On Spatial-Aware Community Search

Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, Xiaojun Chen

Abstract—Communities are prevalent in social networks, knowledge graphs, and biological networks. Recently, the topic of community search (CS) has received plenty of attention. The CS problem aims to look for a dense subgraph that contains a query vertex. Existing CS solutions do not consider the spatial extent of a community. They can yield communities whose locations of vertices span large areas. In applications that facilitate setting social events (e.g., finding conference attendees to join a dinner), it is important to find groups of people who are physically close to each other, so it is desirable to have a *spatial-aware community* (or SAC), whose vertices are close structurally and spatially. Given a graph G and a query vertex q, we develop an exact solution to find the SAC containing q, but it cannot scale to large datasets, so we design three approximation algorithms. We further study the problem of continuous SAC search on a "dynamic spatial graph", whose vertices' locations change with time, and propose three fast solutions. We evaluate the solutions on both real and synthetic datasets, and the results show that SACs are better than communities returned by existing solutions. Moreover, our approximation solutions perform accurately and efficiently.

Index Terms—Community search, spatial graphs, geo-social networks, online queries.

1 Introduction

With the emergence of geo-social networks, such as Twitter and Foursquare, the topic of geo-social networks has gained a lot of attention [1], [2], [3], [4]. In these networks, a user is often associated with location information (e.g., positions of her hometown and check-ins). These networks are collectively known as *spatial graphs*. Figure 1 depicts a spatial graph with nine users in three cities `Berlin`, `Paris`, `London`, and each user has a specific location. The solid lines represent their social relationship, and the dashed lines denote their hometown locations.

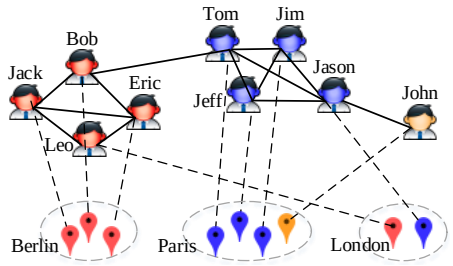


Fig. 1. A geo-social network.

In this paper, we study the problem of performing online community search (CS) on spatial graphs. Given a spatial graph G and a vertex $q \in G$, our goal is to find a subgraph of G, called a *spatial-aware community* (or SAC). Essentially, a community is a social unit of any size that shares common values, or that is situated in a close area [5], [6], [7] An SAC is such a community with high *structure cohesiveness* and *spatial cohesiveness*. The structure cohesiveness mainly measures the social connections within the community, while the spatial cohesiveness focuses on the closeness among their geo-locations. Figure 1 illustrates an SAC with three users {`Tom`, `Jeff`, `Jim`}, in which each user is linked with each other and all of them are in `Paris`.

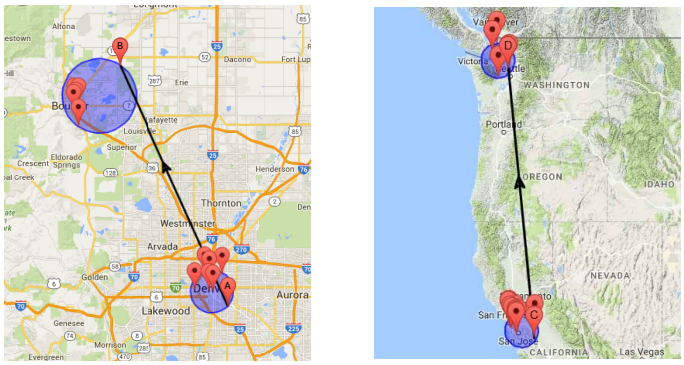
- *Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, and J. Hu are with the Department of Computer Science, The University of Hong Kong, Hong Kong. E-mail: {yxfang, zwang, ckcheng, xdli, sqluo, jhu}@cs.hku.hk*
- *X. Chen (*Corresponding Author) is with College of Computer Science and Software, Shenzhen University, Shenzhen, China. E-mail: xjchen@szu.edu.cn*

TABLE 1
Works on community retrieval (CR).

Graph Type	Community Detection (CD)	Community Search (CS)
Non-spatial	[8], [9]	[10], [11], [12], [13], [14], [15]
Spatial	[16], [17], [18]	**SAC search**

Prior works. The community retrieval (CR) methods can generally be classified into *community detection* (CD) and *community search* (CS), as shown in Table 1. Earlier CD methods [8], [9] mainly focus on link analysis without considering spatial features. Some recent studies [19] have shown that, in networks where vertices occupy positions in an Euclidian space, spatial constraints may have a strong effect on their relationship patterns, so some works [16], [17], [18] have considered the spatial features for CD. All these CD methods often detect all the communities from an entire graph using some predefined global criteria (e.g., modularity [20]), so their focus is beyond personalized CS. Also, their efficiency is inadequate for fast and online CR since they require to enumerate all the communities. To address these limitations, some works [10], [11], [12], [14], [15] focus on *online* CS, a query-dependent variant of CD, and they are able to find communities for a specific vertex. However, almost all these CS works focus on link analysis and do not consider the spatial features. In Figure 1, for example, previous CS methods [10], [11] tend to put `Jason` and `Tom`, `Jeff`, `Jim` into the same community, although `Jason` is located in another city `London`. This community may not be very useful for some location-based services (e.g., setting up events). To alleviate this issue, in this paper we study SAC search which finds communities for a particular query vertex in an "online" manner. Our later experimental results on real datasets show that, the communities found by our methods are often in a much smaller areas than that of previous CS methods, i.e., the radii of the spatial circles covering communities found by [10] and [11] are 50 and 20 times larger than those of SAC search.

SAC search. We now discuss how to measure the structure cohesiveness and spatial cohesiveness of an SAC. We adopt



(a) $user_1$'s SACs (b) $user_2$'s SACs

Fig. 2. SACs in Brightkite dataset.

the commonly used metric *minimum degree* [10], [11], [13] to measure the structure cohesiveness. Note that in our method, the minimum degree metric can be easily replaced by other metrics like k-truss [14] and k-clique [12]. To measure the spatial cohesiveness, we consider the *spatial circle*, which contains all the community members. In particular, given a query vertex $q \in G$, our goal is to find an SAC containing q in the smallest *minimum covering circle* (or MCC) and all the vertices of the SAC satisfy the minimum degree metric. The main features of SAC search are summarized as follows.

• **Adaptability to location changes.** In geo-social networks (e.g., Brightkite and Foursquare), a user's location often changes frequently, due to its nature of mobility. As a result, users' spatially close communities change frequently as well. Let us consider two real examples in Brightkite, which once was a popular location-based social networking website. Figure 2(a) shows a user's two SACs in two consecutive days, when she moves from place "A" to place "B" in US, in which each SAC is located in an MCC denoted by a circle. Note that all the members are different except the user itself. Figure 2(b) shows another user's two SACs in three days, when she moves from place "C" to place "D". These real examples clearly show that a user's communities could evolve over time. In our later experiments, we find that for two SACs with time gap of six hours or more, the average Jaccard similarity of these two community member sets decreases by 25%.

Moreover, the link relationship also evolves over time. So the existing CD methods may easily lose the freshness and effectiveness after a short period of time. On the contrary, our SAC search can adapt to such dynamic easily, as it can answer queries in an "online" manner. Also, our methods do not rely on any offline computation, such as graph clustering or index structures.

• **Personalization.** SAC search allows a query user to find a community that exhibits both high structure cohesiveness and spatial cohesiveness. The parameter k, the minimum degree, allows the user to control the strength of link intensiveness. For example, SAC search can answer queries such as who are my nearby friends so that we can form a particular club? In contrast, existing CD methods [16], [17], [18] often use some global criteria (e.g., modularity), and consider the *static* community detection problem, where the graph is partitioned a-priori with no reference to the particular query vertices.

• **Online search.** Similar to other online CS methods, our method is able to find an SAC from a large spatial graph quickly once a query request arrives. However, existing CD methods for spatial graphs, are generally slower, as they are often designed for generating all the communities for an entire graph.

Applications. We now discuss the applications of SAC search.

• **Event recommendation.** Emerging geo-social applications such as *Meetup* and *Meetin*[1] allow social network users to meet physically for various interesting purposes (e.g., party, dinner, and dating). For example, *Meetup* tracks its users' mobile phone locations, and suggests interesting location-based events to them [2]. Suppose that *Meetup* wishes to recommend an event to a user u. Then we can first find u's SAC, whose members are physically close to u. Events proposed by u's SAC member v can then be introduced to u, so that u can meet v if she is interested in v's activity. Since u's location changes constantly, u's recommendation needs to be updated accordingly. Also, these applications often have to handle requests from a large number of online users efficiently. Our high-performance SAC search algorithms can therefore benefit these applications.

• **Social marketing.** As studied in [21], people with close social relationships tend to purchase in places that are also physically close. To boost sales figures, advertisement messages can be sent to the SACs of users who bought similar products before. For instance, if u has bought an item, the system can advertise this item to u's SAC members.

• **Geo-social data analysis**.A common data analysis task is to study features about geographical regions. As discussed in [22], these features are often related to the people located there. For example, Silicon Valley can be characterized by "information technology" because many residents there are interested in this topic. Hence, by analyzing members of an SAC, it is possible to better understand the characteristics of a geographical area. As also discussed in [23] and Figure 2, SAC search can be used to monitor and analyze the movement of communities. We can thus track the evolution and composition of u's SAC as she moves.

Challenges and contributions. The SAC search problem is very challenging, because the center and radius of the smallest MCC containing q are unknown. A basic exact approach takes $O(m \times n^3)$ time to answer a query, where n and m denote the numbers of vertices and edges in G. This is very costly, and is impractical for large spatial graphs with millions of vertices. So we turn to develop efficient approximation algorithms, which are able to find an SAC in an MCC of similar size with the smallest MCC. We first develop a basic approximation algorithm `AppInc`, which achieves an approximation of 2. Here, the approximation ratio is defined as the ratio of the radius of MCC returned over that of the optimal solution. Inspired by `AppInc`, we develop another approximation algorithm `AppFast`, which is faster and also has a more flexible approximation ratio, i.e., $2+\epsilon_F$, where ϵ_F is an arbitrary small non-negative value. However, `AppInc` and `AppFast` cannot achieve even better accuracy with an approximation ratio less than 2. To tackle this issue, we further propose another approximation algorithm `AppAcc` with an approximation ratio of $1 + \epsilon_A$, where $0< \epsilon_A <1$. Overall, these approximation algorithms theoretically guarantee that, the radius of the MCC containing the SAC found has an arbitrary expected approximation ratio. Finally, inspired by the design of approximation algorithms, we develop an advanced exact algorithm `Exact+`, and our later experiments show that it is four orders of magnitude faster than the basic exact algorithm.

CSAC Search. In the geo-social networks above, the locations of vertices often change over time. For instance, a user may go to

1. Meetup:https://www.meetup.com/ Meetin:https://www.meetin.org/

different restaurants in a day and perform "check-in" actions. As shown in Figure 2 and our experiments in Section 6.2.2, a user's SACs can change after a short period of time, due to the fact that every member of the SACs can move frequently. It is thus desirable to efficiently monitor the change of a user's SACs. In this paper, we study the problem of *continuous SAC* (or CSAC) search, which aims to find the SACs of a query vertex q that can move as time goes by. The CSAC search can be applied to many social applications that need to know the latest communities of the users in order to provide the best recommendation. For example, in Meetup, when a user moves to a new place, it would be interesting to know her current location, and then provides event recommendation and product promotion services for her and members in her SAC in an instant manner.

A simple method to address the CSAC search problem is to invoke an SAC search algorithm upon every location change, which however is very inefficient, because users' locations may change frequently, calling for more efficient algorithms. We observe that, although a user's SACs may change frequently, they often share some vertices, and are spatially close or overlapped. Based on this intuition, we design three fast algorithms by extending SAC search algorithms, which are of theoretical accuracy guarantees.

We have implemented algorithms and performed extensive experiments on four real datasets and two synthetic datasets. We develop several metrics to measure the quality of a community with respect to the users' social relationship and positions. Experimental results show that our solutions yield better communities than those produced by existing CS and CD algorithms. We have also studied the performance of different SAC search solutions. For moderate-size graphs, `Exact+` is the best choice, as it achieves the highest quality with reasonable efficiency; for large graphs with millions of vertices or edges, `AppFast` and `AppAcc` are better options as they are much faster than `Exact+`. As shown in our experiments, our CSAC search algorithms are adaptive to frequent location changes, and faster than the baseline algorithms.

We review the related work in Section 2. We formulate the problem of SAC search in Section 3. Section 4 presents the algorithms for SAC search. In Section 5, we introduce the problem of CSAC search and develop three algorithms. We show the experimental results in Section 6 and conclude in Section 7.

2 Related Work

Community detection (CD). Classical CD solutions [8], [9], [24] often employ link-based analysis to obtain these communities. Some recent studies have used other advanced techniques, such as game-theoretic approach [25] and Laplacian centrality [26], to identify dense communities in large-scale complex networks. However, they do not consider the location information. Some recent works [16], [17], [18], [27] focus on identifying communities from spatially constrained graphs, whose vertices are associated with spatial coordinates [19]. For example, a geo-community [27] is like a community which is a graph of intensely connected vertices being loosely connected with others, but it is more compact in space. Guo et al. [16] proposed the average linkage measure for clustering objects in spatially constrained graphs. Expert et al. [17] uncovered communities from spatial graphs based on modularity maximization. In [18], Chen et al. used modularity maximization for CD on spatially constrained networks. We will compare it with our methods in experiments. The differences of CD algorithms and our SAC search are threefold. First, CD algorithms are generally costly, as they often detect all the communities from an entire network. Second, it is not clear how they can be adapted for online CR. Third, as pointed out by [20], modularity based methods [17], [18] often fail to resolve small-size communities, even when they are well defined.

Community search (CS). In recent years, there is another related but different problem of CD, called *community search* (CS), which aims to obtain communities in an "online" manner, based on a query request. For example, given a vertex q, several existing works [10], [11], [12], [13], [14], [28] have proposed effective algorithms to obtain the most likely community that contains q. The *minimum degree* metric is often used to measure the structure cohesiveness of a community [10], [11]. In [10], Sozio et al. proposed the first algorithm `Global` to find the k-$\widehat{core}$ containing q. In [11], Cui et al. proposed a more efficient algorithm `Local`, which uses local expansion techniques to boost the query performance. We will compare these two solutions in our experiments. In addition, some recent works [13], [15] also use the minimum degree metric to search communities from attributed graphs. Other well known structure cohesiveness, including k-clique [12], k-truss [14] and connectivity [29], have also been considered for online community search. But these works assume non-spatial graphs, and overlook the locations of vertices. Thus, it is desirable to investigate CS over spatial graphs.

In addition, graphs are often evolving in practice, which implies that the communities returned by CS solutions can become stale after a short period of time. Thus, more effective and efficient CS solutions on dynamic graphs are needed. However, designing such solutions is challenging due to the following reasons. First, network structures change dynamically. For example, users can join Facebook and make new friends frequently. Second, the attributes of vertices also update frequently. For instance, the locations of Facebook users may change quickly due to their nature of mobility. As a result, to search and track communities in such dynamic scenarios, we have to consider the dynamics from both network structures and attributes. Moreover, as reviewed below, although there are some existing studies on continuous queries, none of them has studied how to search communities continuously from dynamic spatial graphs. To fill this gap, in this paper we study continuous SAC search on spatial graphs.

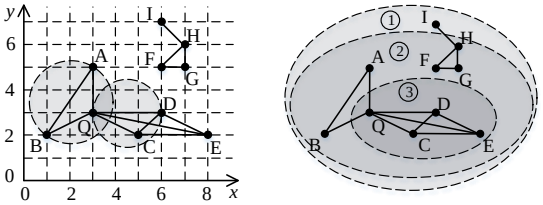
Continuous query (CQ). In recent years, the topic of CQ has received plenty of research interest, and many studies [30], [31], [32] focus on queries of spatial databases. In [30], the continuous queries over data streams are studied. Iwerks et al. [31] examine the continuous k-NN query. Wu et al. [32] study the continuous top-k spatial keyword query for a moving object. However, these studies focus on the spatial features [33], [34] of objects, and do not take the social network and communities into consideration. In recent studies [35], [36], the problem of maintaining k-core on dynamic graphs, where edges can be inserted and deleted dynamically, has been extensively studied. However, they do not consider the spatial features of objects. In summary, none of the existing works solves the problem of continuous SAC search.

3 Problem Definition

Data model. We consider a geo-social network graph $G(V, E)$, which is an undirected graph with vertex set V and edge set E, where vertices represent entities and edges denote their relationships. For each vertex $v \in V$, it has a tuple (id, loc), where id is

its ID and loc=(x, y) is its spatial positions along x- and y-axis in a two-dimensional space. Let n and m be the corresponding sizes of V and E. We illustrate the data model using Example 1. Table 3 shows the notations used in this paper.

Example 1. Figure 3(a) depicts a spatial graph, where each solid black line between two vertices represents an edge.



(a) spatial graph (b) k-core decomposition

Fig. 3. An example of geo-social network.

Spatial-aware community (SAC). Conceptually, an SAC is a subgraph, G', of the graph G satisfying: (1) *Connectivity*: G' is connected; (2) *Structure cohesiveness*: all the vertices in G' are linked intensively; and (3) *Spatial cohesiveness*: all the vertices in G' are spatially close to each other. Note that all the existing CS studies assume that a community should satisfy the connectivity and structure cohesiveness [10], [11], [12], [13], [14], [28].

Structure cohesiveness. A well-accepted notion of structure cohesiveness is the *minimum degree* of all the vertices that appear in the community is at least k [10], [11], [13], [37], [38]. This is used in k-core and our SAC search. Let us discuss the k-core first.

Definition 1 (k-core [37], [38]). Given an integer k ($k \geq 0$), the k-core of G, denoted by H_k, is the largest subgraph of G, such that $\forall v \in H_k$, $deg_{H_k}(v) \geq k$.

We say that H_k has an order of k. The *core number* of a vertex $v \in V$ is then defined as the highest order of the k-$core$ that contains v. A k-core has some properties [38]: (1) H_k contains at least $k+1$ vertices; (2) H_k may not be a connected graph; (3) k-cores are nested, i.e., $H_{k+1} \subseteq H_k$; and (4) Computing the core numbers of all the vertices in a graph, also known as k-core decomposition, can be completed using a linear algorithm [38].

As a k-core may not be a connected subgraph, we denote its connected components by k-$\widehat{core}$s, which are usually the "communities" returned by k-$\widehat{core}$ search algorithms [10], [11]. In Example 1, each k-core is covered by an ellipse as shown in Figure 3(b). Note that 2-core has two 2-$\widehat{core}$s with vertex sets $\{Q, A, B, C, D, E\}$ and $\{F, G, H\}$ respectively.

