# Effective Community Search over Large Spatial Graphs

Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu
Department of Computer Science, The University of Hong Kong, Hong Kong
{yxfang, ckcheng, xdli, sqluo, jhu}@cs.hku.hk

## ABSTRACT

Communities are prevalent in social networks, knowledge graphs, and biological networks. Recently, the topic of community search (CS) has received plenty of attention. Given a query vertex, CS looks for a dense subgraph that contains it. 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 the creation of 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. In this situation, 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 exact solutions for finding an SAC that contains $q$. Since these solutions cannot scale to large datasets, we have further designed three approximation algorithms to compute an SAC. We have performed an experimental evaluation for these solutions on both large real and synthetic datasets. Experimental results show that SAC is better than the communities returned by existing solutions. Moreover, our approximation solutions can find SACs accurately and efficiently.

## 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, 30, 26, 12]. 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.

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 [22]. 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.

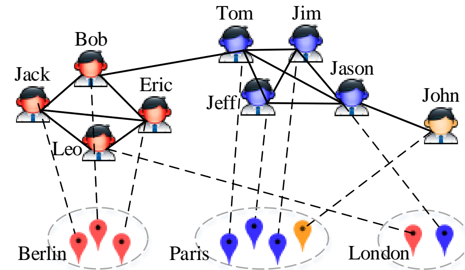


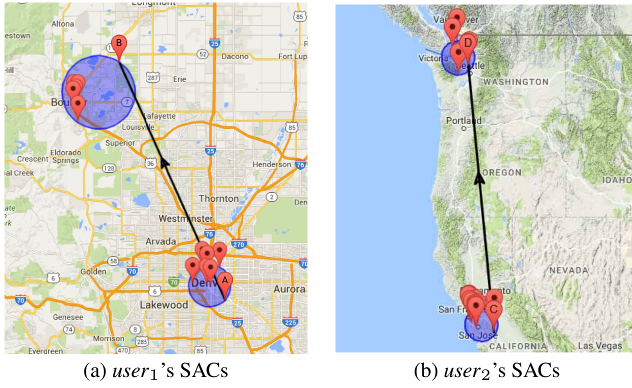

**Figure 1: A geo-social network.**

**Table 1: Works on community retrieval.**

| Graph Type | Community Detection (CD) | Community Search (CS) |
|---|---|---|
| Non-spatial | [25, 14] | [29, 7, 6, 21, 19, 11] |
| Spatial | [16, 10, 4] | **SAC search** |

**Prior works.** The community retrieval methods can generally be classified into *community detection* (CD) and *community search* (CS), as shown in Table 1. Earlier CD methods [25, 14] mainly focus on link analysis without considering spatial features. Some recent studies [2] 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, 10, 4] have considered the spatial features for community detection. 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 community search. Also, their efficiency is inadequate for fast and online community retrieval since they require to enumerate all the communities. To address these limitations, some works [29, 7, 6, 19, 11] focus on *online* community search, a query-dependent variant of community detection, 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 [29, 7] 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
*Proceedings of the VLDB Endowment,* Vol. 10, No. 6
Copyright 2017 VLDB Endowment 2150-8097/17/02.




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

**Figure 2: SACs in Brightkite dataset.**

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 [29] and [7] 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 the commonly used metric *minimum degree* [29, 7, 21] to measure the structure cohesiveness. Note that in our method, the minimum degree metric can be easily replaced by other metrics like $k$-truss [19] and $k$-clique [6]. 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, 10, 4] 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*[1], *Meetin*[2], and *Eventbrite*[3] 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 [30]. 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 [23], 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 [5], these features are often related to the people located there. For example, Silicon Valley can be characterized by "information technology" because many residents/workers 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 [27] 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 $\epsilon_F$ 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

[1]https://www.meetup.com/
[2]https://www.meetin.org/
[3]https://www.eventbrite.hk/

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.

We have implemented our algorithms and performed extensive experiments on four real datasets and two synthetic datasets. We develop several metrics to measure the quality of a community, considering the spatial circles and distances among community members, and compare existing CD and CS methods under these metrics. These results confirm the superiority of SAC search. In addition, we also have run experiments on a dynamic spatial graph, where users' locations change frequently, and the results show that SAC search can well adapt to location changes.

We further evaluate the efficiency of SAC search, and the results show that the developed algorithms are more efficient than the baseline algorithms. From extensive experiments, we conclude that, for moderate-size graphs, Exact+ is the best choice, as it achieves the highest quality with reasonable efficiency, while for large graphs with millions of vertices, AppFast and AppAcc are better choices as they are much faster than Exact+.

**Organization.** We review the related work in Section 2. We formally define the problem studied in this paper in Section 3. Section 4 presents the proposed query algorithms. We report the experimental results in Section 5. Section 6 concludes this work.

