$ is a set such that $\delta(G[W]) \geq |W| - \bar{c}$.

Since a c-core of size $k+1$ is also a $(k+1-c)$-plex, we seek to find a $(k+1-c)$-plex of size $k+1$ to achieve further pruning. $\bar{c}$-plex preserves the locality property because if W is a $\bar{c}$-plex, dropping some users can still make it a $\bar{c}$-plex. In other words, if the maximum $(k+1-c)$-plex in $U' \cup S'$ has a size no less than $k+1$, it is certain that a $(k+1-c)$-plex of size $k+1$ can be found; otherwise, such a $(k+1-c)$-plex cannot be found. Moveover, $(k+1-c)$-plex is more constrained than c-core because the size of the maximum $(k+1-c)$-plex is always no larger than that of the maximum c-core of size no smaller than $k+1$.

The properties of $(k+1-c)$-plex can be used to devise powerful pruning strategies in processing $GSGQ_{kNN}$. First, we prune those users in U who cannot expand the source user set S' to a $(k+1-c)$-plex. This pruning is implemented in Line 5 of Algorithm 4. Second, we estimate the size of a maximum $(k+1-c)$-plex to provide further pruning. Some theoretic bounds on it have been proposed in the literature. In this paper, we adopt the result of [15] and compute an upper bound B on the size of a maximum $(k+1-c)$-plex in a graph G as,

$$B_p(G) = min_{i=1,\ldots,p}\{\frac{1}{i}B(C^i_1, \ldots, C^i_{m_i})\}, \quad (2)$$

and

$$B(C^i_1, \ldots, C^i_{m_i}) = \sum_{j=1}^{m_i} min\{2\bar{c} - 2 + \bar{c}\, mod\, 2, \bar{c} + a_{i,j}, \Delta(G[C^i_j]) + \bar{c}, |C^i_j|\},$$

where $\bar{c} = k+1-c$, $C^i_1, \ldots, C^i_{m_i}$ are co-$\bar{c}$-plexes [15] in which every vertex of V appears exactly i times, $a_{i,j} = max\{n : |\{v | v \in V \wedge deg_G(v) \geq n\}| \geq \bar{c} + l\}$ for each C^i_j, and p is a parameter to limit the iterations of computing.

Fig. 7 shows the steps of both the basic and optimized version of function *FindExactkNN* where user set $W' = \{v_1, v_2, v_3, v_6, v_9, v_8, v_4, v_7\}$ and $Q_{gs} = (v_1, 3NN, 2)$. In the optimized procedure, each step shows the investigated source user set S' and the candidate set U' after filtering. For example, in the first step, we try to check $S' = \{v_1, v_7, v_4\}$ and $U' = \{v_2, v_3, v_6, v_8, v_9\}$. After filtering U' via Line 5 of Algorithm 4, we can get $U' = \{v_2, v_6, v_8, v_9\}$. Since the maximum 2-core of $U' \cup S'$ only has size 3, no 2-core of size 4 can be found in $U' \cup S'$. Thus, all the combinations of these users can be ignored. A similar case can be found in the second step when $S' = \{v_1, v_7, v_9\}$. In the third step, we can get the upper bound of the size of the maximum 2-plex in $U' \cup S'$ as 3 by computing $B_2(G[U' \cup S'])$. Thus, $U' \cup S'$ does not contain a 2-core of size 4. We can stop searching here because no user is filtered from U' in the last step, which means all the combinations are covered. We can see that the optimized function *FindExactkNN* effectively prunes unnecessary explorations and saves significant computation cost.

Algorithm 4 Finding c-core of size $k+1$

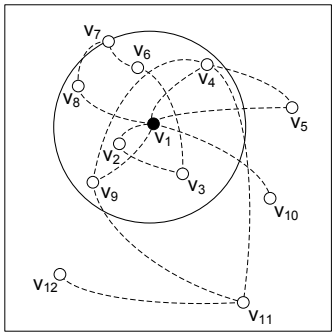
Input: User set U and S, c, k
Output: c-core W

FindExactkNN(U, S, c, k)
1: **if** $|S| = k+1$ **then**
2: return S;
3: **end if**
4: **while** $U \neq \phi$ **do**
5: $S' = S \cup \{u\}$, $U = U - \{u\}$ for some $u \in U$;
6: $U' = \{u \in U : S' \cup \{u\}$ is a $(k+1-c)$-plex $\}$;
7: Compute the maximum c-core W' of $U' \cup S'$;
8: **if** $|W'| \geq k+1$ and $S' \subseteq W'$ **then**
9: **if** $|W'| = k+1$ **then**
10: return W';
11: **else**
12: $U' = W' - S'$;
13: **if** $b_p(G[U' \cup S']) \geq k+1$ **then**
14: W'' = *FindExactkNN*(U', S', c, k);
15: **if** $W'' \neq \phi$ **then**
16: return W'';
17: **end if**
18: **end if**
19: **end if**
20: **end if**
21: Compute the maximum c-core W' of $S \cup U$;
22: **if** $|W'| \geq k+1$ and $S \subseteq W'$ **then**
23: $U = W' - S$;
24: **else**
25: break;
26: **end if**
27: **end while**
28: return ϕ;