Remarks. Although we use the minimum degree in this paper, our proposed solutions can be easily adapted to other commonly-used structure cohesiveness metrics (e.g., k-truss [14] and k-clique [12]). For example, to use k-truss for any of our proposed algorithm, we can simply replace k-core by k-truss in the steps of verifying the structure cohesiveness of an SAC.

Spatial cohesiveness. In this paper, to ensure high spatial cohesiveness, we require all the vertices of an SAC in a minimum covering circle (MCC) with the smallest radius. In the literature [39], [40], [41], [42], the notion of MCC has been widely adopted to achieve high spatial compactness for a set of spatial objects. The MCC and SAC search are defined as follows.

Definition 2 (MCC). Given a set of vertices S, the MCC of S is the spatial circle, which contains all the vertices in S with the smallest radius.

TABLE 2
Notations and meanings

Notation	Meaning
$G(V, E)$	a graph with vertex set V and edge set E
n, m	the sizes of sets V and E respectively
$G[S]$	a subgraph of G induced by vertex set S
$nb(v)$	the neighbor set of vertex v in G
$deg_G(v)$	the degree of vertex v in G
$G' \subseteq G$	G' is a subgraph of G
$O(o, r)$	a spatial circle with center o and radius r
$\|u, v\|$	the Euclidean distance from vertex u to vertex v
Ψ	Results of `Exact` and `Exact+` (optimal solution)
Ψ'	Results of `CExact+` (optimal solution)
Φ, Λ, Γ	Results of `AppInc`, `AppFast`, `AppAcc` respectively
Λ', Γ'	Results of `CAppFast` and `CAppAcc`
r_{opt}	The radius of the MCC covering Ψ
γ	The radius of MCC covering Φ
δ	The radius of MCC (centered at q) covering Φ
$\mathcal{T}$	The quadtree used in `AppAcc` algorithm

Problem 1 (SAC search). Given a graph G, a positive integer k and a vertex $q \in V$, return a subgraph $G_q \subseteq G$, and the following properties hold:

1. **Connectivity**. G_q is connected and contains q;
2. **Structure cohesiveness**. $\forall v \in G_q$, $deg_{G_q}(v) \geq k$;
3. **Spatial cohesiveness**. The MCC of vertices in G_q satisfying Properties 1 and 2 has the minimum radius.

We call a subgraph satisfying properties 1 and 2 a *feasible* solution, and the subgraph satisfying all the three properties the *optimal* solution (denoted by Ψ). We denote the radius of the MCC containing Ψ by r_{opt}. Essentially, SAC search finds the SAC in an MCC with the smallest radius among all the feasible solutions. In Example 1, let C_1=$\{Q, C, D\}$ and C_2=$\{Q, A, B\}$. The two circles in Figure 3(a) denote the MCCs of C_1 and C_2 respectively. Let q=Q and k=2. The optimal solution of this query is $G[C_1]$, and r_{opt}=1.5. Note that $G[C_2]$ and $G[C_1 \cup C_2]$ are feasible solutions. Additionally, although we only consider single query vertex in this paper, our proposed solutions can be easily extended to support multiple query vertices.

We also consider the θ-SAC search, which returns a community satisfying: properties 1 and 2 of SAC search, and all the vertices are in a spatial circle $O(q, \theta)$, where θ is an input parameter. This θ-SAC search is essentially a variant of `Global` [10] by introducing a parameter θ. Consider the graph in Example 1 with q=Q, k=2 and θ=3.1. θ-SAC search will return $G[C_1 \cup C_2]$ as the community, as all of its vertices are in $O(Q, 3.1)$.

The θ-SAC query can be used when a user has some background knowledge (e.g., size of the region containing the SAC, and density of users in the region concerned). However, it can be difficult for a user of an application, such as Meetup, to specify an appropriate value of θ. As will be discussed in our experiments, the effectiveness of θ-SAC search is sensitive to θ. If θ is too small, no community can be found; if θ is too large, then the community is not spatially compact. A casual application user may then have to repeat the query with different θ values, before getting a satisfactory result. For the SAC search, the user does not need to specify θ; instead, SAC search automatically suggests a community with tight structural and spatial cohesiveness. Thus, SAC search is more convenient to use than θ-SAC. In the above example, if θ<2.2, no community is found; if θ>5.1, $G[C_3]$ will be returned, where C_3=$\{Q, A, B, C, D, E\}$. In fact, there are more spatially compact

TABLE 3
Overview of algorithms for SAC search.

Algorithm	Approximation ratio	Time complexity
Exact	1	$O(m \times n^3)$
AppInc	2	$O(mn)$
AppFast	2+ϵ_F ($\epsilon_F \geq 0$)	If $\epsilon_F > 0$, $O(m \cdot \min\{n, \log \frac{1}{\epsilon_F}\})$ If $\epsilon_F = 0$, $O(mn)$
AppAcc	1+ϵ_A ($0 < \epsilon_A < 1$)	$O(\frac{m}{\epsilon_A^2} \times \min\{n, \log \frac{1}{\epsilon_A}\})$
Exact+	1	$O(\frac{m}{\epsilon_A^2} \cdot \min\{n, \log \frac{1}{\epsilon_A}\} + m|F_1|^3)$

SACs (e.g., $G[C_1]$, $G[C_2]$ and $G[C_1 \cup C_2]$), among which the most compact one ($G[C_1]$) is returned by the SAC search.

4 SAC Search Algorithms

Most of our proposed algorithms follow a *two-step framework*: (1) find a community S of vertices, based on some CS algorithm e.g., Global [10], and (2) find a subset of S that satisfies both structure and spatial cohesiveness. Step (2) is computationally challenging; a simple way is to enumerate all the possible subsets of S, and then choose the one that satisfies the two criteria of SAC. In Example 1, when q=Q and k=2, S=$\{Q, A, B, C, D, E\}$; an SAC is then chosen from the 2^6–1=63 subsets of S. This requires the examination of an exponential number of possible subsets of S in Step (2). In our experiments, the typical size of S ranges from $1K$ to $100K$. As a result, the performance of SAC search can be seriously affected. Hence, we study polynomial-time SAC search algorithms for Step (2). Later we will also present the AppInc solution, which does not use Step (1).

We first present a basic exact algorithm Exact, which takes $O(m \times n^3)$ to answer a single query. This is very time-consuming for large graphs, so we turn to design more efficient approximation algorithms. Here, the approximation ratio is defined as the ratio of the radius of MCC returned over that of the optimal solution. Inspired by the approximation algorithms, we also design an advanced exact algorithm Exact+. Their approximation ratios and time complexities are summarized in Table 3, where ϵ_F and ϵ_A are parameters specified by the query user. The value $|F_1|$ is the number of "fixed vertices", and $|F_1|$ is often much smaller than n. We will explain this parameter in Section 4.1.

AppInc is a 2-approximation algorithm, and it is much faster than Exact. Inspired by AppInc, we design another (2+ϵ_F)-approximation algorithm AppFast ($\epsilon_F \geq 0$), which is faster than AppInc. The limitation of AppInc and AppFast is that their approximation ratios are at least 2. To achieve even lower approximation ratio, we further design another algorithm AppAcc, whose approximation ratio is (1+ϵ_A), where $0 < \epsilon_A < 1$ is a value specified by the query user. It is slightly slower than AppFast, as it spends more effort on finding more accurate solutions. Overall, these approximation algorithms guarantee that the radius of the MCC of the community has an arbitrary approximation ratio.

All algorithms except AppInc follow the two-step framework. Note that Step (1) of the two-step framework is not necessary for AppInc, since it works in an incremental manner. In addition, we can observe that, there is a trade-off between the quality of results and efficiency, i.e., algorithms with lower approximation ratios tend to have higher complexities. Our later experiments show that, for moderate-size graphs, Exact+ achieves not only the highest quality results, but also reasonable efficiency. While for large graphs with millions of vertices, AppFast and AppAcc should be better choices as they are much faster than Exact+.

4.1 The Basic Exact Algorithm

As mentioned before, a k-core contains at least $k+1$ vertices. When k=1, we can simply return the subgraph, induced by q and its nearest neighbor, as the result. So we focus on the case $k \geq 2$ in this paper. We begin with a useful lemma about MCC, described in [40], which inspires the design of our algorithms.

Lemma 1. [40] Given a set S ($|S| \geq 2$) of vertices, its MCC can be determined by at most three vertices in S which lie on the boundary of the circle. If it is determined by only two vertices, then the line segment connecting those two vertices must be a diameter of the circle. If it is determined by three vertices, then the triangle consisting of those three vertices is not obtuse.

By Lemma 1, there are at least two or three vertices lying on the boundary of the MCC of the target SAC. We call vertices lying on the boundary of an MCC *fixed* vertices. So a straightforward method of SAC search can follow the two-step framework directly. It first finds the k-$\widehat{core}$ containing q, which is the same as Global does, and then returns the subgraph achieving both the structure and spatial cohesiveness by enumerating all the combinations of three vertices in the k-$\widehat{core}$. We denote this method by Exact. The pseudocodes are presented in the full version of this paper [43]. In addition, we show another two useful lemmas. Note all the proofs of lemmas proposed in this paper are discussed in [43].

Lemma 2. [42] The maximum distance between any pair of vertices, u and v in Ψ, is in the range $[\sqrt{3}r_{opt}, 2r_{opt}]$.

Lemma 3. The overall time complexity of Exact is $O(m \times n^3)$.

4.2 A 2-Approximation Algorithm

The major limitation of Exact is its high computational cost, which makes it impractical for large spatial graphs with millions of vertices. To alleviate this issue, we now develop more efficient approximation algorithms. We first present AppInc, which has an approximation ratio of 2. Our key observation is that, the optimal solution Ψ is usually very close to q. So we consider the smallest circle, denoted by $O(q, \delta)$, which is centered at q and contains a feasible solution, denoted by Φ. Let the radius of the MCC covering Φ be γ ($\gamma \leq \delta$). Note that, γ can be obtained by computing the MCC containing Φ by a linear algorithm [41]. Then, we have the following two interesting lemmas:

Lemma 4. $\frac{1}{2}\delta \leq r_{opt} \leq \gamma$.

Lemma 5. The radius of the MCC covering the feasible solution Φ has an approximation ratio of 2.

AppInc finds Φ in an incremental manner. Specifically, it considers vertices close to q one by one incrementally, and checks whether there exists a feasible solution when a new vertex is considered. It stops once a feasible solution has been found. The codes are detailed in [43]. We illustrate it by Example 2.

Example 2. In Example 1, let q=Q and k=2. AppInc first adds A to S and no SAC can be found. Then, it adds B to S, finds Φ with members set $\{Q, A, B\}$. So γ=1.803 and δ=$|Q, B|$=2.24. The actual approximation ratio is 1.803/1.5=1.202.

Corollary 1. If q is the center of the MCC covering Ψ, AppInc finds the optimal solution, i.e., Φ equals to Ψ.

Corollary 2. The optimal solution Ψ is in $O(q, 2\gamma)$.

Lemma 6. The total time cost of AppInc is $O(mn)$.

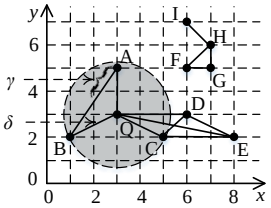


Fig. 4. Illustrating AppInc.

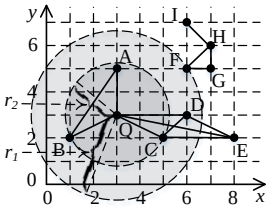


Fig. 5. Illustrating AppFast.

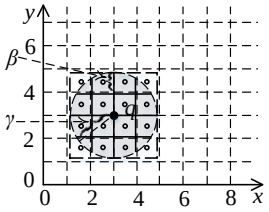


Fig. 6. Splitting $O(q, \gamma)$

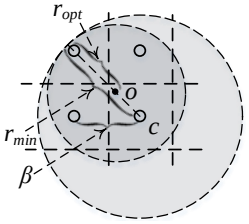


Fig. 7. Illustrating r_{min}.

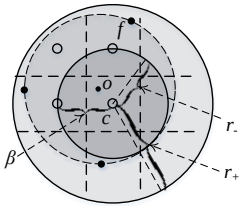


Fig. 8. Exact+: r_-, r_+.

4.3 A (2+ϵ_F)-Approximation Algorithm

Although AppInc is much faster than Exact, it is still inefficient for large graphs, since its time complexity is quadratic. In this section, we propose another fast approximation algorithm, called AppFast, which has a more flexible approximation ratio, i.e., $2 + \epsilon_F$, where ϵ_F is an arbitrary non-negative value.

Instead of finding the circle $O(q, \delta)$ in an incremental manner, AppFast approximates the radius δ by performing binary search. This is based on the observation that, the lower and upper bounds of δ, denoted by l and u, are stated by Eq (1):

$$l = \max_{v \in KNN(q)} |q, v|, \quad u = \max_{v \in X} |q, v|, \tag{1}$$

where X is the list of vertices of the k-$\widehat{core}$ containing q, and $KNN(q)$ contains the k nearest vertices in $X \cap nb(q)$ to q. Hence, we can approximate the radius of the circle $O(q, \delta)$ by performing binary search within $[l, u]$ until the gap $|u - l|$ is less than α, a predefined threshold. More details of AppFast can be found in [43]. We illustrate AppFast by Example 3.

Example 3. In Figure 5 (q=Q, k=2, ϵ_F=0.1), AppFast first initializes l=2.24, u=5.10, and tries to find a feasible solution from $O(Q, r_1)$ and $O(Q, r_2)$, where r_1=3.67 and r_2=2.24. It stops after searching $O(Q, r_2)$, as $r_2 - l$=0. Λ equals to Φ.

Lemma 7. In AppFast, the radius of the MCC covering Λ has an approximation ratio of $(2 + \epsilon_F)$, if α is set as $\frac{r \times \epsilon_F}{2+\epsilon_F}$.

Corollary 3. The optimal solution Ψ is in $O(q, 2r_\Lambda)$, where r_Λ is the radius of the MCC containing Λ in AppFast.

Lemma 8. The total time cost of AppFast is $O(\min\{mn, m \log \frac{1}{\epsilon_F}\})$ if ϵ_F>0, or $O(mn)$ if ϵ_F=0.

4.4 A (1+ϵ_A)-Approximation Algorithm

AppInc and AppFast guarantee that, the radius of the MCC of the returned SAC has an approximation ratio of 2 or more, but cannot achieve even better accuracy. To tackle this issue, we propose another algorithm, called AppAcc, which has an approximation ratio of $(1+\epsilon_A)$, where $0<\epsilon_A<1$. The main idea is based on a key observation from Lemma 4, stated by Corollary 4:

Corollary 4. The center point, o, of the MCC $O(o, r_{opt})$ covering Ψ is in the circle $O(q, \gamma)$.

Although point o is in $O(q, \gamma)$, it is still not easy to locate it exactly, since the number of its possible positions to be explored can be infinite. Instead of locating it exactly, we try to find an approximated "center", which is very close to o. In specific, we split the square containing the circle $O(q, \gamma)$ into equal-sized cells, and the size of each cell is $\beta \times \beta$ (we will explain how to set a proper value of β later). We call the center point of each cell an ***anchor point***. By Corollary 4, we can conclude that o must be in one specific cell. Then we can approximate o using the anchor point of this cell, denoted by c, which is also its nearest anchor point, since their distance $|o, c|$ is at most $\frac{\sqrt{2}}{2}\beta$.

Example 4. In Figure 6, each small circle point in $O(q, \gamma)$ represents an anchor point. In Figure 7, c is the nearest anchor point of o. It is easy to observe that $|o, c| \leq \frac{\sqrt{2}}{2}\beta$.

We consider the circle $O(c, r_{min})$, where r_{min} is the minimum radius such that it contains a feasible solution, which is denoted by Γ. The value of r_{min} is bounded by Lemma 9.

Lemma 9. $r_{min} \leq r_{opt} + \frac{\sqrt{2}}{2}\beta$.

By Lemma 9, we have $\frac{r_{min}}{r_{opt}} \leq 1 + \frac{\sqrt{2}\beta}{2r_{opt}} \leq 1 + \frac{\sqrt{2}\beta}{\delta}$. Thus, we can approximate Ψ using Γ, and the approximation ratio is $(1 + \epsilon_A)$, if we let $\frac{\sqrt{2}\beta}{\delta} \leq \epsilon_A$ $(0 < \epsilon_A < 1)$.

To find $O(c, r_{min})$, the basic method is that, for each anchor point p, we use AppFast to find the circle, which is centered at p and contains a feasible solution, and then return the minimum circle. However, the number of anchor points is $\left(\frac{2\gamma}{\beta}\right)^2$, and each takes $O(mn)$ to find a feasible solution in the worst case. So this is very time-consuming, if β (ϵ_A) is very small. To further improve the efficiency, we develop some optimization techniques.

Specifically, we assume that all the anchor points are organized into a region quadtree [44], where the root node[2] is a square, centered at q with width 2γ. By decomposing this square into four equal-sized quadrants, we obtain its four child nodes. The child nodes of them are built in the same manner recursively, until the width of the leaf node is in $(\beta/2, \beta]$. Note that the center of each leaf node corresponds to an anchor point.

To find $O(c, r_{min})$, we traverse the quadtree level by level in a top-down manner. Let r_{cur}, initialized as γ, record the smallest radius of an MCC containing a feasible solution. For each node, we first obtain the center p of its square, and then use the binary search technique introduced in AppFast to approximate the smallest radius r_p, such that $O(p, r_p)$ contains a feasible solution. During the traversal, for each node, to check whether it can be pruned, we propose three effective pruning criteria:

• *Pruning1:* Consider a node (with center p), which intersects at the boundary of $O(q, r_{cur})$. Then we have $|p, q| \leq r_{cur} + \frac{\sqrt{2}}{2}\beta$. Thus, if the distance from the center of this node to q is larger than $r_{cur} + \frac{\sqrt{2}}{2}\beta$, its sub-trees can be pruned.

• *Pruning2:* If $O(p, r)$ does not contain a feasible solution and $r > r_{cur} + \frac{\sqrt{2}}{2}\beta$, then its sub-trees can be pruned.

• *Pruning3:* Let Γ_i be the SAC found by the binary search on the i-th anchor point, and M_i be the MCC covering Γ_i. If there are more than one anchor points whose corresponding SACs are the same, say Γ_a, then we can only keep one anchor point b whose distance to the center of M_a is the minimum.

2. To avoid ambiguity, we use word "node" for tree nodes.

In addition, we propose a lemma, which may stop the process of searching the quadtree earlier, as follows.

Lemma 10. In AppAcc, if $r_{cur} \leq \frac{1}{2}\delta(1+\epsilon_A)$, then we can stop searching the quadtree and return the SAC with radius r_{cur}.