## 2. RELATED WORK

**Community detection (CD).** Discovering communities from a network is a fundamental problem in network science, and it has been widely studied in the past decades. Classical solutions [25, 14] employ link-based analysis to obtain these communities. However, they do not consider the location information. Some recent works [15, 16, 10, 4] focus on identifying communities from spatially constrained graphs, whose vertices are associated with spatial coordinates [2]. For example, a geo-community [15] 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 (ALK) measure for clustering objects in spatially constrained graphs. In [10], Expert et al. uncovered communities from spatial graphs based on modularity maximization. In [4], Chen et al. proposed an algorithm based on fast modularity maximization for detecting communities from spatially constrained networks. We will compare it with our methods in experiments. The differences of CD algorithms and our SAC search are three-fold. First, CD algorithms are generally costly and time-consuming, as they often detect all the communities from an entire network. Second, it is not clear how they can be adapted for online community retrieval. Third, as pointed out by [20], the modularity based methods [10, 4] often fail to resolve small-size communities, even when they are well defined. In this paper, we propose online algorithms for finding SACs from large spatial graphs.

**Community search (CS).** In recent years, there is another related but different problem of community detection, called *community search*. The goal of community search is to obtain communities in an "online" manner, based on a query request. For example, given a vertex $q$, several existing works [29, 7, 6, 21, 19] 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 [29, 7]. In [29], Sozio et al. proposed the first algorithm Global to find the $k$-$\widehat{core}$ containing $q$. In [7], 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 [21, 11] also use the minimum degree metric to search communities from attributed graphs. Other well known structure cohesiveness metrics, including $k$-clique [6], $k$-truss [19] and connectivity [18], 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 design algorithms for searching communities from spatial graphs.

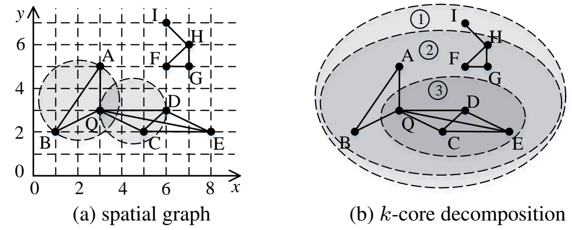
**Table 2: Notations and meanings.**

| Notation | Meaning |
|---|---|
| $G(V, E)$ | a graph with vertex set $V$ and edge set $E$ |
| $n, m$ | the sizes of vertex and edge sets $V$ and $E$ resp. |
| $G[S]$ | 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 circle with center $o$ and radius $r$ |
| $\|u, v\|$ | the Euclidean distance from vertices $u$ to $v$ |
| $\Psi$ | Results of Exact and Exact+ |
| $\Phi, \Lambda, \Gamma$ | Results of AppInc, AppFast, AppAcc resp. |

## 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 location position $(v.x, v.y)$, where $v.x$ and $v.y$ denote its positions along $x$- and $y$-axis in a two-dimensional space. Note that our methods can be easily applied to multi-dimensional space. Let $n$ and $m$ be the corresponding sizes of $V$ and $E$. We illustrate the data model using Example 1. Table 2 shows the notations used in this paper.

EXAMPLE 1. *Figure 3(a) depicts a geo-social network containing 10 vertices $\{Q, A, B, \cdots, I\}$. The solid lines linking the vertices are the edges, denoting their social relationships.*



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

**Figure 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.

**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 $k$ [29, 28, 3, 7, 21]. This is used in $k$-core and our SAC search. Let us discuss the $k$-core first.

DEFINITION 1 ($k$-CORE [28, 3]). *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 important properties [3]: (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 [3].

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 [29, 7]. 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 as the structure cohesiveness metric, our solutions can be easily adapted to other structure cohesiveness criteria like $k$-truss [19] and $k$-clique [6].

**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 [8, 9, 24, 17], 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.*

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 $\Psi$). We denote the radius of the MCC containing $\Psi$ 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.

We also consider the $\theta$-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 $\theta$ is an input parameter. This $\theta$-SAC search is essentially a variant of `Global` [29] by introducing a parameter $\theta$. Consider the graph in Example 1 with $q$=$Q$, $k$=2 and $\theta$=3.1. $\theta$-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 $\theta$-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 $\theta$. As will be discussed in our experiments, the effectiveness of $\theta$-SAC search is sensitive to $\theta$. If $\theta$ is too small, no community can be found; if $\theta$ is too large, then the community is not spatially compact. A casual application user may then have to repeat the query with different $\theta$ values, before getting a satisfactory result. For the SAC search, the user does not need to specify $\theta$; instead, SAC search automatically suggests a community with tight structural and spatial cohesiveness. Thus, SAC search is more convenient to use than $\theta$-SAC. In the above example, if $\theta$<2.2, no community is found; if $\theta$>5.1, $G[C_3]$ will be returned, where $C_3$=$\{Q, A, B, C, D, E\}$. In fact, there are more spatially compact 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. We next focus on SAC search.