6. UPDATE OF SAR-TREES

The SaR-trees, once built, can be used as underlying structures for efficient GSGQ processing with generic spatial constraints. It is particularly favorable for applications where both social relations and user locations (e.g., home addresses) are stable. However, for other applications where users may regularly change their locations and social relations, efficient update of the SaR-trees is required. This is challenging because an update of a user affects not only her own CBR but also those of others. In this section, we propose a lazy update approach tailored for SaR-trees that achieves high update efficiency while still guaranteeing the effectiveness of GSGQ processing.

6.1 Lazy Update in SaR-trees



W'={$v_1,v_2,v_3,v_6,v_9,v_8,v_4,v_7$}

Basic:
Check all 4-user subsets
Cost: 70 combinations

Optimized:
S'={v_1,v_7} U'={v_2,v_3,v_4,v_6,v_8,v_9}
1.S'={v_1,v_7,v_4} U'={v_2,v_6,v_8,v_9}
=> pruned by core-decomposition
2.S'={v_1,v_7,v_9} U'={v_2,v_6,v_8}
=> pruned by core-decomposition
3.S'={v_1,v_7,v_3} U'={v_2,v_6,v_8}
Get C_1={v_1,v_2,v_6,v_7},C_2={v_3,v_8,v_1,v_6},
C_3={v_2,v_7} => $b_2(G[U' \cup S'])=3$
=> pruned by k-plex model
Cost: 3 combinations

Figure 7: Exemplary original and optimized function *FindExactkNN* when $W' = \{v_1, v_2, v_3, v_6, v_9, v_8, v_4, v_7\}$ for $GSGQ_{kNN}$ $Q_{gs} = (v_1, 3NN, 2)$. The entries are omitted here because they are not related to function *FindExactkNN*.

An update from user $v \in G$ means either her location changes from p_v to p'_v or her social relation $N_G(v)$ changes. We establish the following two rules on updating CBRs due to location and social updates, respectively.

UPDATE RULE 1. *For location updates, a $CBR_{u,c}$ might become invalid only if there exists some user v such that $c \leq c_v$, $p_v \notin CBR_{u,c}$, and $p'_v \in CBR_{u,c}$.*

UPDATE RULE 2. *For social updates, a $CBR_{u,c}$ might become invalid only if there exist two users v, v' such that edge vv' is newly added, $min\{c_v, c_{v'}\} \geq c$ and $\{p_v, p_{v'}\} \in CBR_{u,c}$.*

To relieve an update procedure from intensive CBR recomputation, we propose a lazy update model for SaR-trees. Particularly, a memo M is introduced to store those accumulated updates which have not been applied on the CBRs of SaR-trees. Each time when a user update arrives, the user record is updated, and core-decomposition is performed on G to update the core numbers of users if it is a social update. If the core number of a user u changes, the core numbers of the entries along the path from u to the root are updated. Then, this user update is added into M. When the size of M reaches a threshold, named the *Batch Update Size*, a batch update is applied on the CBRs of SaR-trees.

To facilitate CBR updates, an R-tree is built on the CBRs of users. By a point containment query on this R-tree, we can find the CBRs that cover the latest location of an updated user. The retrieved CBRs are then filtered based on Update Rule 1 and Update Rule 2. For the remaining CBRs, we first determine their validity by computing the core numbers of the corresponding users in the subgraphs formed by the users inside the CBRs. Then, each invalid CBR is recomputed by applying Algorithm 1 and its update is propagated to the root along the SaR-tree path.

6.2 GSGQ Processing with Update-Memo on SaR-trees

With an update-memo M, GSGQ processing algorithms on SaR-trees need to be revised for correctness as some CBRs may be invalid. In the following, we outline the major changes of the processing algorithms for different GSGQs.

$GSGQ_{range}$ processing. To revise Algorithm 2, the CBRs will no longer be used to prune entries when traversing the SaR-tree. As a result, the priority queue H is composed of a number of leaf entries, each corresponding to a user with core number equal to or larger than c inside $range$. As such, for each user u in H s.t. $range \subset CBR_{u,c}$, we check the other users in H located inside *range*: if some other user has updates in M which might invalidate $CBR_{u,c}$ according to Update Rule 1 or 2, we keep u in H; otherwise, u is pruned from H. In the end, if the query issuer v is pruned from H, there will be no result; otherwise, we obtain the result from H as Algorithm 2 does.