Based on the analysis above, we design AppAcc. It first performs AppFast, and obtains the k-$\widehat{core}$ in $O(q, 2\gamma)$. Then, it explores the quadtree level by level with pruning in a top-down manner and tries to find feasible solutions using binary search as that of AppFast, until the cell size is at most $\beta \times \beta$. Finally, it returns the SAC (denoted by Γ) in the smallest MCC centered at a specific anchor point. More details are described in [45].

Lemma 11. If we set $\alpha' \leq \frac{1}{4}\delta\epsilon_A$ and $\beta = \frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$ in AppAcc, where α' is the threshold of the gap between the upper and lower bounds of the binary search, and $0 < \epsilon_A < 1$, the radius of the MCC covering Γ has an approximation ratio of $(1+\epsilon_A)$.

Lemma 12. AppAcc takes $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\})$ time.

4.5 The Advanced Exact Algorithm

The design of previous algorithms provide us many useful insights for developing more advanced exact algorithms. For example, Corollary 2 states that, the optimal solution Ψ is in $O(q, 2\gamma)$. This implies that, we can first run AppInc, then only enumerate the vertex triples for vertices in $O(q, 2\gamma)$, which is a subset of V. Similarly, we can find Ψ by Corollary 3 based on AppFast. Although these methods could be faster than Exact, they are still far from perfect, because the number of potential fixed vertices in $O(q, 2\gamma)$ may still be very large. In this section, we propose a very efficient exact algorithm based on AppAcc, called Exact+, which largely reduces the number of potential fixed vertices, and thus improves the efficiency significantly.

Recall that, AppAcc approximates the center, o, of the MCC covering Ψ by its nearest anchor point c, and $|o, c| \leq \frac{\sqrt{2}}{2}\beta$. Also, r_{opt} is well approximated, i.e., $\frac{r_\Gamma}{r_{opt}} \leq 1 + \epsilon_A$, which implies that,

$$\frac{r_\Gamma}{1+\epsilon_A} \leq r_{opt} \leq r_\Gamma, \tag{2}$$

where $0 < \epsilon_A < 1$. So the value of r_{opt} is in a small interval, especially if ϵ_A is small.

Besides, for any fixed vertex, f, of the MCC of Ψ, its distance to o (i.e., $|f, o|$) is exactly r_{opt}. By triangle inequality, we have

$$|f, c| \leq |f, o| + |o, c| \leq r_\Gamma + \frac{\sqrt{2}}{2}\beta, \tag{3}$$

$$|f, c| \geq |f, o| - |o, c| \geq \frac{r_\Gamma}{1+\epsilon_A} - \frac{\sqrt{2}}{2}\beta. \tag{4}$$

Let us denote the rightmost items of above two inequations by r_+ and r_- respectively. Then, we conclude that, for any fixed vertex f, its distance to c is in the range $[r_-, r_+]$. If ϵ_A is very small, the gap between r_+ and r_-, i.e., $r_+ - r_- = r_\Gamma(1 - \frac{1}{1+\epsilon_A}) + \sqrt{2}\beta$, is also very small, which implies that the locations of the fixed vertices are in a very narrow annular region. Hence, a large number of vertices out of this annular region, which are not fixed vertices, can be pruned safely. We illustrate this in Figure 8, in which the annular region is the area in $O(c, r_+)$, but not in $O(c, r_-)$. Based on the analysis above, we design Exact+, which first runs AppAcc, then prunes fixed vertices using annular regions, and finally computes the target SAC by enumerating all the combinations of three fixed vertices. The detailed pseudocodes of Exact+ are presented in [43].

Lemma 13. The overall time cost of Exact+ is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\} + m|F_1|^3)$, where ϵ_A is the parameter of AppAcc and $|F_1|$ ($|F_1| \ll n$) is the number of fixed vertices left after pruning by annular regions.

5 CSAC SEARCH

In this section, we first formally define problem of CSAC search and then introduce the proposed algorithms for CSAC search.

5.1 Problem Definition

Data model. We consider a *dynamic* spatial graph $G(V, E)$, which is an undirected graph with vertex set V and edge set E, where vertices represent entities and edges denote their relationships. Besides, each vertex v has a triple (id, loc, t), where id is v's ID and loc is v's location at the timestamp t, and its location loc changes as the time t goes on.

Given a query vertex, the goal of CSAC search is to dynamically track her SACs in a future time interval. We formulate the CSAC search problem as follows and illustrate it by Example 5.

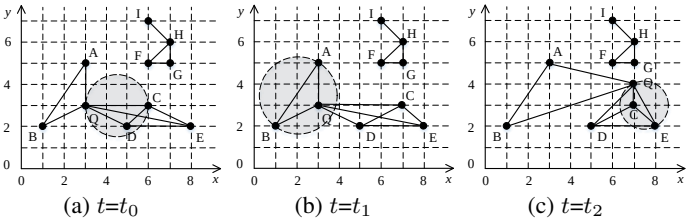
Problem 2 (CSAC search). Given a dynamic spatial graph G, a positive integer k, a future timestamp t_e, and a vertex $q \in V$, find a list $G_q(t)$ of SACs such that

$$G_q(t) = \begin{cases} G_{q,1}, & t \in [t_s, t_1) \\ G_{q,2}, & t \in [t_1, t_2) \\ \cdots, & t \in [\cdots, \cdots) \\ G_{q,l}, & t \in [t_{l-1}, t_e] \end{cases}, \tag{5}$$

where t_s is the timestamp of starting the query, t_i ($1 \leq i \leq l-1$) is a future timestamp, l is the total number of SACs, $G_{q,i}$ is the i-th SAC, and $G_{q,i}$ is different with $G_{q,i+1}$.

Example 5. Figure 9 shows a dynamic spatial graph during the time period $[t_0, t_2]$, where t_0 is the current timestamp. Note that initially the graph is the same as that in Figure 3(a). Let q=Q, k=2 and t_e=t_2, we can observe that: when the time t=t_0, the SAC of Q is $G[C_1]$; at a later time t=t_1, the SAC of Q changes to $G[C_2]$; and then t=t_2, the SAC of Q changes to $G[C_4]$, C_4=$\{Q, C, E\}$. Thus, the result of this query is

$$G_Q(t) = \begin{cases} G[C_1], & t \in [t_0, t_1) \\ G[C_2], & t \in [t_1, t_2) \\ G[C_4], & t \in [t_2, t_2] \end{cases}. \tag{6}$$



(a) t=t_0 (b) t=t_1 (c) t=t_2

Fig. 9. An example of CSAC search.

The dynamics of an SAC are two-fold: (1) its member set may change since its members may move out of the MCC of the SAC, and other members which are not from the SAC may be close to q after their movement; and (2) its location may change, even when its member set remains unchanged after its members' movement. A simple method to address the CSAC search problem is to invoke an SAC search algorithm upon every location change.

This method, however, is very inefficient due to the users' nature of high mobility. Thus, it is desirable to develop fast algorithms.

We observe that, although a user's SACs change frequently, they often share some vertices, and are spatially close or overlapped. Based on this intuition, we design three fast algorithms, i.e., CAppFast, CAppAcc, and CExact+, by extending AppFast, AppAcc and Exact+ respectively. These algorithms generally follow the same framework, and guarantee that the returned SAC achieves an arbitrary small approximation ratio. Note that, we do not extend AppInc and Exact because they are slower than AppFast and Exact+ respectively [45].

CAppFast is a (2+ϵ_F)-approximation continuous algorithm, where $\epsilon_F \geq 0$, and its theoretical approximation ratios is at least 2. The CAppAcc algorithm is extended from AppAcc, which achieves an approximation ratio of 1+ϵ_A, where $0<\epsilon_A<1$ is a value specified by the query user. It is slightly slower than CAppFast, as it spends more effort on finding more accurate solutions. Similarly, we design a fast exact continuous algorithm CExact+ based on Exact+. In the following, we first discuss the framework and then introduce the algorithms one by one.

Remark. In this paper, we mainly focus on the dynamic updates from vertices' locations; for the changes of edges (relationship) of vertices, there are many existing algorithms [35], [36] for maintaining the k-core dynamically, which can be easily incorporated into our framework, and thus we skip them.

5.2 The Framework of CSAC Algorithms

In the framework, we consider the updates one by one and allow any of the extended algorithms to be incorporated. We denote a specific algorithm for static scenario by Algo (one of AppFast, AppAcc, and Exact+), and also denote its extended version for dynamic graph by CAlgo (one of CAppFast, CAppAcc, and CExact+ respectively). We let the vertex that generates the update be p. After p's movement, we denote it by p'. In other words, p and p' are of the same id, but different locations and timestamps. Similarly, we let the SACs and radiuses of MCCs covering them before and after p's movement be Δ and r, Δ' and r' respectively. In addition, for each instance of CAlgo, in the framework we maintain additional variables, which will be updated during running CAlgo. We will discuss them later.

In the framework, we sequentially consider the moving records one by one, and update q's SACs based on a key observation that, if a moving vertex is too far from q's SAC, then we may skip it directly as its movement may not change the SAC; otherwise, we need to call CAlgo to update the SAC. The main technical challenge is how to design a fast algorithm CAlgo, so that we can avoid invoking Algo and reduce the cost of updating the SAC. We will detail this in the following sections.

Algorithm 1 presents the framework. The input is a dynamic spatial graph G, a vertex q, an integer k, a non-negative value ϵ, a future timestamp t_e, and two algorithms. During the dynamic location change process, a list of different SACs of q are outputted. We first find the k-$\widehat{core}$ of q, and put its vertices into X and $PriQueue$, which gives high priorities for vertices near to q (lines 2-3). Then, we initialize a variable t_s as the current timestamp (line 4). Next, in the while loop (lines 5-15), we monitor all the updates and handle them one by one. If the update (from p to p') is the first one, then we initialize Δ and r by calling Algo (lines 6-8); otherwise, we decide whether to update Δ by CAlgo using Lemma 14. In case that we need to update Δ and r, we run CAlgo

Algorithm 1 Framework of CSAC algorithms

```
1: function CSAC(G, q, k, ϵ, t_e, Algo, CAlgo)
2:     find the vertex list X of the k-core^ containing q;
3:     sort vertices of X and store them in PriQueue;
4:     initialize t_s as the current timestamp;
5:     while current time t ≤t_e and has an update (p,p') do
6:         if p.t<t_s and p'.t≥t_s then
7:             call Algo(G, q, k, ϵ) to initialize Δ and r;
8:             output Δ;
9:         else
10:            if p∉X or (|p', q|>2r and |p, q|>2r) then continue;
11:            call CAlgo(G, p, p', q, k, ϵ) to get Δ' and r';
12:            if Δ'≠Δ then
13:                output Δ, and [t_s, t) for Δ';
14:                Δ ← Δ', r ←r', t_s ←t;
15:        update p by p'; //update the graph
16:    output [t_s, t_e] for Δ;
```

(lines 11-14). Finally, we update the graph G (line 15). Note that the duration of each returned SAC is also reported (lines 13,16).

Lemma 14. If $p \in X$, $|p, q| > 2r$, and $|p', q| > 2r$, we have Δ'= Δ, which means the movement of p does not change the SAC.

5.3 The CAppFast Algorithm

Similar to the idea of AppFast, we perform binary search to find the target SAC in CAppFast. The key step of CAppFast is to compute the tight upper and lower bounds, i.e., l' and u', of the binary search, so that we can update SAC efficiently. In the framework, we maintain an additional variable $\widehat{p}$, called *mark* vertex, which is the farthest vertex to q in the returned SAC.

To compute Λ', the updated SAC after p's movement, we consider three cases. (1) If $p.id$=$q.id$, which implies that q moves, we have to run AppFast again. (2) If $p.id \neq q.id$ and p'=$\widehat{p}$, which implies the mark vertex $\widehat{p}$ moves, then we have

- If $|p', q|<r_\Lambda$, then l'=$|p', q|$, u'=$|q, PriQueue.tail|$.
- If $|p', q|\geq r_\Lambda$, then l'=$|q, PriQueue.peak|$, u=$|p', q|$.

(3) If $p.id \neq q.id$ and $p' \neq \widehat{p}$, we have

- $|p', q|<r_\Lambda$, then l'=$|p', q|$, u'=r_Λ.
- $|p', q|\geq r_\Lambda$, then l'=r_Λ, u'=$|p', q|$.

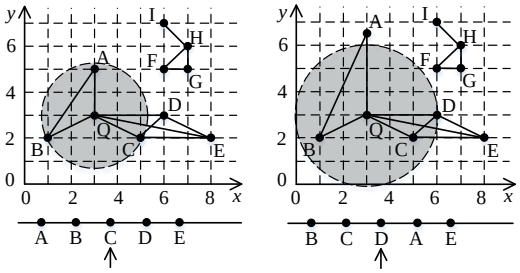
Algorithm 2 presents CAppFast. If q moves, we compute the lower and upper bounds (lines 2-3); otherwise, we check whether p is $\widehat{p}$ or not. If $\widehat{p}$ moves, we get the lower and upper bounds by the $PriQueue$ (lines 4-8); otherwise, we reset the bounds (lines 9-11). After that, we perform the binary search to find the SAC (line 12), which is the same as lines 4-14 of AppFast. Finally, $\widehat{p}$ is updated and Λ' is returned (lines 13-14). For the update of $\widehat{p}$, we can simply reset it as the farthest vertex to q in the returned SAC according to the definition of mark vertex.

Next, we elaborate CAppFast via Example 6.

Example 6. Consider the graph in Figure 10 where t_0 is the current timestamp. Let q=Q, k=2, and t_e=t_1. We first sort the vertices in X: A, B, C, D, E, and put them in $PriQueue$. Then, we follow the framework and run AppFast to initialize Λ which contains Q, A, and B, r_Λ=$|B, Q|$, and $\widehat{p}$=B. Next, A moves out of $O(q, |Q, B|)$ at t=t_2, the queue changes to B, C, D, A, E. The upper and lower bounds of the binary search are l'=r_Λ, u'=$|A, Q|$. After the binary search, we update $\widehat{p}$=D and r_Λ=$|D, Q|$. Note that the SACs are circled.

Algorithm 2 Query algorithm: `CAppFast`

```
1: function CAppFast(G, p, p', q, k, ε_F)
2:     if p'.id=q.id then // case (1)
3:         l'= max_{v∈X∩nb(q)} |q,v|, u'=max_{v∈Λ} |q,v|;
4:     if p'.id=p̂.id then // case (2)
5:         if |p',q| < r_Λ then
6:             l' ← |p',q|, u' ← |q, PriQueue.tail|;
7:         else
8:             l' ← |q, PriQueue.peak|, u' ← |p',q|;
9:     else // case (3)
10:        if |p',q| < r_Λ then l' ← |p',q|, u' ← r_Λ;
11:        else l' ← r_Λ, u' ← |p',q|;
12:    BiSearch(l', u');//lines 4-14 in AppFast
13:    update the additional variable p̂;
14:    return Λ';
```



(a) t=t_0, $\widehat{p}$=B (b) t=t_1, $\widehat{p}$=D

Fig. 10. Illustrating `CAppFast`.

5.4 The CAppAcc Algorithm

Recall that in `AppAcc`, to find a (1+ϵ_A)-approximation solution Γ, we explore a quadtree centered at q. After p's movement, to find an updated (1+ϵ_A) solution Γ' efficiently, we reuse the previous quadtree so that the cost of finding Γ' is minimized.

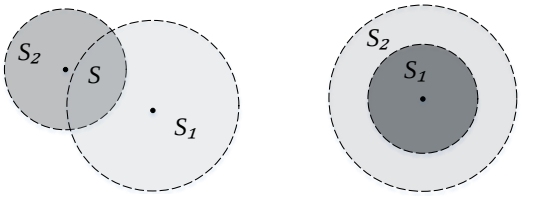
In `CAppAcc`, we need to maintain several additional variables, which are the quadtree $\mathcal{T}$, γ, r_Γ, and δ. These variables are updated after running `CAppAcc`. We denote the corresponding variables of $\mathcal{T}$, γ, r_Γ, and δ after p's movement by $\mathcal{T}'$, γ', $r_\Gamma{}'$, and δ' respectively. To simplify the presentation, we consider two cases in `CAppAcc`, where the first case is that q moves while the second case is that q does not move. We discuss them as follows.

5.4.1 The First Case ($q.id$=$p.id$)

Since the query vertex q moves, the center of quadtree changes. Considering whether the areas covering $\mathcal{T}$ and $\mathcal{T}'$ intersect or not, we have the following two subcases: (1) If $|p', q|\geq\gamma'+\gamma$, then the areas of $\mathcal{T}$ and $\mathcal{T}'$ are separated or tangent, and we need to rerun `AppAcc` to obtain Γ'. (2) If $|p', q|<\gamma'+\gamma$, then the areas of $\mathcal{T}$ and $\mathcal{T}'$ are intersected with each other. We illustrate this in Figure 11(a), where S_1 is the area of $\mathcal{T}$, S_2 is the area of $\mathcal{T}'$, and S the intersection area. Let $r_\Gamma[S]$ denote the minimum radius of an MCC, whose corresponding anchor is in S and contains a feasible solution. Then, we need to find a feasible solution and the radius of MCC covering it is $r_\Gamma{}'$=$\min\{r_\Gamma[S], r_\Gamma{}'[S_2{-}S]\}$. For the area S, we can search the community by directly using $\mathcal{T}$; while for $S_2{-}S$, we need to search the community using $\mathcal{T}'$.

5.4.2 The Second Case ($q.id$≠$p.id$)

In this case, the center of the quadtree does not change after q's movement (i.e., it is at q), but its size may change. We proceed to discuss the algorithm by considering three subcases: (1) If γ=γ' and δ=δ', then we have Γ'=Γ. The reason is that, by Lemma 4, we have $1/2\delta'\leq {r_{opt}}' \leq \gamma'$. This implies that if $\frac{r_\Gamma}{{r_{opt}}'} \leq \frac{r_\Gamma}{1/2\delta'} = \frac{r_\Gamma}{1/2\delta} \leq 1+\epsilon_A$, we have $\frac{{r_\Gamma}'}{{r_{opt}}'} \leq 1+\epsilon_A$. (2) If $\gamma'\leq\gamma$ and $\delta\neq\delta'$, the movement of p is likely to change the fixed vertexes of the MCC covering the optimal solution. So we need to search anchor points in $\mathcal{T}'$. Since $r_{opt} \leq r_\Gamma \leq \gamma'$, we can narrow the area containing $\mathcal{T}'$ as $O(q, r_\Gamma)$ and find Γ' from $\mathcal{T}'$. (3) If $\gamma' > \gamma$, then the area of $\mathcal{T}$ is fully contained by the area of $\mathcal{T}'$. We explain this in Figure 11(b), where S_1, S_2 are the areas of $\mathcal{T}$ and $\mathcal{T}'$ respectively. It is easy to observe that, r'_Γ=$\min\{r_\Gamma[S_1], {r_\Gamma}'[S_2 - S_1]\}$. In addition, since q does not move, the center of the quadtree does not change, we only need to update $\mathcal{T}$, if S_2 is larger than S_1 after running `CAppAcc`.



(a) S_1 intersects S_2 (b) S_1 is covered by S_2

Fig. 11. Illustrating `CAppAcc`.

5.4.3 The Overall Algorithm of CAppAcc

We present the overall steps of `CAppAcc` in Algorithm 3. It first computes γ' using `AppFast` and then considers two cases for finding Γ' separatively. The first case is that q moves (lines 3-10), and the second case is that q does not move (lines 11-21). The steps for each case exactly follow our previous discussion. After obtaining Γ', we update all the additional variables introduced in the framework (line 22), and return Γ' (line 23).

Algorithm 3 Query algorithm: `CAppAcc`

```
1: function CAppAcc(G, p, p', q, k, ε_A)
2:     compute γ' using AppFast;
3:     if p'.id=q.id then //The first case
4:         if |p',q| ≥ γ' + γ then
5:             run AppAcc(G, q, k, ε_A) to get Γ';
6:         else
7:             obtain the intersection area S;
8:             compute r_S using T in area S;
9:             compute r_S' in S2–S;
10:            find Γ' such that r_Γ'=min(r_S, r_S');
11:    else//The second case
12:        if γ'=γ and δ'=δ and 2r_Γ/δ ≤ 1+ε_A then
13:            r'_Γ ← r_Γ, Γ' ← Γ;
14:        else
15:            if γ' ≤ γ then
16:                find Γ' in O(q, r_Γ) using T;
17:            else
18:                obtain the intersection area S;
19:                compute r_S using T in area S;
20:                compute r_S' in S2–S;
21:                find Γ' such that r_Γ'=min(r_S, r_S');
22:    update additional variables T, γ, r_Γ, and δ;
23:    return Γ';
```

5.5 The CExact+ Algorithm

Recall that in `Exact+`, we first get the anchor points using `AppAcc` and then use the annular region to prune a large number of fixed vertices. In `CExact+`, we let ${r_{opt}}'$ be the radius of MCC covering the optimal solution Ψ' after p's movement. We follow the framework, and maintain two additional variables F_1 and R_1,

where F_1 is the set of the candidate fixed vertices and R_1 is the set of the fixed vertex for the latest optimal solution Ψ. Moreover, we propose Lemmas 15 and 16, which greatly reduce the size of the annular region and improve the query efficiency.

Recall that when computing Ψ in `Exact+`, we have introduced variables δ, r_Γ, γ, r_-, r_+. We denote these corresponding variables when computing Ψ' by δ', $r_\Gamma{}'$, γ', $r_-{}'$, $r_+{}'$ respectively. We now present two interesting lemmas.

Algorithm 4 Query algorithm: `CExact+`

1: **function** CEXACT+(G, p, p', q, k, ϵ_A)
2: run CAPPACC(G, p, p', q, k, ϵ_A) to get F_1';
3: obtain r' and r'_+ using Lemmas 15 and 16;
4: **if** $F_1 = F_1{}'$ **and** $p' \notin F_1$ **then return** Ψ; // case (a)
5: **if** $F_1 = F_1{}'$ **and** $p' \in F_1$ **and** $p' \notin R_1$ **then** //case (b)
6: $\Psi' \leftarrow$ enumFix(p',$F_2{}'$,$F_3{}'$);
7: **else** // case (c)
8: $\Psi' \leftarrow$ enumFix($F_1{}'$,$F_2{}'$,$F_3{}'$);
9: update additional variables F_1 and R_1;
10: **return** Ψ';

Lemma 15. The range of $r_{opt}{}'$ is

$$\max\left\{\frac{\delta'}{2\gamma}r_{opt}, \frac{r_\Gamma{}'}{1+\epsilon_A}\right\} \leq r_{opt}{}' \leq \min\left\{\frac{2\gamma'}{\delta}r_{opt}, r_\Gamma{}'\right\}. \quad (7)$$

Lemma 16. If δ'=δ and $\frac{2r_\Gamma}{\delta} - 1 \leq \epsilon_A \leq \frac{2\gamma'}{\delta} - 1$, then the range of r_{opt} can be further contracted as

$$\max\left\{\frac{\delta'}{2\gamma}r_{opt}, \frac{r_\Gamma{}'}{1+\epsilon_A}\right\} \leq r_{opt}{}' \leq r_\Gamma{}'. \quad (8)$$

Similar to `Exact+`, we can compute $r_-{}'$ and $r_+{}'$. After that, we use the annular region to find the set F_1' of candidate fixed vertices and the set R_1' of fixed vertices. To improve the efficiency of query, we further consider three cases by proposing Lemma 17.

Lemma 17. Let $F_1{}'$, $F_2{}'$, and $F_3{}'$ be the corresponding variables of F_1, F_2 and F_3 in `Exact+` after p's movement. We have 3 cases: (a) If F_1=F_1' and $p' \notin F_1$, we can conclude that that the movement of p will not affect F_1. In other words, we have Ψ'=Ψ. (b) If F_1=F_1' and $p' \in F_1$, and $p' \notin R_1$, then we only need to enumerate all the combinations of three vertices, where one is p' and the other two are from $F_2{}'$ and $F_3{}'$ respectively, to find Ψ'. (c) For other cases that cannot be handled by cases 1 and 2, we need to enumerate all the combinations of three vertices from $F_1{}'$, $F_2{}'$, and F_3' respectively to compute Ψ'.

We present `CExact+` in Algorithm 4. It first runs `CAppAcc` and gets the set F_1' of fixed vertices (line 2). Then, it uses the annular contraction to get the tight upper and lower bounds (line 3). Next, it computes the optimal solution Ψ' considering three different cases (line 4 for case (a), lines 5-6 for case (b), and lines 7-8 for case (c)) in Lemma 17. The additional variables are updated (line 9). Finally, the SAC Ψ' is returned (line 10).

6 EXPERIMENTAL RESULTS

6.1 Setup

Datasets. We consider four real datasets: Brightkite[3], Gowalla[3], Flickr[4] and Foursquare[5]. For all the datasets, each vertex represents a user and each link represents the friendship between two users. Both Brightkite dataset and Gowalla dataset contain a collection of check-in data shared by users of Brightkite service and Gowalla service. In particular, for Brightkite dataset, there are 4,491,143 checkins collected during the period of Apr. 2008 - Oct. 2010 on 772,783 distinct places; the Gowalla dataset contain 6,442,892 checkins collected on 1,280,969 places. In Flickr dataset, we mark the user a location if she has taken a photo there. The Foursquare dataset [3] is extracted from Foursquare website. Each user has a location of her hometown position. It contains 33,278,683 checkin records collected during a period from April 2012 to September 2013. For Brightkite, Gowalla, and Foursquare datasets, we consider the users' locations can be both static (for SAC search) and dynamic (for CSAC search). The static location associated with a user is the place she checks in (or takes photos) most frequently, or hometown location. Note that users without locations are skipped.

TABLE 4
Datasets used in our experiments.

Type	Name	Vertices	Edges	$\widehat{d}$	BC	CC
Real	Brightkite	51,406	197,167	7.67	7.34E-5	0.1795
	Gowalla	107,092	456,830	8.53	3.40E-5	0.2487
	Flickr	214,698	2,096,306	19.5	1.65E-5	0.1113
	Foursquare	2,127,093	8,640,352	8.12	1.68E-6	0.1044
Synthetic	Syn1	30,000	300,000	20	1.26E-4	0.0077
	Syn2	400,000	4,000,000	20	1.10E-5	0.0006

We have also performed experiments on synthetic datasets. We are not aware of any existing spatial graph data generators. Thus, we create synthetic data in the following way. First, we use GTGraph[6], a well-known graph generator, to generate a (non-spatial) graph. We adopt the default parameter values of GTGraph. The degrees of the graph follow a power-law distribution, which is often exhibited in social networks. To generate the location of each graph vertex, we first randomly select a vertex v and give it a random position in the $[0, 1] \times [0, 1]$ space. Then we place v's neighbors at random positions, whose distances follow a normal distribution with mean μ and standard deviation σ. We repeat this step for other vertices, starting from v's neighbors, until every vertex has a location. We set μ=0.09 and σ=0.16; these values are derived from the Brightkite dataset. Following these settings, we create two spatial graphs, namely Syn1 and Syn2.

The statistics of each dataset are summarized in Table 4, where the average degree ($\widehat{d}$), the average betweenness centrality (BC), and the clustering coefficient (CC) of each vertex in the dataset are included. Without loss of generality, we normalize all the locations of each dataset into the unit square $[0, 1]^2$.

Parameters. We consider 5 parameters: k (denoting the minimum degree), ϵ_F (the parameter of `AppFast`), ϵ_A (the parameter of `AppAcc`), θ (the parameter of θ-SAC search), and the percentage of vertices n. In general, the value of k should not be too small or too large, because a small value of k (e.g., k=1 or 2) may result in a community with many vertices having just few neighbors; a too large value of k (k=20) may result in many "false queries (i.e., queries returning empty results). The ranges of the parameters and their default values are shown in Table 5. The default values of ϵ_F and ϵ_A are set as 0.5, since these values practically result in good approximation ratios with reasonable efficiency. Note that when varying n for scalability testing, we randomly extract subgraphs of 20%, 40%, 60%, 80% and 100%

3. http://snap.stanford.edu/data/index.html
4. https://www.flickr.com/
5. https://archive.org/details/201309_foursquare_dataset_umn
6. http://www.cse.psu.edu/~madduri/software/GTgraph/

TABLE 5
Parameter settings.

Parameter	Range	Default
k	4, 7, 10, 13, 16	4
ϵ_F (AppFast)	0.0, 0.5, 1.0, 1.5, 2.0	0.5
ϵ_A (AppAcc)	0.01, 0.05, 0.1, 0.5, 0.9	0.5
θ	10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}	10^{-4}
n	20%, 40%, 60%, 80%, 100%	100%

vertices of the original graph with a default value of 100%. When varying a certain parameter, the values for all the other parameters are set to their default values.

Queries of SAC search. For each dataset, we randomly select 200 query vertices with core numbers of 4 or more. Such a core number constraint ensures a meaningful community containing the query vertex. In the results reported in the following, each data point is the average result for these 200 queries. We use the term "AppFast(ϵ)" ("AppAcc(ϵ)") to denote the algorithm AppFast (AppAcc) with the parameter ϵ_F=ϵ (ϵ_A=ϵ).

Queries of CSAC search. We use three real datasets, i.e., Brightkite, Gowalla, and Foursquare, to evaluate the efficiency of CSAC search. In each of these datasets, users have continuous check-in records. We first sort all the checkin records in chronological order, and then we divide them into two groups G_1 and G_2, where G_2 contains the checkin records int the last 30 days while G_1 contains the other records. In the experiments, we first update users' locations by records in G_1, and then simulate users' movement by records from G_2. For each proposed algorithm (CAppFast, CAppAcc, and CExact+), we design its counterpart by simply calling the corresponding SAC search algorithm repeatedly without any optimization. The parameter settings are the same as that of SAC search. In the results reported, each data point is the average result for these queries.

We implement all the algorithms in Java, and run experiments on a machine having a quad-core Intel i7-3770 3.40GHz processor and 32GB of memory, with Ubuntu installed.

6.2 Effectiveness Evaluation

6.2.1 Comparison with the State-of-the-Arts

In this subsection, we show that SAC search returns communities with higher spatial cohesiveness compared with the state-of-the-art CR methods: Global [10], Local [11] and GeoModu [18]. The first two methods are CS methods designed for non-spatial graphs, while GeoModu is a CD method for spatial graphs. We also compare it with θ-SAC search. We briefly introduce these algorithms as follows (Let q be a query vertex):

• Global: it finds the k-$\widehat{core}$ containing q.

• Local: it expands and explores from q, until it forms a subgraph whose minimum vertex degree is at least k.

• GeoModu: it first redefines the weight of each edge of graph G as $e_{i,j} = \frac{1}{d_{i,j}{}^{\mu}}$, where $d_{i,j}$=$|v_i, v_j|$ and μ (1 or 2) is a decay factor, and then detects the communities using modularity maximization. Given a query vertex, we return the community which contains it.

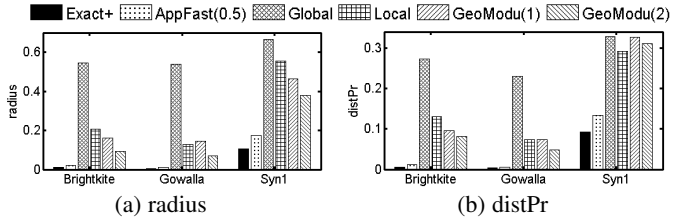
• θ−SAC search: it first performs BFS search on G starting at q to find a set S of vertices, which are connected with q and in the circle $O(q, \theta)$, and then returns the k-$\widehat{core}$ containing q in $G[S]$.

Both Global and Local use the minimum degree metric for structure cohesiveness. GeoModu has two variants, i.e., GeoModu(1) and GeoModu(2), as the typical values of μ are 1 and 2. To measure the spatial cohesiveness of a community G_q with MCC $O(c, r)$, we introduce two metrics as follows:

• **radius**: the value of radius r.

• **distPr**: average pairwise distance of vertices of G_q.

Intuitively, lower values of these metrics for a community imply that it achieves higher spatial cohesiveness. To compare these methods, we consider both exact and approximation algorithms. We search communities using these algorithms and compute the average values of above metrics for these communities. We report the results on Brightkite and Gowalla datasets in Figure 12.



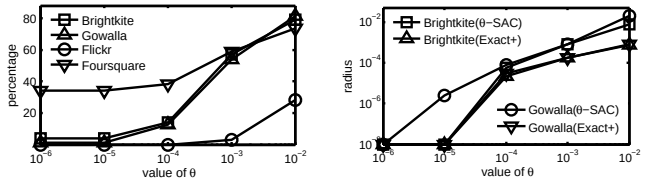
(a) radius (b) distPr

Fig. 12. Comparison with existing CD and CS methods.

1. CS comparison. We see that Local performs better than Global, as it finds communities through local expansion. The vertices of communities returned by Global and Local spread in larger areas than those of SAC search methods. For example, the average radii of the MCCs covering the communities of Global and Local are respectively 50 and 20 times larger than that of our approach. The main reason is that they overlook the spatial locations. Note that although G_q is in the smallest MCC, G_q may not be the subgraph with the minimum number of vertices satisfying the minimum degree metric. In other words, a proper subset of vertices in G_q may form a qualified community, which has the same MCC as that of SAC search. Among the SAC search methods, the exact algorithm Exact+ achieves better spatial cohesiveness than approximation algorithms consistently.

2. CD comparison. Since GeoModu considers both links and locations, the returned communities achieve better spatial cohesiveness than Global and Local. However, the average radius and distPr values of GeoModu are larger than SACs because a non-trivial number of queries in GeoModu return communities whose MCCs are smaller than those of SAC (e.g., 19% and 18% of queries whose communities returned by GeoModu(1) are in MCCs with smaller radii than those of Exact+, in Brightkite and Gowalla datasets respectively). However, the structure cohesiveness of communities detected by GeoModu is weaker. For example, the corresponding average degrees of vertices in communities returned by GeoModu(1) and GeoModu(2) on Brightkite dataset are 2.2 and 1.1. This is because GeoModu partitions the graph into clusters using a global criterion, i.e., Geo-Modularity [18], with no reference to the query vertices.

3. Comparison with θ-SAC search. We vary the value of θ in θ-SAC search, and compute the percentage of queries returning non-empty subgraphs. Figure 13(a) reports the results. Notice that the percentage is low when θ is small. This is because many users' SACs are spread in large areas. Also, the percentage varies greatly for different datasets. Thus, setting a proper value of θ is not easy. In contrast, SAC search does require the specification of θ, and it always returns an SAC, if there is any. For queries returning non-empty SACs, we compute the average radius of MCCs covering these SACs. We also compute the average radius for MCCs of SACs found by Exact+. Figure 13(b) compares their results. We observe that the average radius of MCCs covering SACs found



(a) percentage (b) radius

Fig. 13. Results of θ-SAC search.

by θ-SAC search is 5 to10 times larger than that of `Exact+`. This means that SAC search achieves better spatial cohesiveness than this variant. Hence, SAC search is easier to be used, and also achieves higher spatial cohesiveness than θ-SAC.

6.2.2 Adaptability to Location Changes

To study the adaptability of location changes of CSAC search, we use a "dynamic" spatial graphs, where vertices' locations change frequently. We consider Brightkite dataset and assume the link relationships do not change. We first sort all the checkin records in chronological order. Then, we divide them into two groups R_1 and R_2, where R_1 contains records collected before 2010 and R_2 contains the remaining records. Finally, we compute the total travel distance of each user, by adding up the distances between each consecutive pair of checkins, and select a set Q of 100 query users, who travel the longest and have at least 20 friends.

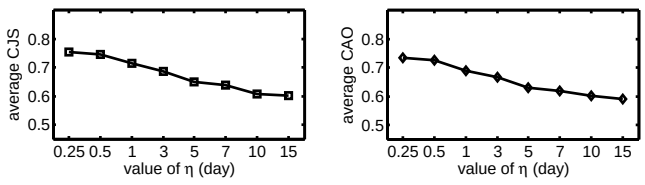
To perform queries, we first go through checkin records in R_1 and update users' locations according to their latest checkin timestamps. Then, for each user $q \in Q$, we do the same operation for records in R_2, and if the record was generated by q, we search her SAC by `CExact+`. Finally, we get a list of SACs, L_q={$C_1, C_2, \cdots, C_l$}, where $C_i(1 \leq i \leq l)$ is an SAC at the timestamp of the i-th checkin record, and l is the number of q's check-in records.

To measure the overlap of member sets and spatial areas between two communities C_i and C_j, we define two metrics: *community jaccard similarity* (CJS) and *community area overlapping* (CAO), based on the classical Jaccard similarity.

$$CJS(C_i, C_j) = \frac{V(C_i) \cap V(C_j)}{V(C_i) \cup V(C_j)}, \quad (9)$$

$$CAO(C_i, C_j) = \frac{A(C_i) \cap A(C_j)}{A(C_i) \cup A(C_j)}, \quad (10)$$

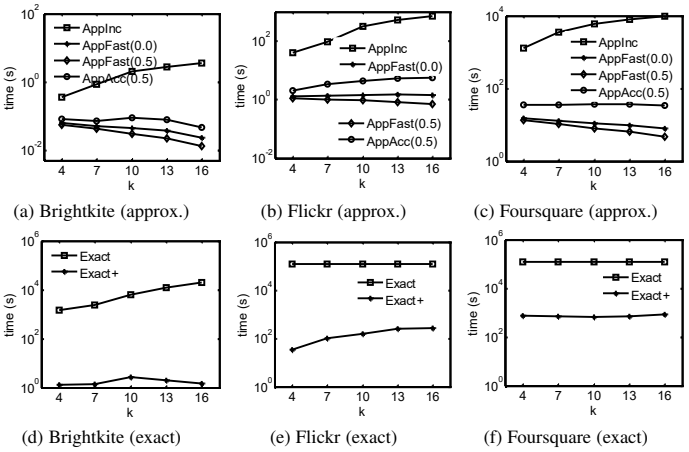
where $V(C_i)$ is the member set of C_i and $A(C_i)$ is the area of the MCC covering C_i. Notice that $CJS(C_i, C_j)$ and $CAO(C_i, C_j)$ range from 0 to 1. A smaller value of CJS (CAO) implies lower overlapping of community member sets (spatial areas).