$GSGQ_{rkNN}$ (or $GSGQ_{kNN}$) processing. To revise Algorithm 3, we use the second priority queue H' to store the entries of H in ascending order of their minimal distances to v. When putting an entry e into H, if $d_{in}(v, CBR_{e,c}) > d(v, MBR_e)$, we need to verify the validity of $CBR_{e,c}$. For a non-leaf entry e, we simply set $d_e = d(v, MBR_e)$ to avoid the validating cost. For a leaf entry e, let u be the corresponding user. We retrieve all users with shorter distances to the query issuer v than $d_{in}(v, CBR_{e,c})$ by exploring H', denoted as U. Then, we filter out the users in U who has no update in M or cannot invalidate $CBR_{e,c}$ according to Update Rule 1 or 2. If U is not empty, $d_{in}(v, CBR_{e,c})$ is updated as $min_{u' \in U} d(v, u')$. It is easy to verify that if $p_u \in \odot(v, min_{u' \in U} d(v, u'))$, any user group with u inside $\odot(v, min_{u' \in U} d(v, u'))$ cannot be a c-core. This guarantees the correctness of the algorithm.

Table 1: System parameter settings

Parameter	Value	Parameter	Value
c	$1-5$	r	$0.002-0.01$
k	$20-250$	Page size	$4KB$
Page Acc. Time	2ms		
Gowalla			
User #	$107,092$	Edge #	$456,830$
Max degree	9967	Avg. degree	9.177
Max core num.	43	Avg. core num.	4.839
Dianping			
User #	$2,673,970$	Edge #	$922,977$
Max degree	11423	Avg. degree	5.184
Max core num.	24	Avg. core num.	2.741

Table 2: Minimum degree of the result group given $k = 50$ on Gowalla.

Query	ρ				
	1	2	3	4	5
kNN	0	0	0	0	0
$SSGQ(p = \rho)$	0.05	0.08	0.11	0.16	0.21
$GSGQ_{rkNN}(c = \rho)$	1	2	3	4	5

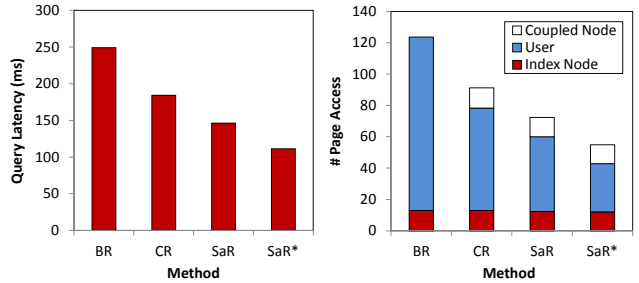
7. PERFORMANCE EVALUATION

In this section, we evaluate the proposed methods on two real datasets, namely, *Gowalla* and *Dianping*, and investigate the impact of various parameters. The code is written in C++ and compiled by GNU gcc x64 4.5.2. All the experiments are performed on a Ubuntu server with Intel Core i7 2.13 GHz CPU and 2GB RAM.

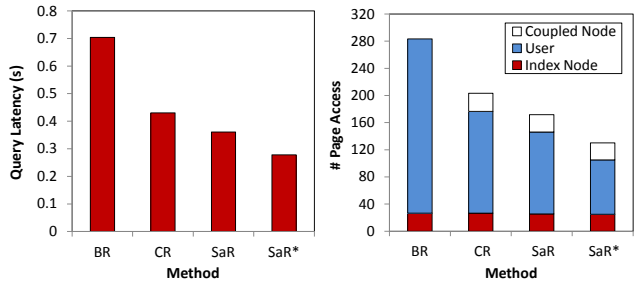
7.1 Experimental Setting

The Gowalla dataset was collected from a location-based social network and made available on website http://snap.stanford.edu/data/loc-gowalla.ht while the Dianping dataset was collected from a Chinese restaurant review site by ourselves. Both datasets have location-based social networks that consist of users (nodes), friend relations (edges), and user check-ins (locations). To enable our experiments, we prune users with no check-ins and select the first check-in position of each user as his/her location of interest. As a result, the purified Gowalla dataset has 107,092 nodes and $456,830$ edges, and the purified Dianping dataset has 2,673,970 nodes and $922,977$ edges. Finally, we project the space of the locations to domain $[0, 1]$ in each dimension.