(a) CJS (b) CAO

Fig. 14. Effectiveness on dynamic spatial graph.

To show how the values of CJS and CAO vary with time, we select communities from L_q, where the time gap between each pair of communities is at least η, and compute their CJS and CAO values. We report their average results in Figure 14, where η varies from 0.25 day to 15 days. From Figure 14(a), we observe that, the CJS decreases as the time threshold increases. For example, after 6 hours, the value of CJS decreases to 75%, which implies the SACs indeed have changed, but they share a remarkable proportion of common members. Figure 14(b) shows that the SACs' locations change frequently over time as well. In addition, we plot the SACs of two mobile users in Figure 2. Thus, these results well confirm that, CSAC search has high adaptability to location changes.



(a) Brightkite (approx.) (b) Flickr (approx.) (c) Foursquare (approx.)
(d) Brightkite (exact) (e) Flickr (exact) (f) Foursquare (exact)

Fig. 15. Efficiency of SAC search on real datasets.

6.3 Efficiency Evaluation of SAC Search

We report the efficiency results of SAC search on three real datasets in Figure 15. The results on other datasets are in [43].

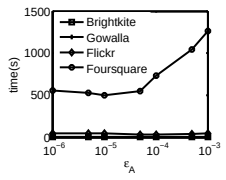
1. Effect of k for approximation algorithms. We show the results in Figures 15(a)-(c). `AppFast` runs consistently faster than `AppInc` and `AppAcc`. For example, for the largest real dataset Foursquare, `AppFast(0.0)` is at least two orders of magnitude faster than `AppInc`, although they return the same SACs. `AppFast(0.0)` is 2 to 5 times faster than `AppAcc(0.5)`. This is because `AppFast` has a lower time complexity. In addition, the running time of `AppFast` decreases as the value of k increases. This is because $O(q, \delta)$ becomes larger as k increases, and finding a larger $O(q, \delta)$ tends to need less binary search.

The running time of `AppInc` increases clearly as the value of k grows, because for a larger value of k, the corresponding $O(q, \delta)$ is also larger. As it finds $O(q, \delta)$ starting from the query vertex q incrementally, a larger value of k results in a higher cost.

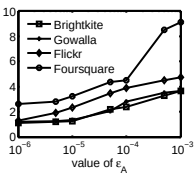
`AppAcc(0.5)` is slower than `AppFast`. This is because its first step is to run `AppFast` and it needs extra effort to find smaller MCCs. Also, its time cost tends to be stable. As discussed before, the number of anchor points is mainly affected by ϵ_A. Since ϵ_A is always 0.5 for different k, the numbers of anchor points are the same, and thus the running time remains stable.

2. Effect of k for exact algorithms. Figure 16(a) shows that the efficiency of `Exact+` is not very sensitive to ϵ_A on all real datasets except Foursquare. Figure 16(b) shows $|F_1|$ increases with ϵ_A; that is, fewer vertices are pruned as ϵ_A grows. Recall in Section 5.5 that `Exact+` is composed of two phases. When ϵ_A is small, the cost of Phase (1) dominates the overall cost; as ϵ_A increases, the cost of Phase (2) grows. Thus, there is a local minimum in Figure 16(a). In practice, a sensitivity test can be done to choose a suitable value of ϵ_A.

Figures 15(d)-(f) show the efficiency results of exact algorithms (ϵ_A=10^{-4}). We skip the results of `Exact`, if a single query takes more than 10 hours. Clearly, `Exact` performs extremely slow, even on the smallest dataset, which calls for the design of more efficient algorithms. `Exact+` is at least four orders of

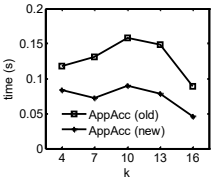


(a) Efficiency

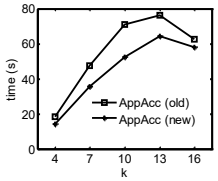


(b) Size of F_1

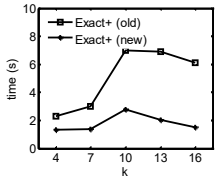
Fig. 16. Effect of ϵ_A on Exact+.



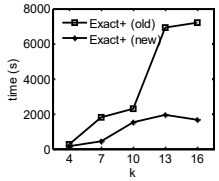
(a) Brightkite (AppAcc)



(b) Syn2 (AppAcc)

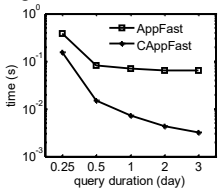


(c) Brightkite (Exact+)

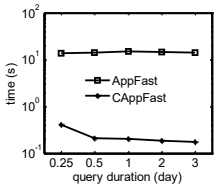


(d) Syn2 (Exact+)

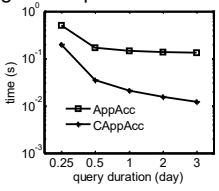
Fig. 17. Improvement of AppAcc and Exact+.



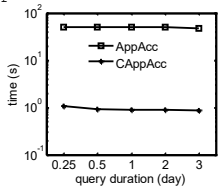
(a) Brightkite ($|t_e–t_s|$)



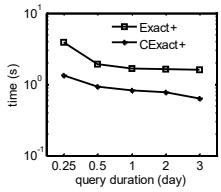
(b) Foursquare ($|t_e–t_s|$)



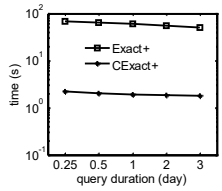
(c) Brightkite ($|t_e–t_s|$)



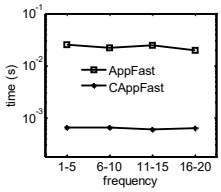
(d) Foursquare ($|t_e–t_s|$)



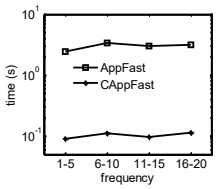
(e) Brightkite ($|t_e–t_s|$)



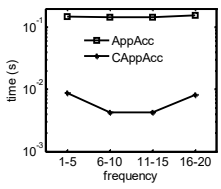
(f) Foursquare ($|t_e–t_s|$)



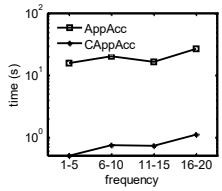
(g) Brightkite (frequency)



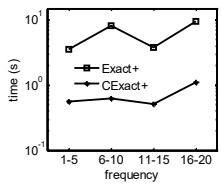
(h) Foursquare (frequency)



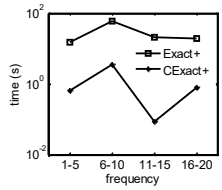
(i) Brightkite (frequency)



(j) Foursquare (frequency)



(k) Brightkite (frequency)



(l) Foursquare (frequency)

Fig. 18. Efficiency of CSAC search.

magnitude faster than Exact. This is because, it uses AppAcc to find narrow annular regions, in which fixed vertices are supposed to be contained, and thus the number of fixed vertices needed to be enumerated is reduced significantly. The performance of Exact+ either slightly increases or decreases as the value of k increases. This is because, after pruning by the annular regions, the numbers of fixed vertices left may be different, but generally larger datasets have more fixed vertices, and thus more time cost is needed. Additionally, since Exact+ runs AppAcc in the first step, it is slower than AppAcc, but it takes reasonable time, i.e., few seconds, on moderate-size graphs like Brightkite and Syn1.

3. Improvement of AppAcc and Exact+. In this experiment, we consider the Brightkite and Syn2 datasets. We vary the value of k, and compare the efficiency of the improved AppAcc and Exact+ with their old versions published in [45]. The results are reported in Figures 17(a)-(d). We observe that, AppAcc and Exact+ are faster than their old versions, i.e., achieving an acceleration up to 2 to 5 times. In addition, their performance trends are similar with those discussed before.

4. Results on noisy graphs. We consider two new datasets with anomalous vertices and also add different kinds of noise vertices into the datasets of Table 4. The efficiency results on these datasets show that our algorithms generally are very robust to noise vertices, and their parameters are not very sensitive to the noise. In addition, we have evaluated the effect of vertices' betweenness centrality and clustering coefficient on efficiency. Due to the space limitation, we present the detailed experimental results and discussions in the supplemental materials.

6.4 Evaluation of CSAC Search

In this section, we study the efficiency of CSAC search and report results in Figure 18 (results on Gowalla dataset are in [43]).

1. Effect of query duration $|t_e–t_s|$. In this experiment, we randomly select 100 query users, and set the query duration as 0.25, 0.5, 1.0, 2.0, and 3 days respectively. Then, we run CAlgo and Algo to answer the queries of CSAC search. The efficiency results are reported in Figures 18(a)-(f). The exact algorithm CExact+ is slower than the approximation algorithms, and CAppFast is faster than CAppAcc. We observe that, when the value of query duration increases, the average time of both AppFast and CAppFast becomes smaller, but the decrease of CAppFast is more pronounced. The reason is that, as the query duration increases, CAppFast makes full use of the previous results and does not require repeated computation. Also, with the value of $|t_e–t_s|$ increased, the gap between CAppFast and CAppFast grows, and CAppFast is up to 80 times faster than CAppFast. Thus, CAppFast is more suitable for dynamic spatial graphs.

Besides, CAppAcc is over an order of magnitude faster than AppAcc, and the gap between them also increases as the length of query duration grows and their performance trends are similar with CAppFast, as discussed before. In addition, CExact+ is faster than Exact on dynamic graphs. The performance trends are also similar with those of CAppFast and CAppAcc.

2. Effect of update frequency. In this experiment, we first count the numbers of checkin times for each user. Then, we divide users into four groups according to their check-in frequency, each corresponding to 1∼5, 6∼10, 11∼15, and 16∼20 times. Next, for each group we randomly select 100 users. Finally, we perform CSAC search for users in each group, and report the results in Figures 18(g)-(l). We observe that, when the value of frequency increases, the efficiency of AppFast and CAppFast does not change much, so both of them are not sensitive to the frequencies of users' movement. Again, CAppFast is still much more than AppFast, which is consistent with the previous results.

Besides, the performance of CAppAcc is not very stable when varying the value of frequency. This is because, if the query user q does not move, CAppAcc needs little computational cost to update the SAC; otherwise, CAppAcc needs to consider more anchor points, which involves more cost. Again, the gap between AppAcc and CAppAcc is over an order of magnitude. In addition, the results of CExact+ are similar with those of CAppAcc.

7 CONCLUSIONS

In this paper, we examine the online search of communities that exhibit both structure and spatial cohesiveness on spatial graphs. Given a vertex q of a graph G, we study the SAC search problem, which finds the community containing q within the smallest minimum covering circle (MCC). We propose two exact algorithms, and three efficient approximation algorithms. We further study the problem of continuous SAC (CSAC) search on dynamic spatial graphs, and propose three efficient algorithms. We evaluate the algorithms on both real and synthetic datasets, and find that SAC search achieves better effectiveness than the existing methods. Also, the approximation algorithms are very fast on large datasets. In addition, the algorithms of CSAC search are very efficient.

In the future, we will examine other structure cohesiveness measures (e.g., k-truss and k-clique) and spatial cohesiveness measures (e.g., pair-wise vertex distances). We will study how to adapt our SAC search solutions for directed spatial graphs by using D-core [46], which is extended from k-core for directed graphs. It is also of interest to find SAC from spatial graphs where edges are with weights denoting proximity among vertices. We will develop a web-based prototype system so that users can easily perform queries and visualize the returned SACs on the spatial map.

REFERENCES

[1] N. Armenatzoglou et al., "A general framework for geo-social query processing," in *PVLDB*, vol. 6, no. 10, 2013, pp. 913–924.

[2] W. Zhang et al., "Combining latent factor model with location features for event-based group recommendation," in *KDD*, 2013, pp. 910–918.

[3] M. Sarwat et al., "Lars*: a scalable and efficient location-aware recommender system," *TKDE*, vol. 26, no. 6, pp. 1384–1399, 2014.

[4] Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. Yang, "Scalable algorithms for nearest-neighbor joins on big trajectory data," *TKDE*, vol. 28, no. 3, pp. 785–800, 2016.

[5] K. M. MacQueen et al., "What is community? an evidence-based definition for participatory public health," *AJPH*, vol. 91, no. 12, pp. 1929–1938, 2001.

[6] B. R. S. Saurabh Sen, Anshuman, *International Perspectives on Socio-Economic Development in the Era of Globalization*. Business Science Reference (an imprint of IGI Global), USA, 2016.

[7] R. E. Ricklefs, "Community diversity: relative roles of local and regional processes." *Science(Washington)*, vol. 235, no. 4785, pp. 167–171, 1987.

[8] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, pp. 026–113, 2004.

[9] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[10] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *KDD*, 2010, pp. 939–948.

[11] W. Cui, Y. Xiao, H. Wang, and W. Wang, "Local search of communities in large graphs," in *SIGMOD*. ACM, 2014, pp. 991–1002.

[12] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *SIGMOD*. ACM, 2013, pp. 277–288.

[13] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," in *PVLDB*, vol. 8, no. 5, 2015, pp. 509–520.

[14] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," in *PVLDB*, 2015, pp. 276–287.

[15] Y. Fang, R. Cheng, S. Luo, and H. Jiafeng, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.

[16] D. Guo, "Regionalization with dynamically constrained agglomerative clustering and partitioning," *IJGIS*, vol. 22, no. 7, pp. 801–823, 2008.

[17] P. Expert et al., "Uncovering space-independent communities in spatial networks," *PNAS*, vol. 108, no. 19, pp. 7663–7668, 2011.

[18] Y. Chen, J. Xu, and M. Xu, "Finding community structure in spatially constrained complex networks," *IJGIS*, vol. 29, no. 6, pp. 889–911, 2015.

[19] M. Barthélemy, "Spatial networks," *Physics Reports*, vol. 499, no. 1, pp. 1–101, 2011.

[20] A. Lancichinetti and S. Fortunato, "Limits of modularity maximization in community detection," *Phys. Rev. E*, vol. 84, no. 6, p. 066122, 2011.

[21] P. Manchanda et al, "Social dollars: the economic impact of customer participation in a firm-sponsored online customer community," *Marketing Science*, vol. 34, no. 3, pp. 367–387, 2015.

[22] Y. Chon, N. D. Lane, F. Li, H. Cha, and F. Zhao, "Automatically characterizing places with opportunistic crowdsensing using smartphones," in *UbiComp*. ACM, 2012, pp. 481–490.

[23] J. Scott, *Social network analysis*. Sage, 2012.

[24] M. E. Newman, "Modularity and community structure in networks," *PNAS*, vol. 103, no. 23, pp. 8577–8582, 2006.

[25] P. Chopade and J. Zhan, "A framework for community detection in large networks using game-theoretic modeling," *IEEE Transactions on Big Data*, vol. 3, no. 3, pp. 276–288, 2017.

[26] P. Chopade, J. Zhan, "Structural and functional analytics for community detection in large-scale complex networks," *Journal of Big Data*, vol. 2, no. 1, 2015.

[27] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *PNAS*, vol. 99, no. 12, pp. 7821–7826, 2002.

[28] Y. Fang, R. Cheng, S. Luo, H. Jiafeng, and H. Kai, "C-explorer: Browsing communities in large graphs," *PVLDB*, vol. 10, no. 12, 2017.

[29] J. Hu et al., "Querying minimal steiner maximum-connected subgraphs in large graphs," in *CIKM*. ACM, 2016, pp. 1241–1250.

[30] S. Babu and J. Widom, "Continuous queries over data streams," *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.

[31] G. S. Iwerks et al., "Continuous k-nearest neighbor queries for continuously moving points with updates," in *PVLDB*, 2003, pp. 512–523.

[32] D. Wu et al., "Efficient continuously moving top-k spatial keyword query processing," in *ICDE*. IEEE, 2011, pp. 541–552.

[33] Y. Tao and D. Papadias, "Spatial queries in dynamic environments," *TODS*, vol. 28, no. 2, pp. 101–139, 2003.

[34] M. F. Mokbel et al, "Sina: Scalable incremental processing of continuous queries in spatio-temporal databases," in *SIGMOD*, 2004, pp. 623–634.

[35] A. E. Sarıyüce et al., "Incremental k-core decomposition: algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, pp. 425–447, 2016.

[36] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *TKDE*, vol. 26, no. 10, pp. 2453–2465, 2014.

[37] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[38] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *arXiv*, 2003.

[39] D. J. Elzinga and D. W. Hearn, "The minimum covering sphere problem," *Management science*, vol. 19, no. 1, pp. 96–104, 1972.

[40] J. Elzinga et al., "Geometrical solutions for some minimax location problems," *Transportation Science*, vol. 6, no. 4, pp. 379–394, 1972.

[41] N. Megiddo, "Linear-time algorithms for linear programming in r3 and related problems," in *FOCS*. IEEE, 1982, pp. 329–338.

[42] T. Guo, X. Cao, and G. Cong, "Efficient algorithms for answering the m-closest keywords query," in *SIGMOD*. ACM, 2015, pp. 405–418.

[43] Y. Fang et al, "On spatial-aware community search (technical report)," http://www.cs.hku.hk/research/techreps/document/TR-2016-04.pdf.

[44] R. A. Finkel and J. L. Bentley, "Quad trees: a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[45] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, "Effective community search over large spatial graphs," *PVLDB*, vol. 10, no. 6, pp. 709–720, 2017.

[46] C. Giatsidis et al, "D-cores: Measuring collaboration of directed graphs based on degeneracy," in *ICDM*. IEEE, 2011, pp. 201–210.

[47] L. Akoglu, M. McGlohon, and C. Faloutsos, "Oddball: Spotting anomalies in weighted graphs," in *PAKDD*. Springer, 2010, pp. 410–421.

[48] S. Shekhar et al, "Detecting graph-based spatial outliers: algorithms and applications (a summary of results)," in *KDD*, 2001, pp. 371–376.

Yixiang Fang received the Ph.D. degree from the University of Hong Kong (HKU) in 2017. Currently, he is a postdoc researcher in The University of New South Wales, under the supervision of Prof. Xuemin Lin. His research interests mainly focus on graph queries.