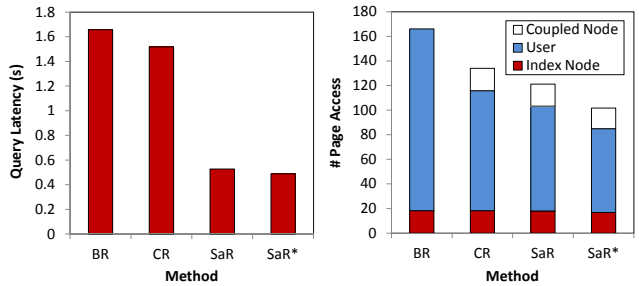
We implement four indexes for performance evaluation, namely, *R-tree*, *C-imbedded R-tree*, *SaR-tree*, and *SaR*-tree*. C-imbedded R-tree also stores the core numbers of the index entries. The sizes of SaR-trees are 15.5MB for Gowalla and 257MB for Dianping. The corresponding GSGQ processing methods on these indexes are denoted as *BR* (baseline R-tree), *CR*, *SaR* and *SaR**, respectively.



(a) $GSGQ_{range}$ (r=0.002, c=4) (b) $GSGQ_{range}$ (r=0.002, c=4)

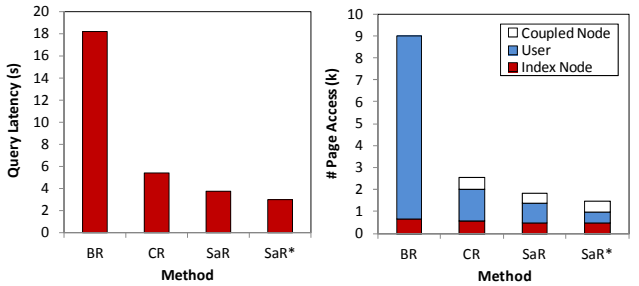


(c) $GSGQ_{rkNN}$ (k=100, c=4) (d) $GSGQ_{rkNN}$ (k=100, c=4)

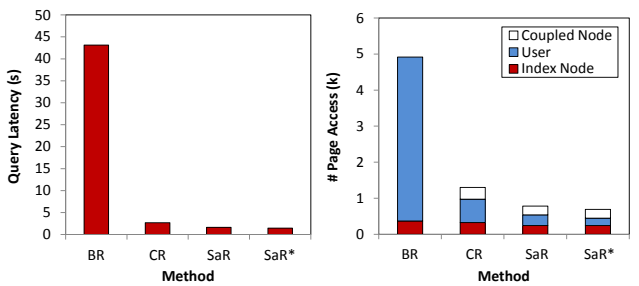


(e) $GSGQ_{kNN}$ (k=100, c=3) (f) $GSGQ_{kNN}$ (k=100, c=3)

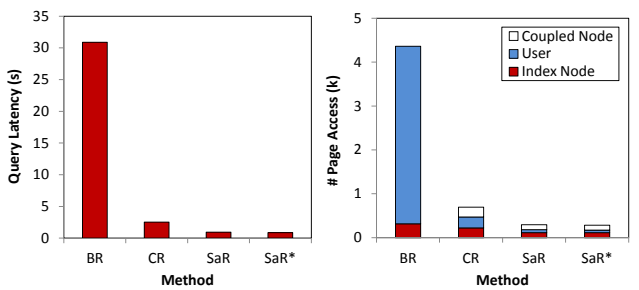
Figure 8: Overall performance comparison on Gowalla.



(a) $GSGQ_{range}$ (r=0.002, c=4) (b) $GSGQ_{range}$ (r=0.002, c=4)



(c) $GSGQ_{rkNN}$ (k=100, c=4) (d) $GSGQ_{rkNN}$ (k=100, c=4)



(e) $GSGQ_{kNN}$ (k=30, c=8) (f) $GSGQ_{kNN}$ (k=30, c=8)

Figure 9: Overall performance comparison on Dianping.

CR enhances from *BR* by pruning those nodes whose core numbers cannot satisfy the minimum degree constraint c.

To have a fair comparison, we implement CR, SaR, and SaR* by coupling extra pages with each index node to store the information of CBRs (for SaR and SaR*) or core numbers (for CR). These extra pages are called the *coupled nodes*. To evaluate the performance of the methods, we mainly use two metrics, namely, the page access cost and the query latency. The former includes the page accesses of index nodes, coupled nodes, and user data. On the other hand, the query latency measures the actual clock time to process a GSGQ, including the CPU time and the I/O time. In the experiments, no cache is used for GSGQ processing and the page access time is set as $2ms$. Each test ran the same 1,000 randomly generated GSGQs and we report the average performance.