Zheng Wang is a master student in the Department of Computer Science, HKU. His research interests focus on graph queries.

Reynold Cheng is an associate professor in the Department of Computer Science, University of Hong Kong. He has served as a PC member and reviewer for international conferences (e.g., SIGMOD, VLDB, ICDE and KDD) and journals (e.g., TKDE, TODS, VLDBJ and IS). He is an associate editor of TKDE, and on the EIC selection committee for TKDE. He is a member of the IEEE and ACM.

Xiaodong Li is a PhD student in the Department of Computer Science, University of Hong Kong. His research area is graph management.

Siqiang Luo is a PhD student in the Department of Computer Science, University of Hong Kong. His research area is graph management.

Jiafeng Hu is a PhD student in the Department of Computer Science, University of Hong Kong. His research area is graph management.

Xiaojun Chen is an assistant professor in Shenzhen University. His research interests include data mining and machine learning.

THE SUPPLEMENTAL MATERIALS

8 PROOFS OF LEMMAS

Lemma 1. Given a set S ($|S| \geq 2$) of vertices, its MCC can be determined by at most three vertices in S which lie on the boundary of the circle. If it is determined by only two vertices, then the line segment connecting those two vertices must be a diameter of the circle. If it is determined by three vertices, then the triangle consisting of those three vertices is not obtuse.

Proof: This lemma can be found from [40].

Lemma 2. The maximum distance between any pair of vertices, u and v in Ψ, is in the range $[\sqrt{3}r_{opt}, 2r_{opt}]$.

Proof: This lemma can be found from [42].

Lemma 3. The overall time complexity of `Exact` is $O(m \times n^3)$.

Proof: Since there are three nested for-loops and finding a k-$\widehat{core}$ from a graph takes linear time cost $O(m)$ (we assume $m \geq n$) [38], the overall time complexity of `Exact` is $O(m \times n^3)$, □

Lemma 4. $\frac{1}{2}\delta \leq r_{opt} \leq \gamma$.

Proof: We have $r_{opt} \leq \gamma$ obviously, as Ψ is the optimal. We prove $\frac{1}{2}\delta \leq r_{opt}$ by contradiction. Suppose $r_{opt} < \frac{1}{2}\delta$. Since the MCC of Ψ contains q, for any $v \in \Psi$, we have $|v, q| \leq 2 \times r_{opt}$. As $r_{opt} < \frac{1}{2}\delta$, we have $|v, q| \leq 2 \times r_{opt} < \delta$. This implies that Ψ must be in a circle, whose center is q and radius is smaller than δ. This contradicts the fact that, $O(q, \delta)$ is the minimum circle with center q containing a feasible solution. Hence, Lemma 4 holds. □

Lemma 5. The radius of the MCC covering the feasible solution Φ has an approximation ratio of 2.

Proof: Let S be the set of vertices in $O(q, \delta)$. Since the vertex set of Φ is a subset of S, the MCC of Φ has a radius no larger than that of S, i.e., $\gamma \leq \delta$. By Lemma 4, we have $\frac{1}{2}\gamma \leq \frac{1}{2}\delta \leq r_{opt}$. This implies that $\frac{\gamma}{r_{opt}} \leq 2.0$. Hence, Lemma 5 holds. □

Corollary 1. If q is the center of the MCC covering Ψ, `AppInc` finds the optimal solution, i.e., Φ equals to Ψ.

Proof: This can be proved directly by contradiction. □

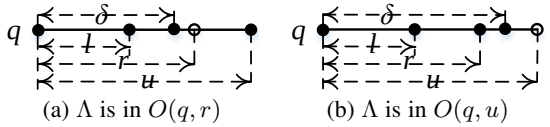
Corollary 2. The optimal solution Ψ is in $O(q, 2\gamma)$.

Proof: By Lemma 4, we have $r_{opt} \leq \gamma$. This implies that, for any $v \in \Psi$, we have $|q, v| \leq 2 \times \gamma$. Thus, all the vertices of Ψ are in $O(q, 2\gamma)$, and Corollary 2 holds. □

Lemma 6. The total time cost of `AppInc` is $O(mn)$.

Proof: In `AppInc`, the while loop is executed at most n times, and each takes $O(m)$, as computing k-$\widehat{core}$ takes $O(m)$. As a result, the total time cost of `AppInc` is $O(mn)$. □

Lemma 7. In `AppFast`, the radius of the MCC covering Λ has an approximation ratio of $(2 + \epsilon_F)$, if α is set as $\frac{r \times \epsilon_F}{2+\epsilon_F}$.



(a) Λ is in $O(q, r)$ (b) Λ is in $O(q, u)$

Fig. 19. Illustrating the proof of Lemma 7.

Proof: Consider the last loop in Algorithm 7 when returning Λ. Let the gap between the radii, which result in a feasible solution and no solution, be α.

If Λ' does exist (lines 8-10), the returned Λ is contained in $O(q, r)$. We have $l \leq \delta \leq r \leq u$ and $r - l \leq \alpha$ (see Figure 19(a)). So we have $r \leq \delta + \alpha$.

If Λ' does not exist (lines 12-13), the returned Λ is contained in $O(q, u)$. We have $l \leq r \leq \delta \leq u$ and $u - r \leq \alpha$ (see Figure 19(b)). So we have $r \leq \delta + \alpha$.

Therefore, we always have $r \leq \delta + \alpha$. We denote the radius of the MCC covering Λ by r_Λ. Considering Lemma 4, we have

$$r_\Lambda \leq r \leq 2r_{opt} + \alpha. \tag{11}$$

Eq (11) also implies that, $r_{opt} \geq \frac{1}{2}(r - \alpha)$. Then,

$$\frac{r_\Lambda}{r_{opt}} \leq \frac{2r_{opt}+\alpha}{r_{opt}} = 2 + \frac{\alpha}{r_{opt}} \leq 2 + \frac{2\alpha}{r-\alpha}. \tag{12}$$

Let $\frac{2\alpha}{r-\alpha} \leq \epsilon_F$, then we have $\frac{r_\Lambda}{r_{opt}} \leq 2 + \epsilon_F$, if α is set as $\frac{r \times \epsilon_F}{2+\epsilon_F}$. Hence, Lemma 7 holds. □

Corollary 3. The optimal solution Ψ is in $O(q, 2r_\Lambda)$, where r_Λ is the radius of the MCC containing Λ in `AppFast`.

Proof 1. Since we have $r_{opt} \leq r_\Lambda$, for any $v \in \Psi$, we have $|q, v| \leq 2 \times r_\Lambda$. Thus, all the vertices of Ψ are in $O(q, 2r_\Lambda)$, and the corollary holds.

Lemma 8. The total time cost of `AppFast` is $O(\min\{mn, m \log \frac{1}{\epsilon_F}\})$ if ϵ_F>0, or $O(mn)$ if ϵ_F=0.

Proof: In `AppFast`, the while loop needs to be executed $O(\min\{n, \log \frac{1}{\epsilon_F}\})$ times, since the number of vertices to be processed in each loop is different with that of its previous loop. Besides, each loop takes $O(m)$ time cost to find a k-$\widehat{core}$. Therefore, the total time cost of `AppFast` is $O(\min\{mn, m \log \frac{1}{\epsilon_F}\})$ if ϵ_F>0, or $O(mn)$ if ϵ_F=0. □

Lemma 9. $r_{min} \leq r_{opt} + \frac{\sqrt{2}}{2}\beta$.

Proof: We prove by contradiction. Suppose that $r_{min} > r_{opt} + \frac{\sqrt{2}}{2}\beta$. As mentioned before, we have $|o, c| \leq \frac{\sqrt{2}}{2}\beta$. For any point c' in $O(o, r_{opt})$, we have $|o, c'| \leq r_{opt}$. By triangle inequality, we conclude that $|c, c'| \leq |c, o| + |o, c'| \leq r_{opt} + \frac{\sqrt{2}}{2}\beta$. This contradicts that r_{min} is the minimum radius such that $O(c, r_{min})$ contains a feasible solution. Hence, Lemma 9 holds. □

Lemma 10. In `AppAcc`, if $r_{cur} \leq \frac{1}{2}\delta(1 + \epsilon_A)$, then we can stop searching the quadtree and return the SAC whose MCC radius r_{cur}. We can safely stop searching and guarantee the radius of the MCC covering Γ has an approximation ratio of (1+ϵ_A).

Proof: For lemma 10. Recall that in Lemma 4, $\frac{1}{2}\delta$ is a lower bound of r_{opt}, so we have $\frac{r_{cur}}{r_{opt}} \leq \frac{r_{cur}}{\frac{1}{2}\delta}$. This implies that, if $r_{cur} \leq \frac{1}{2}\delta(1 + \epsilon_A)$, we then have $\frac{r_{cur}}{r_{opt}} \leq 1 + \epsilon_A$. Hence, we can safely stop the search on the quadtree and return the SAC whose MCC radius is r_{cur} as the target SAC. □

Lemma 11. If we set $\alpha' \leq \frac{1}{4}\delta\epsilon_A$ and $\beta = \frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$ in `AppAcc`, where α' is the threshold of the gap between the upper and lower bounds of the binary search, and $0 < \epsilon_A < 1$, the radius of the MCC covering Γ has an approximation ratio of (1+ϵ_A).

Proof: Consider the binary search of an anchor point p. Let r_p be the smallest radius such that $O(p, r_p)$ contains a feasible solution. From the proof of Lemma 7, we can conclude that, $r \leq r_p + \alpha'$ when the binary search stops. Then, we have

$$\frac{r}{r_p} \leq 1 + \frac{\alpha'}{r_p} \leq 1 + \frac{\alpha'}{r_{opt}} \leq 1 + \frac{2\alpha'}{\delta}. \tag{13}$$

Let $\alpha' = \frac{1}{4}\delta\epsilon_A$. Then we have $\frac{2\alpha'}{\delta} = \frac{\epsilon_A}{2}$, and $r \leq \left(1 + \frac{\epsilon_A}{2}\right) r_p$.

Consider the updated r_{cur} after the binary search for all the anchor points. We have $r_{cur} \leq \left(1+\frac{\epsilon_A}{2}\right) r_{min}$. Let r_Γ be the radius of the MCC covering Γ. By Lemmas 4 and 9, we have

$$\frac{r_\Gamma}{r_{opt}} \leq \frac{r_{cur}}{r_{opt}} \leq 1+\frac{\epsilon_A}{2}+\frac{(2+\epsilon_A)\sqrt{2}\beta}{2\delta}. \quad (14)$$

Let $\frac{(2+\epsilon_A)\sqrt{2}\beta}{2\delta} = \frac{\epsilon_A}{2}$. Then we have $\frac{r_\Gamma}{r_{opt}} \leq 1+\epsilon_A$, if β=$\frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$. Hence, the approximation ratio of AppAcc is (1+ϵ_A), if we set the parameters α'=$\frac{1}{4}\delta\epsilon_A$ and β=$\frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$. □

Lemma 12. The total time cost of AppAcc is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\})$.

Proof: There are at most $O((\frac{2\gamma}{\beta})^2)$=$O((\frac{1}{\epsilon_A})^2)$ anchor points. Similar to that in AppFast, the binary search for each anchor point needs to be executed $O(\min\{n, \log\frac{1}{\epsilon_A}\})$ times. So the total time cost of AppAcc is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\})$. □

Lemma 13. The overall time cost of Exact+ is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\} + m|F_1|^3)$, where ϵ_A is the parameter of AppAcc and $|F_1|$ ($|F_1| \ll n$) is the number of fixed vertices left after pruning by annular regions.

Proof: Exact+ consists of two phases: (1) pruning of the fixed vertices (lines 2-5) and (2) enumeration of three vertex combinations (lines 6-16). As discussed before, Phase (1) takes $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\})$, while Phase (2) needs $O(m|F_1|^3)$. Thus, the total cost of Exact+ is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\} + m|F_1|^3)$. □

Lemma 14. If $p \in X$, $|p,q| > 2r$, and $|p',q| > 2r$, then we have Δ'=Δ. In other words, the movement of p does not change the target SAC.

Proof: It is easy to observe that before p's movement, both the optimal solution and Δ are fully contained in $O(q, 2r)$. So $|p,q| > 2r$ and $|p',q| > 2r$ imply that the movement of p is always out of $O(q, 2r)$. Consequently, both the optimal solution and Δ are unchanged. □

Lemma 15. The range of ${r_{opt}}'$ is

$$\max\left\{\frac{\delta'}{2\gamma}r_{opt}, \frac{{r_\Gamma}'}{1+\epsilon_A}\right\} \leq {r_{opt}}' \leq \min\left\{\frac{2\gamma'}{\delta}r_{opt}, {r_\Gamma}'\right\}. \quad (15)$$

Proof: By Lemma 4, we get $\frac{1}{2}\delta' \leq {r_{opt}}' \leq \gamma'$ and $\frac{1}{2}\delta \leq r_{opt} \leq \gamma$. Considering them together, we have

$$\frac{\delta'}{2\gamma}r_{opt} \leq {r_{opt}}' \leq \frac{2\gamma'}{\delta}r_{opt}. \quad (16)$$

On the other hand, since the approximation ratio of Γ' is (1+ϵ_A), we get

$$\frac{{r_\Gamma}'}{1+\epsilon_A} \leq {r_{opt}}' \leq r'_\Gamma. \quad (17)$$

Hence, Lemma 15 holds. □

Lemma 16. If δ'=δ and $\frac{2r_\Gamma}{\delta} - 1 \leq \epsilon_A \leq \frac{2\gamma'}{\delta} - 1$, then the range of r_{opt} can be further contracted as

$$\max\left\{\frac{\delta'}{2\gamma}r_{opt}, \frac{{r_\Gamma}'}{1+\epsilon_A}\right\} \leq {r_{opt}}' \leq {r_\Gamma}'. \quad (18)$$

Proof: If $\delta' = \delta$ and $\frac{2r_\Gamma}{\delta} - 1 \leq \epsilon_A$, according to previous discussions, we have ${r_\Gamma}'$ =r_Γ, $\frac{r'_\Gamma}{\frac{2\gamma'}{\delta}r_{opt}} = \frac{\delta}{2\gamma'}\cdot\frac{{r_\Gamma}'}{r_{opt}} = \frac{\delta}{2\gamma'}\cdot\frac{r_\Gamma}{r_{opt}} \leq \frac{\delta}{2\gamma'}(1+\epsilon_A)$ if $\frac{\delta}{2\gamma'}(1+\epsilon_A) \leq 1$, then $\epsilon_A \leq \frac{2\gamma'}{\delta} - 1$, so ${r_\Gamma}' \leq \frac{2\gamma'}{\delta}r_{opt}$. Additionally, for the lower bound: $\frac{\frac{{r_\Gamma}'}{1+\epsilon_A}}{\frac{\delta'}{2\gamma}r_{opt}} = \frac{1}{1+\epsilon_A}\cdot\frac{2\gamma}{\delta'}\cdot\frac{{r_\Gamma}'}{r_{opt}} \leq \frac{2\gamma}{\delta'} = \frac{2\gamma}{\delta}$, we know $1 \leq \frac{2\gamma}{\delta} \leq 2$, so we can not claim which of the lower bounds is bigger. □

Lemma 17. Let ${F_1}'$, ${F_2}'$, and ${F_3}'$ be the corresponding variables of F_1, F_2 and F_3 in Exact+ after p's movement. We have 3 cases: (a) If F_1=F'_1 and $p' \notin F_1$, we can conclude that that the movement of p will not affect F_1. In other words, we have Ψ'=Ψ. (b) If F_1=F'_1 and $p' \in F_1$, and $p' \notin R_1$, then we only need to enumerate all the combinations of three vertices, where one is p' and the other two are from ${F_2}'$ and ${F_3}'$ respectively, to find Ψ'. (c) For other cases that cannot be handled by cases 1 and 2, we need to enumerate all the combinations of three vertices from ${F_1}'$, ${F_2}'$, and F'_3 respectively to compute Ψ'.

Proof: The lemma directly follows the observation. □

9 Pseudocodes of SAC Search Algorithms

Algorithm 5 shows Exact. It first finds a list X of vertices of the k-$\widehat{core}$, and sorts them according to their distances from q in ascending order (lines 2-3). Note X_i denotes i-th vertex. For each three vertex combination, it verifies whether there is a k-$\widehat{core}$ in the MCC fixed by it, and finally returns Ψ (lines 4-14).

Algorithm 5 Query algorithm: Exact

```
1: function EXACT(G, q, k)
2:     find the vertex list X of the k-core^ containing q;
3:     sort vertices of X;
4:     initialize r ← +∞, Ψ ← ∅;
5:     for i ← 3 to |X| do
6:         for j ← 1 to i–2 do
7:             for h ← j + 1 to i–1 do
8:                 compute the MCC mcc of {X_i, X_j, X_h};
9:                 if mcc.radius < r then
10:                    R ← a set of vertices in mcc;
11:                    if exist a k-core^ with q in G[R] then
12:                        r ← mcc.radius, Ψ ← this k-core^;
13:        if |q, X_i| > 2r then break;
14:    return Ψ;
```

Algorithm 6 presents AppInc. First, it initializes four variables $Queue$, S, T and Φ: $Queue$ is a priority queue of vertices, in which vertices are sorted in an ascending order according to their distances to q; S is the set maintaining vertices close to q incrementally; T is a set for recording vertices added to $Queue$; and Φ is the approximated SAC. Then, it adds q to $Queue$ in the beginning (line 3). In the while loop (lines 4-15), it first gets the nearest vertex, p, from $Queue$, and adds it to S (lines 5-6). Next, it considers q's neighbors (lines 7-12). For each neighbor $v \in X$, if it is in $O(q, |p,q|)$, we add it to S directly; otherwise, we put it into $Queue$ as it is already in $O(q, |p,q|)$. Note that in any feasible solution, each vertex has at least k neighbors. So if both p and q have at least k neighbors in S, it checks whether there exists an SAC in $G[S]$. If it exists, then AppInc returns it (lines 13-16).