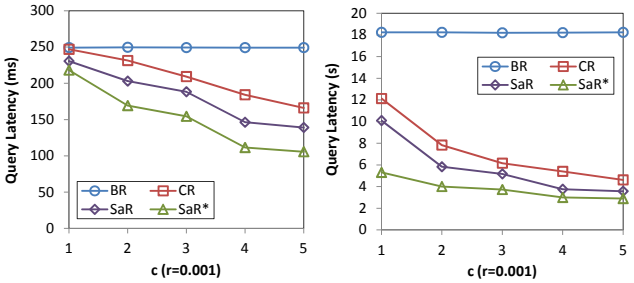
Three query types, namely, $GSGQ_{range}$, $GSGQ_{rkNN}$, and $GSGQ_{kNN}$, are tested. For a $GSGQ_{range}$, the range r is defined as a square centered at the location of the query issuer. Therefore, in the following, we use the edge length to represent r, which is set at 0.002 by default. For $GSGQ_{rkNN}$ and $GSGQ_{kNN}$, k is selected from 20 to 250, which represents large-scale time-consuming queries for real-life social applications, e.g., the marketing example shown in Section 1. Finally, the minimum degree constraint c is selected from 1 to 5. Table 1 summarizes the major parameters and their values used in the experiments, where the average degree only counts connected nodes.

7.2 Overall Performance

Table 2 summarizes the average minimum degree of the result groups for three different query semantics on Gowalla, where kNN denotes a classic k-nearest-neighbor query and $SSGQ$ denotes the socio-spatial group query proposed in [22]. As expected, GSGQ always retrieves groups that satisfy the minimum degree constraints, while the other two query types have close-to-zero minimum degree. This justifies the improved social constraint introduced by GSGQ.

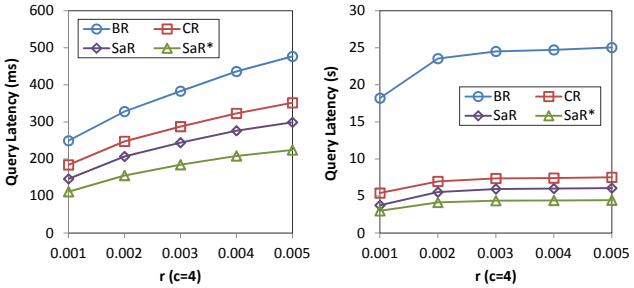
Fig. 8 and Fig. 9 show the overall performance of the GSGQ methods under three different queries on Gowalla and Dianping, respectively. Generally, CR, SaR, and SaR* achieve significant improvement over BR. Moreover, SaR and SaR* outperform CR in all the cases tested. Take Gowalla as an example. For $GSGQ_{range}$, CR, SaR, and SaR* outperform BR by 26.1%, 41.3%, and 55.3%, respectively, in terms of the query latency (see Fig. 8(a)). This is mainly due to the savings in access the index data and user data. Similar improvement can be found with respect to page accesses in Fig. 8(b). More specifically, SaR and SaR* check much fewer users (around 182.2 users) than CR (around 290.4 users) and BR (around 749.5 users) to derive the results. SaR* further reduces the page accesses to 54.9 compared to SaR (72.3), CR (91.3), and BR (123.6). Finally, compared to CR, SaR (resp. SaR*) saves up to 20.6% (resp. 39.5%) query latency and page accesses, which exhibits the high pruning power of CBRs for $GSGQ_{range}$ processing.

For $GSGQ_{rkNN}$, CR, SaR, and SaR* achieve similar improvement over BR in terms of the query latency and the page access cost as in $GSGQ_{range}$ (see Fig. 8(c) and Fig. 8(d)). The improvement becomes more significant for the CPU time. Specifically, CR, SaR, and SaR* save 82.1%, 86.5%, and 86.8% CPU time of BR,



(a) Gowalla (b) Dianping

Figure 10: Query latency of the methods for $GSGQ_{range}$ queries with different c.



(a) Gowalla (b) Dianping

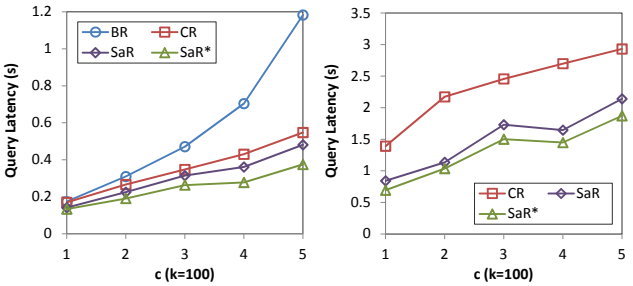
Figure 11: Query latency of the methods for $GSGQ_{range}$ queries with different r.

respectively. The result also shows that SaR (resp. SaR*) outperforms CR by 16.1% (resp. 35.4%) with regard to query latency. For $GSGQ_{kNN}$, the improvement on query latency becomes even more significant for SaR and SaR* because of the in-memory optimizations. That is, compared to BR (resp. CR), the methods SaR and SaR* save 68.2% (resp. 65.3%) and 70.6% (resp. 67.8%) query latency (see Fig. 8(e)). This shows that by optimizing the accessing order of the entries based on the CBRs, a great performance improvement can be achieved.

7.3 $GSGQ_{range}$ Processing

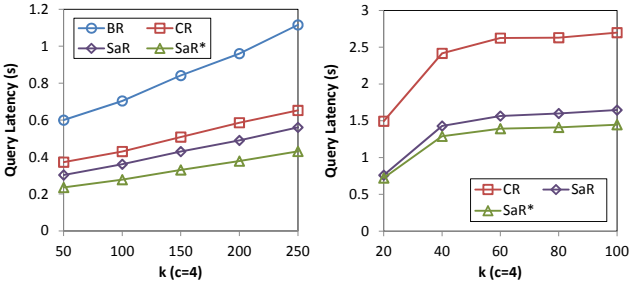
For a $GSGQ_{range}$ $Q_{gs} = (v, r, c)$, Fig. 10 shows the performance with different c and r settings on Gowalla and Dianping. All methods except BR incur shorter query latency for a larger c. The performance gap between BR and CR (or SaR, SaR*) increases as c grows. This is because more users and index nodes can be pruned in CR (and SaR, SaR*) for a large c. Both SaR and SaR* outperform CR in all cases.The improvement reduces a little at $c = 3$ and $c = 5$ because only approximate CBRs are used for query processing in these cases. Moreover, SaR* benefits more from index than CR and SaR, because it groups the users based on both spatial and social closenesses, making the pruning of index nodes and user pages more powerful. While SaR, by using CBRs of the entries, performs better in reducing the page access cost than CR and BR, it falls far behind SaR*. As for various settings of query range r (see Fig. 11), we can observe that the performance of all methods degrade when r grows, because more users within the range need to be checked. In terms of query latency, BR performs significantly worse than the other three methods. On the other hand, SaR* incurs much less page access than other methods and thus is the most favorable approach.

7.4 $GSGQ_{rkNN}$ Processing



(a) Gowalla (b) Dianping

Figure 12: Query latency of the methods for $GSGQ_{rkNN}$ queries with different c (BR: $3.9s \sim 58.5s$ on Dianping).



(a) Gowalla (b) Dianping

Figure 13: Query latency of the methods for $GSGQ_{rkNN}$ queries with different k (BR: $13.7s \sim 43.1s$ on Dianping).

This subsection investigates the performance of the methods for $GSGQ_{rkNN}$ under various c and k settings. As we observed similar performance trends for $GSGQ_{kNN}$ under these settings, we omit the details on $GSGQ_{kNN}$ here.

For a $GSGQ_{rkNN}$ $Q_{gs} = (v, rkNN, c)$, Fig. 12 shows the performance with different c settings on Gowalla and Dianping. All methods incur higher query latency for a larger c. This is because a large c tightens the social constraint of $GSGQ_{rkNN}$ and thus more users need to be visited. Similar to $GSGQ_{range}$, the performance gaps between BR and CR (SaR, SaR*) increase as c grows. This is because as c grows, the candidate users for $GSGQ_{rkNN}$ processing tend to share similar CBRs. Thus, the social-ware user organization of SaR* can effectively reduce the page access.

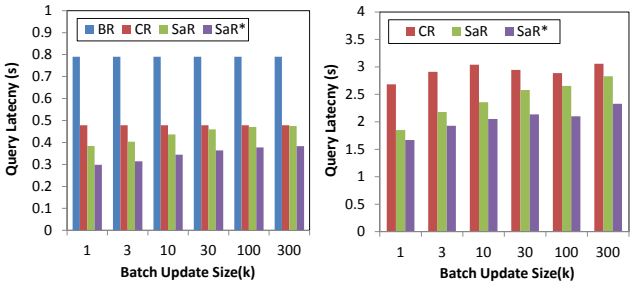
Fig. 13 shows the performance with different k settings on Gowalla and Dianping. Compared to c, the increment of k causes only a moderate increase in cost. CR, SaR, and SaR* beat BR for all k settings and the performance gaps become larger as k grows. Finally, SaR and SaR* are always superior to CR in terms of both query latency and page access cost (not shown here) This implies that the pruning powers of SaR and SaR* are stable under various settings of k.

7.5 Update Performance of SaR-trees

This section investigates the update performance of SaR-trees. We take the locations of user check-ins along the timeline of Gowalla and Dianping to generate location updates and random new edges to generate social updates on users. Due to the fact that social updates are relatively infrequent in real social networks [13], the proportion of social updates is set to 5%. We first investigate the effect of batch update size. In general, the average amortized update time decreases as more updates are applied in a batch processing. This is mainly because fewer CBRs, on average, are required to update as summarized in Table 3. Fig. 14(a) (resp. Fig. 14(b)) shows the performance for the $GSGQ_{rkNN}$ queries with default settings

Table 3: Average # of updated CBRs w.r.t. batch update size.