Algorithm 7 presents AppFast. We denote the SAC returned by AppFast by Λ. ϵ_F is an input parameter. By following the two-step framework, it first computes the k-$\widehat{core}$ (line 2), and then finds Λ from the k-$\widehat{core}$ (lines 3-14). Some variables such as Λ, l and u are initialized (line 3). In while loop (lines 4-14), it first finds an SAC Λ' from $O(q, r)$ using breadth first search (BFS). If Λ' does exist, it first updates Λ, since this solution has a smaller radius. It then checks whether the gap, i.e., $r - l$, is smaller than α (we will discuss how to set this gap later). If it is not larger than α,

Algorithm 6 Query algorithm: `AppInc`

1: **function** APPINC(G, q, k)
2: initialize $Queue$, $S \leftarrow \emptyset$, $T \leftarrow \emptyset$, $\Phi \leftarrow \emptyset$;
3: $Queue$.add(q);
4: **while** $|Queue| > 0$ **do**
5: $p \leftarrow Queue$.poll();
6: S.add(p);
7: **for** $v \in nb(p)$ **do**
8: **if** $deg_G(v) \geq k$ **then**
9: **if** $|v, q| \leq |p, q|$ **then**
10: S.add(v);
11: **else if** $v \notin T$ **then**
12: $Queue$.add(v); T.add(v);
13: **if** $|S \cap nb(q)| \geq k \wedge |S \cap nb(p)| \geq k$ **then**
14: **if** exist a k-$\widehat{core}$ containing q in $G[S]$ **then**
15: $\Phi \leftarrow$ this k-$\widehat{core}$; break; //stop
16: **return** Φ;

Algorithm 7 Query algorithm: `AppFast`

1: **function** APPFAST(G, q, k, ϵ_F)
2: find the vertex list X of the k-$\widehat{core}$, Λ, containing q;
3: initialize l, u using Eq (1);
4: **while** $u > l$ **do**
5: $r \leftarrow \frac{l+u}{2}$;
6: $S \leftarrow$ vertices in $O(q, r)$;
7: $\Lambda' \leftarrow$ the k-$\widehat{core}$ containing q in $G[S]$;
8: **if** $\Lambda' \neq \emptyset$ **then**
9: $\Lambda \leftarrow \Lambda'$;
10: **if** $r - l \leq \alpha$ **then return** Λ;
11: $u \leftarrow \max_{v \in \Lambda} |q, v|$;
12: **else**
13: **if** $u - r \leq \alpha$ **then return** Λ;
14: $l \leftarrow \min_{v \in \Lambda \wedge v \notin S} |q, v|$;

then it returns Λ; otherwise, it updates u as the maximum distance from q to vertices in Λ, which ensures that the feasible solution found later has at least one less vertex than Λ. If Λ' does not exist, it returns Λ if the gap, i.e., $u - r$, is small enough; otherwise, it updates l as the minimum distance from q to vertices in Λ, but not in S, which ensures that the set S in the next iteration has at least one more vertex than current S.

Algorithm 8 presents the steps of `AppAcc`. ϵ_A is an input parameter. It first performs `AppFast` (ϵ_F=0), and obtains the k-$\widehat{core}$ in $O(q, 2\gamma)$ (lines 2-3), which contains Ψ by Corollary 2. Then it initializes four variables: Γ is the target SAC, β equals to γ, r_{cur} is the radius of the smallest MCC covering a feasible solution, and $achList$ contains the center points of four child nodes of the root node (line 4). In the while loop (lines 5-27), we consider nodes in the region quadtree level by level in a top-down manner. Specifically, for each point $p \in achList$, we first check whether it can be pruned using *Pruning1* (line 8), and then use binary search introduced in `AppFast` to find a feasible solution (lines 12-22), and finally update r_{cur} and Γ, if the radius of the MCC covering the feasible solution is smaller than r_{cur}. After considering nodes in this level, we use *Pruning2* to prune some nodes. Additionally, we also prune some anchors whose corresponding SACs are the same according to the *Pruning3* (line 25). Note that map keeps $<key, value>$ pairs, where key is a center point and $value$ denotes the radius that results in no feasible solution. Next, we update β and collect all the child nodes needed to be considered in the next level (lines 26-27). The loop is executed until β is smaller than the threshold $\frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$ (we will show how to set β later). Finally, Γ is returned (line 28).

Algorithm 9 presents the pseudocodes of `Exact+`. It first

Algorithm 8 Query algorithm: `AppAcc`

1: **function** APPACC(G, q, k, ϵ_A)
2: obtain Φ, δ and γ using `AppFast`;
3: $S \leftarrow$ vertices of the k-$\widehat{core}$, containing q, in $O(q, 2\gamma)$;
4: $\Gamma \leftarrow \Phi$, $\beta \leftarrow \gamma$, $r_{cur} \leftarrow \gamma$, $achList \leftarrow$ center points;
5: **while** $\beta \geq \frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$ **and** $r_{cur} > \frac{1}{2}\delta(1+\epsilon_A)$ **do**
6: $map \leftarrow \emptyset$;
7: **for** each point $p \in achList$ **do**
8: **if** $|p, q| \leq r_{cur} + \frac{\sqrt{2}}{2}\beta$ **then** //*Pruning1*
9: $\Gamma_p \leftarrow$ find an SAC in $O(p, r_{cur} + \frac{\sqrt{2}}{2}\beta)$;
10: **if** $\Gamma_p \neq \emptyset$ **then**
11: $u \leftarrow r_{cur} + \frac{\sqrt{2}}{2}\beta$, $l \leftarrow \frac{\delta}{2}$, map.put(p, l);
12: **while** $u \geq l$ **do**
13: $r \leftarrow \frac{l+u}{2}$;
14: $\Gamma'_p \leftarrow$ find an SAC in $O(p, r)$;
15: **if** $\Gamma'_p \neq \emptyset$ **then**
16: $\Gamma_p \leftarrow \Gamma'_p$;
17: **if** $r - l \leq \alpha'$ **then** break;
18: $u \leftarrow \max_{v \in \Gamma_q} |q, v|$;
19: **else**
20: map.put(p, l);
21: **if** $u - r \leq \alpha'$ **then** break;
22: $l \leftarrow \min_{v \in S \wedge v \notin O(p,r)} |q, v|$;
23: $r \leftarrow$ radius of the MCC covering Γ_p;
24: **if** $r < r_{cur}$ **then** $r_{cur} \leftarrow r$; $\Gamma \leftarrow \Gamma_p$;
25: prune anchor points in map using *Pruning2,3*;
26: $\beta \leftarrow \beta/2$;
27: update anchor point list $achList$ using map;
28: **return** Γ;

Algorithm 9 Query algorithm: `Exact+`

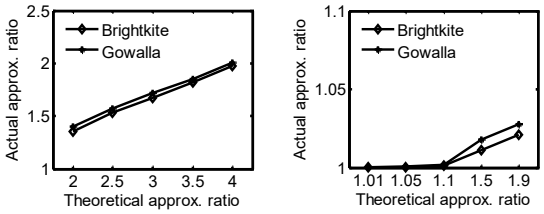
1: **function** EXACT+(G, q, k, ϵ_A)
2: run APPACC(G, q, k, ϵ_A);
3: $T \leftarrow$ anchor points in map of `AppAcc`;
4: initialize $F_1 \leftarrow \emptyset$;
5: **for** $p \in T$ **do** F_1.add$\{v | r_- \leq |p, v| \leq r_+ \ \wedge v \in S\}$;
6: **for** $v_1 \in F_1$ **do**
7: $F_2 \leftarrow \{v | \sqrt{3}r_- \leq |v_1, v| \leq 2r_{cur} \ \wedge v \in F_1\}$;
8: **for** $v_2 \in F_2$ **do**
9: $F_3 \leftarrow \{v | |v_1, v| \leq |v_1, v_2| \ \wedge v \in F_1\}$;
10: **for** $v_3 \in F_3$ **do**
11: compute the MCC mcc of $\{v_1, v_2, v_3\}$;
12: **if** $mcc.radius < r_{cur}$ **then**
13: $R \leftarrow$ a set of vertices in mcc;
14: **if** exist a k-$\widehat{core}$ in $G[R]$ **then**
15: $r_{cur} \leftarrow mcc.radius$;
16: $\Psi \leftarrow$ this k-$\widehat{core}$;
17: **return** Ψ;

runs `AppAcc` with a small value of ϵ_A (line 2). Note that Ψ is initialized as Γ, and S and r_{cur} are updated by `AppAcc`. Then, it collects a set, T, of anchor points that are not pruned in the last while loop of `AppAcc` (line 3). Finally, an empty set F_1 is initialized (line 4). For each anchor point p, it finds the potential fixed vertices by Eqs (3) and (4), and adds them into F_1 (line 5).

Next, it considers the three vertex combinations. It considers each vertex $v_1 \in F_1$ as a fixed vertex of an MCC, and its farthest fixed vertex v_2 for this MCC. By Lemma 2, we have $|v_1, v_2| \in [\sqrt{3}r_{opt}, 2r_{opt}]$. So a set F_2 of potential farthest fixed vertices is collected (line 7). Next, it collects a set, F_3, of the third fixed vertices (line 9). Finally, it computes the MCC fixed by three vertices from F_1, F_2 and F_3 respectively, keeps the SAC with the smallest MCC radius (lines 11-16), and returns it (line 17). Note that r_- and r_+ are also updated during the enumeration.

10 Approximation Ratio of SAC Search Algorithms

In this section, we focus on the approximation algorithms of SAC search and compare their theoretical approximation ratios and actual approximation ratios. In Figure 20, we report the theoretical and actual approximation ratios of AppFast and AppAcc on Brightkite and Gowalla datasets. Note that if we set ϵ_F=0.0, the results of AppFast are the same with those of AppInc, so we do not report results of AppInc. We can see that, the actual approximation ratios of AppFast and AppAcc are much smaller than the theoretical approximation ratios. For example, when ϵ_F=2, the theoretical approximation ratio of AppFast is 4.0, but its actual approximation ratios are around 2.0 on these two datasets. Similar results can be observed from AppAcc in Figure 20(b).



(a) AppFast (b) AppAcc

Fig. 20. Approximation ratio.

11 More Experimental Results of SAC Search and CSAC Search

The efficiency results of SAC search on Gowalla, Syn1, and Syn2 datasets are reported in Figure 21. We can easily observe that, the results on synthetic datasets are generally highly similar to those of real datasets. From these efficiency results, again we can conclude that, for moderate-size graphs, Exact+ is the best choice, as it achieves the highest quality with reasonable efficiency; for large graphs with millions of vertices or edges, AppFast and AppAcc are better options as they are much faster than Exact+.

The efficiency results of CSAC search on Gowalla dataset are reported in Figure 22. Clearly, the experimental results are generally similar to those on Brightkite and Foursquare datasets.

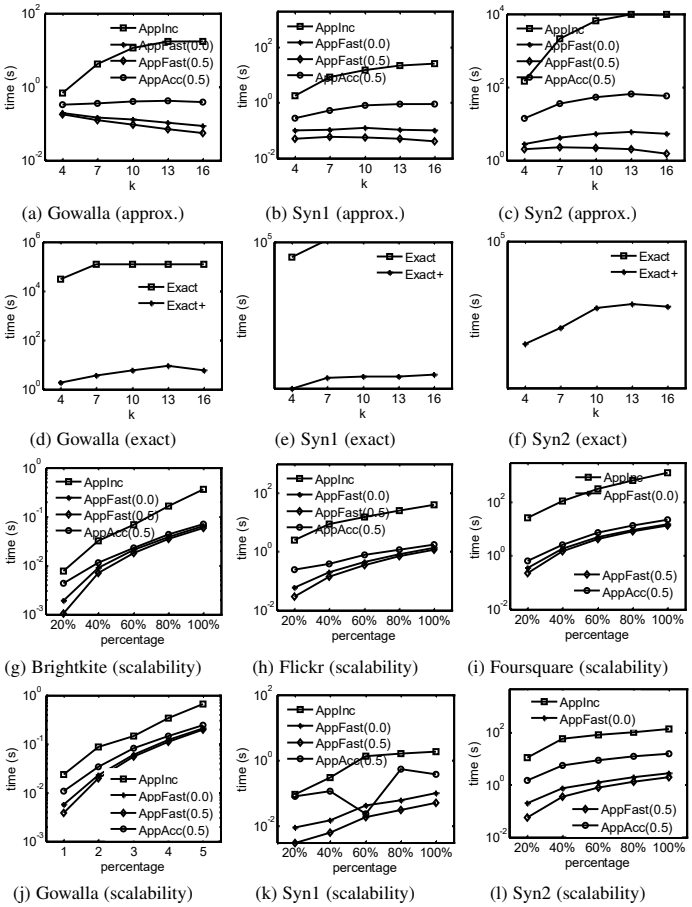
12 Results on Noisy Graphs

TABLE 6
New datasets used in our experiments.

Name	Vertices	Edges	$\widehat{d}$	BC	CC
Oregon	10,670	22,002	4.12	2.44E-5	0.2969
Enron	36,692	367,662	20.04	1.69E-4	0.3076

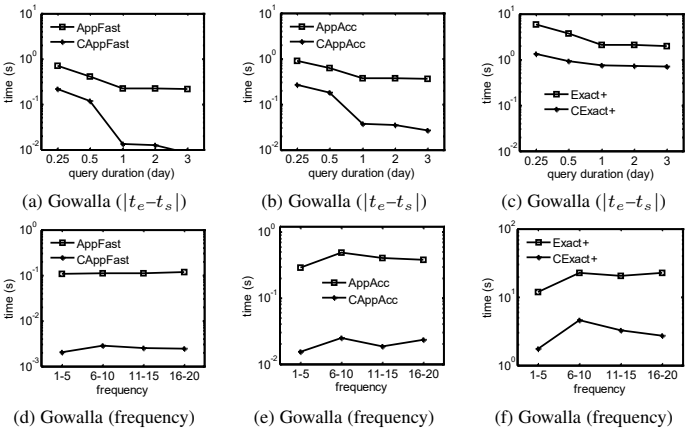
In this section, we evaluate the performance of our algorithms on networks with anomalous vertices. However, to our best knowledge, there is no public spatial graph, where anomalous vertices are explicitly labelled. Fortunately, there are some works [47], [48] that can identify anomalous vertices though, and they have performed studies on some public datasets. For example, in [47], a vertex is considered as an anomalous vertex, if its neighbors are almost fully connected (one extreme case called "*near-clique*" [47], [48]) or rarely connected (the other extreme case called "*near-star*" [47], [48]). if it is well connected with its neighbors (near-cliques) or not connected (stars). This is because, in most social networks, friends of friends are often friends, which implies that a vertex v's neighbors are often linked, but the extreme cases "*near-clique*" (any two neighbors of vertex v are linked) and "*near-star*" (any two neighbors of vertex v are not linked), are suspicious.

In our experiments, we study two networks (i.e., Oregon and Enron), which are used in [47] but not in our original submission. The statistics of these two datasets are reported in Table 6. Note that these two networks are not spatial graphs, so we need to



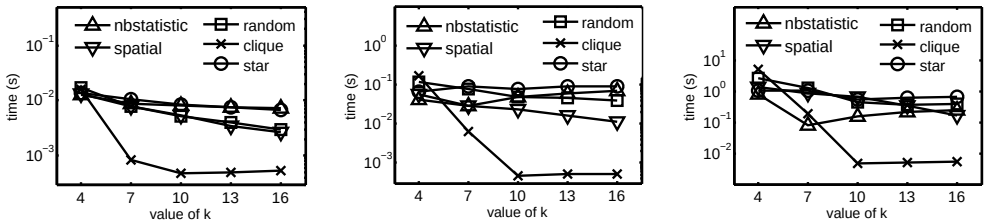
(a) Gowalla (approx.) (b) Syn1 (approx.) (c) Syn2 (approx.)
(d) Gowalla (exact) (e) Syn1 (exact) (f) Syn2 (exact)
(g) Brightkite (scalability) (h) Flickr (scalability) (i) Foursquare (scalability)
(j) Gowalla (scalability) (k) Syn1 (scalability) (l) Syn2 (scalability)

Fig. 21. Efficiency of SAC search on Gowalla, Syn1, and Syn2 datasets.



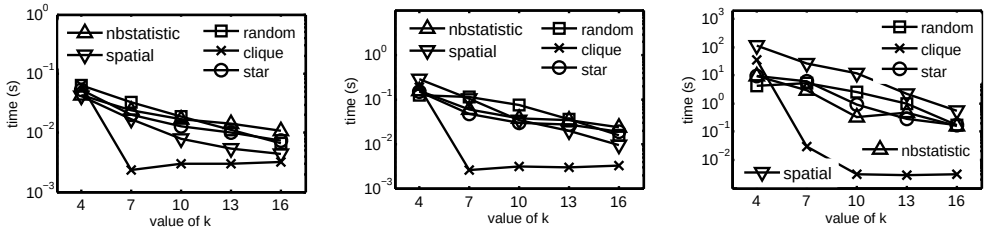
(a) Gowalla ($|t_e - t_s|$) (b) Gowalla ($|t_e - t_s|$) (c) Gowalla ($|t_e - t_s|$)
(d) Gowalla (frequency) (e) Gowalla (frequency) (f) Gowalla (frequency)

Fig. 22. Efficiency of CSAC search on Gowalla dataset.



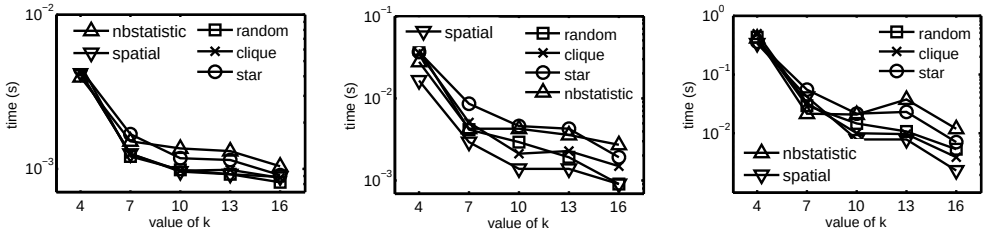
(a) AppFast (b) AppAcc (c) Exact+

Fig. 23. Efficiency on the Enron dataset.



(a) AppFast (b) AppAcc (c) Exact+

Fig. 24. Efficiency on the Oregon dataset.



(a) AppFast (b) AppAcc (c) Exact+

Fig. 25. Efficiency on the BrightKite dataset.

associate locations to their vertices. Specifically, we assume that all the vertices are in the [0, 1]^2 space, and let n be the number of vertices in the graph. We randomly divide all the vertices into two groups of sizes (n–100) and 100, with the aim of making the 100 vertices in the second group spatially anomalous. For vertices in the first group, we assume that their locations follow a normal distribution in each dimension, i.e., $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$, which correspond to locations in the x- and y-axes respectively. For vertices in the second group, their locations are generated, such that they are significantly deviated from the distributions of the locations of vertices in the first group, i.e., they are in the space [0, 1]^2, but not in the "3-standard deviation" [48] area [μ_1-3δ_1, μ_1+3δ_1]×[μ_2-3δ_2, μ_2+3δ_2]. In other words, the vertices of the second group can be considered as spatially anomalous vertices.