Batch Upd. Size (k)	1	3	10	30	100	300
Gowalla	13.14	5.71	2.27	1.05	0.45	0.16
Dianping	11.50	5.19	1.95	0.83	0.33	0.11



(a) Gowalla (b) Dianping

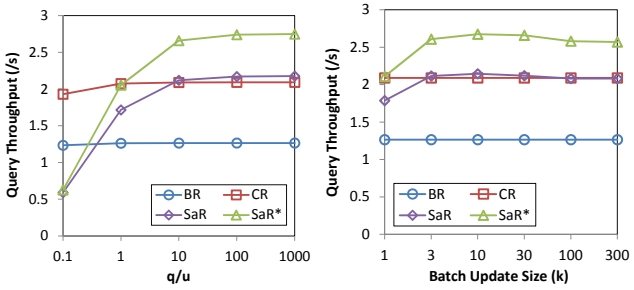
Figure 14: The performance of the lazy update model. Query latency of BR is around $50s$ on Dianping.

under different batch update sizes on Gowalla (resp. Dianping). We can see that the performance of SaR and SaR* degrades as the batch update size grows, which is mainly because more CBRs are invalidated by the updates of M and less pruning power could be achieved (albeit better than BR or CR).

To further measure the impact of updates on query processing, we generate workloads of mixed update and query requests (i.e., the $GSGQ_{rkNN}$ queries with default settings). Fig. 15(a) shows the throughputs under various query/update ratios (workloads) on Gowalla. SaR* and SaR achieve higher throughput than CR when the workload has lighter updates, i.e., $q/u > 1$ and 10, respectively, because the performance gain from query processing can compensate for the additional CBR update cost. Fig. 15(b) shows the thoughputs under different batch update sizes on Gowalla. We can see that SaR outperforms CR only for a range of the batch update size. It is because large batch update size leads to obvious performance degradation of SaR for GSGQ processing, making it can not compensate for the CBR update cost any more. In comparison, SaR* always achieves the highest throughput. Similar results can also be observed on Dianping, which are omitted here due to space limit.

8. CONCLUSION

This paper has studied the problems of processing geo-social group queries (GSGQs) with minimum acquaintance constraint on large location-based social networks. To achieve high efficiency, two new social-aware index structures, namely SaR-tree and SaR*-tree, have been proposed. Based on SaR-trees, we have developed efficient algorithms to process various GSGQs. Extensive experiments on a real-world dataset demonstrate that our proposed methods substantially outperform the baseline methods based on R-tree under various system settings. As for future work, we plan to extend GSGQs to incorporate more sophisticated spatial queries such as skyline and distance based joins.



(a) Batch Update Size = $30k$ (b) $q/u = 10$

Figure 15: The query throughput of the methods on Gowalla.

9. REFERENCES

[1] S. B. B. Balasundaram and I. V. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. In *Operations Research*, 2009.

[2] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. In *CoRR*, 2003.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD*, 1990.

[4] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD Conference*, 2011.

[5] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *Proc. ICDE*, 2011.

[6] Y. Doytsher, B. Galon, and Y. Kanza. Querying geo-social data by bridging spatial networks and social networks. In *ACM LBSN*, 2010.

[7] C. Faloutsos, K. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD*, 2004.

[8] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proc. ICDE*, 2008.

[9] R. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. In *Acta Informatica*, 1974.

[10] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. In *Proceedings of the National Academy of Sciences of the USA*, 2002.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, 1984.

[12] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. In *Sociometry*, 1957.

[13] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins. Microscopic evolution of social networks. In *KDD*, 2008.

[14] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of friend query in geo-social networks. In *DASFFA*, 2012.

[15] B. McClosky and I. V. Hicks. Combinatorial algorithms for max k-plex. In *Journal of Combinatorial Optimization*, 2012.

[16] H. Moser, R. Niedermeier, and M. Sorge. Algorithms and experiments for clique relaxations-finding maximum s-plexes. In *SEA*, 2009.

[17] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, 2004.

[18] S. B. Seidman. Network structure and minimum degree. In *Social Networks*, 1983.

[19] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[20] D. Wu, M. L. Yiu, and C. S. Jensen. Moving spatial keyword queries: Formulation, methods, and analysis. In *ACM Trans. Database Syst.*, 2013.

[21] D.-N. Yang, Y.-L. Chen, W.-C. Lee, and M.-S. Chen. On social-temporal group query with acquaintance constraint. In *Proc. VLDB*, 2011.

[22] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On socio-spatial group query for location-based social networks. In *KDD*, 2012.

[23] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *Proc. ICDE*, 2009.

APPENDIX

A. PROOFS

A.1 Proof of Theorem 1

PROOF. As we will show in Section 3.3, processing a $GSGQ_{range}$ can be completed by running core-decomposition once, while processing a $GSGQ_{rkNN}$ can be completed by running core-decomposition at most $|V|$ times. Since the time complexity of core-decomposition is $O(|V| + |E|)$, both of the queries can be solved in polynomial time. □

A.2 Proof of Theorem 2

PROOF. Considering that a c-core of size $k+1$ is equivalent to a $(k+1-c)$-plex, i.e., a set W such that $\delta(G[W]) \geq |W| - \bar{c}$, we can find a $(k+1-c)$-plex of size $k+1$ by iteratively applying $GSGQ_{kNN}$ for each user v in G. If a c-core of size $k+1$ is found for a user v, then a $(k+1-c)$-plex of size $k+1$ exists; otherwise such a $(k+1-c)$-plex does not exist. In this way, the $\bar{c}$-plex problem, i.e., determining whether there exists a $\bar{c}$-plex for a given G and positive integers $\bar{c}$ and k, can be polynomially reduced to $GSGQ_{kNN}$. Moreover, as proved in [4], the $\bar{c}$-plex problem is NP-complete. This proves that $GSGQ_{kNN}$ is NP-hard. □

A.3 Proof of Theorem 3

PROOF. We prove it by contradiction. If the theorem is not true, i.e., a user $u \in V_e$ belongs to the result group W. Since the users of $W \cup \{v\}$ are located inside $range$ and $range \subset CBR_{e,c}$ does not contain any bounding edge of $CBR_{e,c}$, $W \cup \{v\}$ is a c-core with u inside $CBR_{e,c}$ (not including the users on the bounding edges), which is contradictory to the CBR definition for an entry. □

A.4 Proof of Theorem 4

PROOF. If $d_e > d_{max}(v, W)$, since $d(v, MBR_e) \leq d(v, u) \leq d_{max}(v, W)$, we have $d_{in}(v, CBR_{e,c}) > d_{max}(v, W)$. It means $CBR_{e,c}$ contains u and inside which a c-core $G[W \cup \{v\}]$ with u exists, which is contradictory to the CBR definition for an entry. Thus, $d_{max}(v, W) \geq d_e$ for any entry e which covers a user $u \in W$. □

A.5 Proof of Theorem 5

PROOF. Let W be the user set returned by Algorithm 3. and user $v' = \arg_{u \in W} \max d(v, u)$. If W is not the result of Q_{gs}, suppose another user set W' ($W' \neq W$) is the result. Then, $d_{max}(v, W') < d_{max}(v, W)$. If $\exists u' \in W'$ and $u' \notin W$, it means $\exists e'$ with $u' \in V_{e'}$ and $d_{e'} \geq d_{v'} \geq d_{max}(v, W)$. However, according to Theorem 4, we have $d_{e'} \leq d_{max}(v, W') < d_{max}(v, W)$, which is contradictory to the assumption $d_{e'} \geq d_{max}(v, W)$. So $W' \subseteq W$.

Now consider the last accessed user v'' in Algorithm 3. Since $W' \subseteq W$, we have $v'' \in W'$ (if not, W' is found before W to make $G[W' \cup \{v\}]$ a c-core, which is a contradiction to the fact that W is the first). For $d(v, v') = d_{max}(v, W) > d_{max}(v, W')$, we still have $v' \notin W'$. Thus, v' must be accessed before v''. It means $\exists e''$ with $u'' \in V_{e''}$ and $d_{e''} \geq d_{v'} \geq d_{max}(v, W)$. However, according to Theorem 4, we have $d_{e''} \leq d_{max}(v, W') < d_{max}(v, W)$, which is contradictory to the assumption $d_{e''} \geq d_{max}(v, W)$. To conclude, W is the result of Q_{gs}.

Let S and S' be the entries been further explored by Algorithm 3 and by the baseline accessing model based on $d(v, MBR_e)$, respectively, for finding the result group W. For an entry e_u which covers $u \in W$, based on Theorem 4, we have $d_{e_u} \leq d_{max}(v, W)$. Thus, for any entry $e \in S$, it must satisfy $d_e \leq d_{max}(v, W)$ (if not, e will not be further explored since all the users of W have been accessed and the result group W has been found). Considering that $S' = \{e | d(v, MBR_e) \leq d_{max}(v, W)\}$, then $S \subseteq \{e | e \in S' \wedge d_{in}(v, CBR_{e,c}) \leq d_{max}(v, W)\} \subseteq S'$. □