To evaluate the performance, for each dataset, we select five sets of query vertices, each of which has 100 vertices in the 4-core of the graph, as follows:

Set-1: Vertices in this set are randomly selected;

Set-2: We use the algorithms in [47] to detect the top-100 vertices, which most-likely tend to form near-cliques with their neighbors;

Set-3: We use the algorithms in [47] to detect the top-100 vertices, which most-likely tend to form near-stars with their neighbors;

Set-4: Vertices in this set are anomalous vertices, which are detected by the algorithm in [48] based on the statistics of vertices' neighbors;

Set-5: This set contains all the vertices from the second group in location generation as discussed above.

Notice that the algorithms in [47], [48] mainly detect vertices that are structurally anomalous. Therefore, vertices in Set-2, Set-3, and Set-4 are *structurally* anomalous. Vertices in Set-5 are *spatially* anomalous, since their locations significantly deviate from those of others as discussed above. In summary, there are four sets of anomalous query vertices.

In addition, we consider another real dataset, i.e., Brightkite (see Section 6.1), and also select five sets of query vertices. Set-1, Set-2, Set-3, and Set-4 are selected respectively using the same methods above. For Set-5, to select the spatially anomalous vertices, we assume that all the vertices follow a normal distribution in each dimension, i.e., $N(\mu_x, \sigma_x^2)$ and $N(\mu_y, \sigma_y^2)$, which correspond to locations in x-axis and y-axis respectively. Then, we randomly select 100 vertices in the space [0, 1]^2, but not in the "3-standard deviation" [48] area [μ_x-3δ_x, μ_x+3δ_x]×[μ_y-3δ_y, μ_y+3δ_y].

We run `AppFast(0.5)`, `AppAcc(0.5)`, and `Exact+` with the five query vertex sets on each dataset. Note that, we do not run `AppInc` and `Exact`, because they are much slower than `AppFast` and `Exact+`, as shown by our previous experimental studies. The experimental results on the Enron, Oregon, and Brightkite datasets are reported in Figures 23, 24, and 25 respectively, where "random", "clique", "star", "nbstatistics", and "spatial" denote the results of query vertices in Set-1, Set-2, Set-3, Set-4, and Set-5 respectively.

We can observe that, the efficiency trends on the Brightkite dataset and two new datasets (i.e., Enron and Oregen), which

contain anomalous vertices as shown in [47], are very similar, i.e., the running time of AppFast, AppAcc, and Exact+ decreases as the value of k increases (the detailed reasons are discussed in the earlier version of this paper [45]). However, for different anomalous query vertices, their running time costs are different. Specifically, we have some interesting observations as follows:

Set-2: Vertices in Set-2 tend to achieve higher efficiency than those in Set-1. For example, on Enron and Oregon datasets, all the algorithms perform the fastest when using vertices of Set-1. The main reason is that the query vertices are near-clique vertices, i.e., they tend to form cliques with their neighbors, so their SACs are in small subgraphs and the queries can be performed efficiently.

Set-3: Vertices in Set-3 have lower efficiency than those in Set-1. This is because, the query vertices are near-star vertices, i.e., their neighbors are rarely connected with each other. This means that, to find an SAC for a query vertex q, we may have to consider vertices which are structurally far away from q. As a result, more time cost is needed.

Set-4: The performance of vertices in Set-4 fluctuates differently on different datasets. We conjecture that this is because, in [48], a vertex is identified as an anomalous vertex, if its neighbor statistics highly deviate from those of other vertices, so many kinds of anomalous vertices are selected, resulting in different efficiency results.

Set-5: The performance of vertices in Set-5 is different with those in Set-1, but it fluctuates differently on different datasets. For example, on the Oregon dataset, AppFast performs very fast using vertices of Set-5, but Exact+ runs slowly for vertices in Set-5. This is because, although these vertices are spatially anomalous, some of them may still be structurally normal, so they may achieve normal efficiency.

In summary, for anomalous networks, our algorithms generally achieve similar efficiency trends with those on the real dataset (i.e., Brightkite) that we have used before, but for different sets of anomalous vertices, the efficiency results are different as discussed above.

13 Effect of Network Characteristics

In this section, we consider two typical network characteristics, namely *betweenness centrality (BC)* and *clustering coefficient (CC)*, and study their effects on the efficiency of SAC search.

• The **betweenness centrality (BC)** is a measure based on shortest paths. Essentially, it represents the degree of which vertices stand between each other. For example, in a flow network, a vertex with higher betweenness centrality would have more control over the network, because more information will pass through that vertex. The betweenness centrality value of a vertex v in the graph G is defined as

$$g(v) = \sum_{v \neq s, v \neq t} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \tag{19}$$

, where $\sigma_{s,t}$ is the total number of shortest paths between any pair of vertices s and t, and $\sigma_{s,t}(v)$ is the number of those paths that pass through v.

• The **clustering coefficient (CC)** of a vertex in a graph measures how close its neighbors are to being a clique (complete graph). In other words, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. The clustering coefficient of a vertex v of the graph $G(V, E)$ is defined as

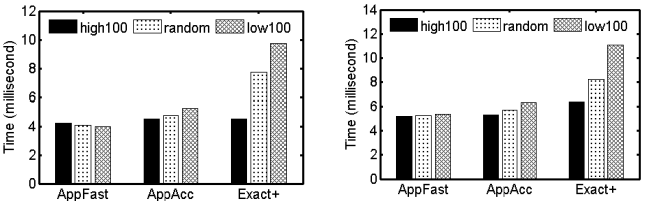
$$h(v) = \frac{|\{(v_i, v_j) : v_i, v_j \in nb(v), (v_i, v_j) \in E\}|}{|nb(v)|(|nb(v) - 1|)} \tag{20}$$

, where $nb(v)$ denotes the set of v's neighbors in G.

In Table 4, the average values of BC and CC in each dataset are reported. We can observe that for larger datasets, the values of BC tend to be smaller, because in a larger graph, a vertex tends to be passed through by less shortest paths. While for CC, the values are dataset dependent, since the CC value of a vertex mainly depends on its neighbors. By observing the efficiency of SAC search on different datasets, we can conclude that generally the running time increases with the dataset size. In addition, the SAC search algorithms tend to be slower for datasets whose average BC values are lower, while for CC values, we cannot obtain similar observation.

To further examine the detailed effect of these characteristics on the efficiency, we perform the following experimental study. Specifically, for each dataset, we first compute the BC value of each vertex, and then select three groups of query vertices with each having 100 vertices. The first group contains 100 svertice whose BC values are the highest, the second group contains 100 vertice which are randomly selected, and the last group contains 100 vertices whose BC values are the lowest. We denote these three groups by "high100", "random", and "low100" respectively. Finally, we run our SAC search algorithms, i.e., AppFast(0.5), AppAccf(0.5), and Exact+, for each group, and record the running time. Note that, we do not run AppInc and Exact, because they are much slower than AppFast and Exact+, as shown by our previous experimental studies. In addition, for each dataset, we perform similar steps for vertices with different CC values as above.

The efficiency results are reported in Figures 26 and 27. Interestingly, we can observe that, for AppFast, the efficiency gaps among these groups of queries are very small; while for AppAcc and Exact+, vertices whose BC and CC values are higher, they tend to perform faster than vertices whose BC and CC values are lower. This is mainly because AppFast is purely based on binary search and the number of binary search times is often limited. While for other algorithms, after running AppFast, they still need to perform graph search. Since for vertices whose BC and CC values are higher, they tend to have more connection with others. As a result, it is easier for them to find communities, resulting in higher query efficiency.

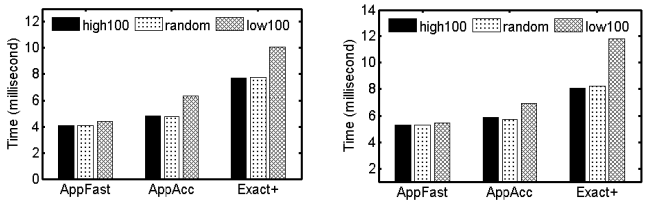


(a) Brightkite (b) Gowalla

Fig. 26. Effect of betweenness centrality.

14 Effect of Noise

In this section, we study the behaviors of SAC search algorithms on graphs, which are associated with noise. To add noise into the graph G, we create some new vertices with locations, and also link them to the vertices of G (details to follow). Before generating the noise, we collect some statistics of the graph G as follows. (1) We compute the probability distribution of numbers of neighbors for the vertices. Specifically, we obtain a degree distribution



(a) Brightkite (b) Gowalla

Fig. 27. Effect of clustering coefficient.

function $p(x)$, which means that the probability that a vertex v has x neighbors is $p(x)$. (2) We assume that the locations of these vertices follow a normal distribution along each dimension, and thus compute two normal distributions, i.e., $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$, which correspond to vertices' locations in the x-axis and y-axis respectively. Based on these statistics, we add noise vertices into G by following the three cases below respectively.

• **Case 1:** Their numbers of neighbors follow the distribution $p(x)$ of the graph G, and their locations follow the distribution of the locations of vertices in the graph G.

• **Case 2:** Their locations follow the distribution of locations of vertices in the graph G, but their numbers of neighbors greatly deviate from the distribution $p(x)$. To be specific, let the range of x's values be $[a, b]$. Then, we only consider the x values close to a or b, particularly in the ranges $[a, a + (b - a) \times 10\%]$ and $[b-(b-a)\times 10\%, b]$, and their numbers of neighbors are randomly selected from these two sub-ranges.

• **Case 3:** Their numbers of neighbors follow the distribution $p(x)$ of the graph G, but their locations greatly deviate from the original distribution, i.e., they are selected from $[0, 1]^2$ but not in the "3-standard deviation" [48] area $[\mu_1 - 3\sigma_1, \mu_1 + 3\sigma_1] \times [\mu_2 - 3\sigma_2, \mu_2 + 3\sigma_2]$.

We consider two real datasets, i.e., Brightkite and Gowalla, and add noise vertices into these graphs following three cases above respectively. Note that in all cases above, if a noise vertex v has x neighbors, then we randomly select x vertices from G and link them to v. Moreover, for each case, we vary the numbers of noise vertices as $n\times 0\%$, $n\times 3\%$, $n\times 5\%$, $n\times 10\%$, and $n\times 15\%$ respectively, where n is the number of vertices in G. After that, we run three algorithms (i.e., `AppFast(0.5)`, `AppAcc(0.5)`, and `Exact+`) using the query vertices (which are introduced in Section 6.1), and then report their values on the following two measures:

• **Efficiency**. The average running time of an SAC query.

• **Similarity**. Let $C(q)$ be the set of vertices in the SAC of q in the graph G, and $C(q)'$ be the set of vertices in the SAC of q in the noisy graph. Then, we compute the Jaccard similarity between $C(q)$ and $C(q)'$, and report the average similarity value.

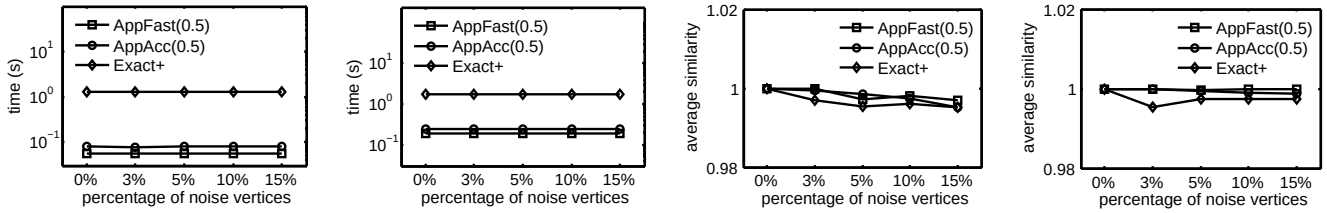
We report the experimental results for the three cases in Figures 28, 29, and 30 respectively. From these figures, we can easily observe that, in each case, when the number of noise vertices varies from $n\times 0\%$ to $n\times 15\%$, the running time increases very slightly and the similarity decreases also very slightly. For example, on the Gowalla dataset, the running time increases by only 1% in the first case. Moreover, the effect of noise on the efficiency and similarity is similar in all these three cases. This is mainly because, in the first step of all the three algorithms, they use the binary search to quickly reduce the search space from the whole data space to $O(q, \delta)$. Since $O(q, \delta)$ is much smaller than the whole data space, the number of noise vertices it contains is very limited. Consequently, the noise vertices have little or no effect on the overall efficiency and members of the SAC. Therefore, our algorithms are robust to the noise vertices.

15 Sensitivity Test of Parameters of SAC Search Algorithms on Noisy Network

In this section, we experimentally evaluate the sensitivity of the choice of parameters on graphs with noise. We perform experiments on the graphs with noise vertices, which are introduced by the three cases in Section 14. The results show that, our algorithms generally achieve similar sensitivity as that on the original datasets.

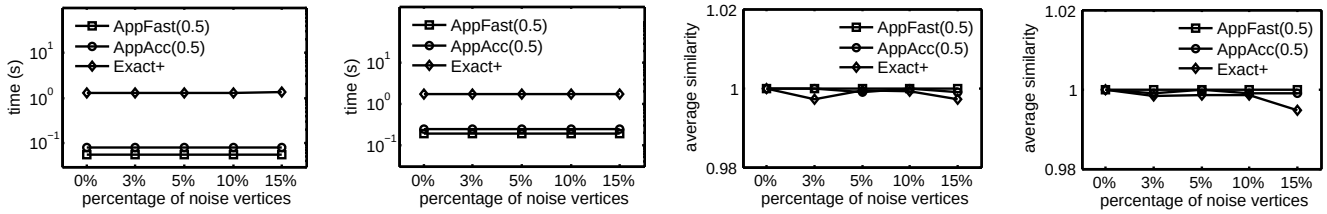
Specifically, we consider three parameters, which are ϵ_F in `AppFast`, ϵ_A in `AppAcc`, and ϵ_A in `Exact+`. We use the graphs with noise vertices, which are introduced by the three cases in Section 14. For each graph in each case, we add $n\times 10\%$ noise vertices, where n is the number of vertices in the original graph (which are not added noisy vertices). After that, we vary the values of these parameters and run `AppFast`, `AppAcc`, and `Exact+` on the noisy datasets. For comparison, we also run these algorithms on the original graphs.

In Figures 31, 32, and 33, we report the efficiency results on the Brightkite and Gowalla datasets, and their three noisy versions of datasets respectively. Note that "Brightkite*" and "Gowalla*" denote the noisy versions of the Brightkite and Gowalla datasets respectively. As can be seen, for each dataset in each case, the gap of efficiency on each original graph and its noisy graph is very small. For example, in Case 1, `AppFast` achieves almost the same efficiency on the graphs Gowalla and Gowalla*. In addition, for `Exact+`, it always achieves the highest efficiency when ϵ_A is set to 10^{-4}, and this result is consistent with our previous result of parameter sensitivity test. Thus, our algorithms are not very sensitive to the newly added noise, and thus are very robust.



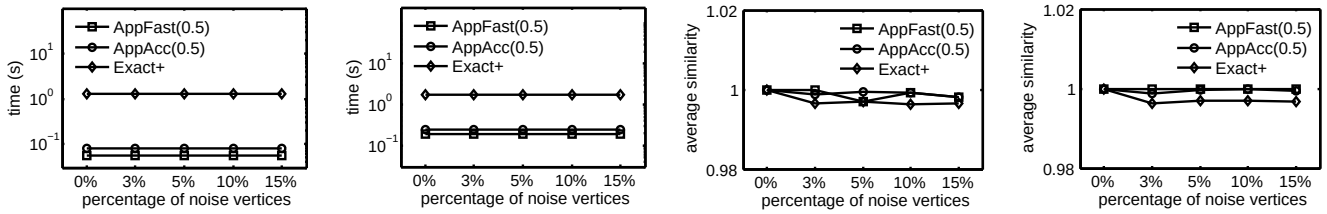
(a) Brightkite (Efficiency) (b) Gowalla (Efficiency) (c) Brightkite (Similarity) (d) Gowalla (Similarity)

Fig. 28. Effect of noise in Case 1.



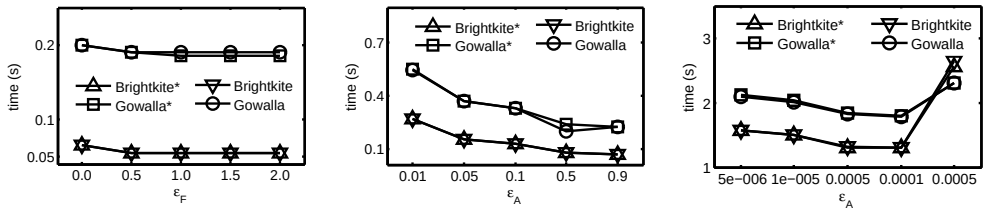
(a) Brightkite (Efficiency) (b) Gowalla (Efficiency) (c) Brightkite (Similarity) (d) Gowalla (Similarity)

Fig. 29. Effect of noise in Case 2.



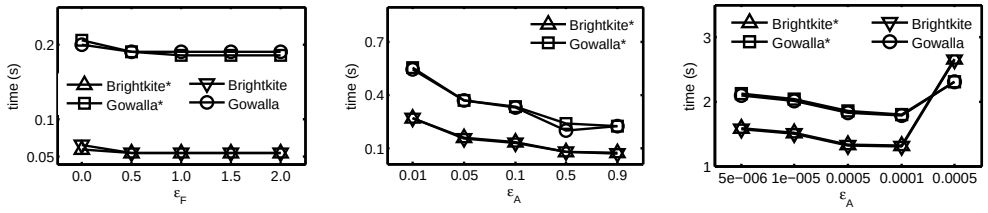
(a) Brightkite (Efficiency) (b) Gowalla (Efficiency) (c) Brightkite (Similarity) (d) Gowalla (Similarity)

Fig. 30. Effect of noise in Case 3.



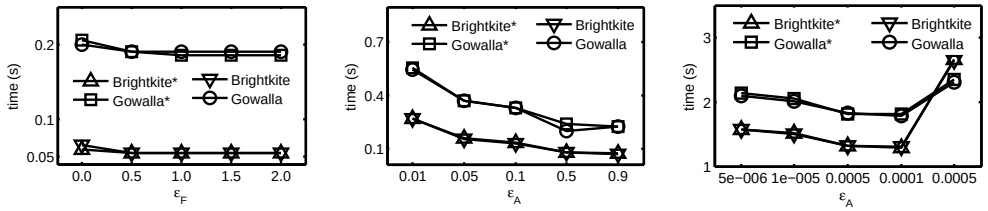
(a) ϵ_F in `AppFast` (b) ϵ_A in `AppAcc` (c) ϵ_A in `Exact+`

Fig. 31. Effect of parameters on noise network in Case 1.



(a) ϵ_F in `AppFast` (b) ϵ_A in `AppAcc` (c) ϵ_A in `Exact+`

Fig. 32. Effect of parameters on noise network in Case 2.



(a) ϵ_F in `AppFast` (b) ϵ_A in `AppAcc` (c) ϵ_A in `Exact+`

Fig. 33. Effect of parameters on noise network in Case 3.