# 4. SAC SEARCH ALGORITHMS

We now present fast SAC search algorithms. Most of our solutions follow the *two-step framework*: (1) find a community $S$ of vertices, based on some CS algorithm e.g., `Global` [29], 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).

**Table 3: Overview of algorithms for SAC search.**

| Algo. | Approx. ratio | Time complexity |
|---|---|---|
| `Exact` | 1 | $O(m \times n^3)$ |
| `AppInc` | 2 | $O(mn)$ |
| `AppFast` | 2+$\epsilon_F$ ($\epsilon_F$≥0) | If $\epsilon_F$>0, $O(m \cdot \min\{n, \log \frac{1}{\epsilon_F}\})$<br>If $\epsilon_F$=0, $O(mn)$ |
| `AppAcc` | 1+$\epsilon_A$ (0<$\epsilon_A$<1) | $O(\frac{m}{\epsilon_A^2} \times \min\{n, \log \frac{1}{\epsilon_A}\})$ |
| `Exact+` | 1 | $O(\frac{m}{\epsilon_A^2} \cdot \min\{n, \log \frac{1}{\epsilon_A}\} + m\|F_1\|^3)$ |

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+`, which is at least four orders of magnitude faster than `Exact` as shown by our experiments. Their approximation ratios and time complexities are summarized in Table 3, where $\epsilon_F$ and $\epsilon_A$ are parameters specified by the query user. The value $|F_1|$ is the number of "fixed vertices", which will be defined in Section 4.1; $|F_1|$ is often much smaller than $n$. We will explain this parameter in more detail. Note that the space cost of each algorithm is linear with the size of graph $G$.

`AppInc` is a 2-approximation algorithm, and it is much faster than `Exact`. Inspired by `AppInc`, we design another (2+$\epsilon_F$)-approximation algorithm `AppFast`, where $\epsilon_F$≥0, which is faster than `AppInc`. The limitation of `AppInc` and `AppFast` is that their theoretical approximation ratios are at least 2. To achieve even lower approximation ratio, we further design another algorithm `AppAcc`, whose approximation ratio is (1+$\epsilon_A$), 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 expected 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 the input $k$=1, we can simply return the subgraph, induced by $q$ and its nearest neighbor, as the result. So in the rest of this paper, we mainly focus on the case $k \geq 2$.

We now describe a useful lemma about MCC, described in [9], which inspires the design of our algorithms.

LEMMA 1. *[9] 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. Algorithm 1 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 $\Psi$ (lines 4-14).

**Algorithm 1** 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;
```

In addition, we present another useful lemma, which is about the maximum pair-wise distance for vertices in $\Psi$.

LEMMA 2. *[17] The maximum distance between any pair of vertices, $u$ and $v$ in $\Psi$, is in the range $[\sqrt{3}r_{opt}, 2r_{opt}]$.*

**Complexity.** The time complexity of Exact is $O(m \times n^3)$, since there are three nested for-loops and finding a $k$-$\widehat{core}$ takes linear time cost $O(m)$ (we assume $m \geq n$) [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 $\Psi$ 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 $\Phi$. Let the radius of the MCC covering $\Phi$ be $\gamma$ ($\gamma \leq \delta$). Note that, $\gamma$ can be obtained by computing the MCC containing $\Phi$ by a linear algorithm [24]. Then, we have the following two interesting lemmas:

LEMMA 3. $\frac{1}{2}\delta \leq r_{opt} \leq \gamma$.

PROOF. We have $r_{opt} \leq \gamma$ obviously, as $\Psi$ 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 $\Psi$ 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 $\Psi$ must be in a circle, whose center is $q$ and radius is smaller than $\delta$. This contradicts the fact that, $O(q, \delta)$ is the minimum circle with center $q$ containing a feasible solution. Hence, Lemma 3 holds. □

LEMMA 4. *The radius of the MCC covering the feasible solution $\Phi$ has an approximation ratio of 2.*

PROOF. Let $S$ be the set of vertices in $O(q, \delta)$. Since the vertex set of $\Phi$ is a subset of $S$, the MCC of $\Phi$ has a radius no larger than that of $S$, i.e., $\gamma \leq \delta$. By Lemma 3, 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 4 holds. □

AppInc finds $\Phi$ 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.

**Algorithm 2** Query algorithm: AppInc

```
1:  function APPINC(G, q, k)
2:      initialize Queue, S ← ∅, T ← ∅, Φ ← ∅;
3:      Queue.add(q);
4:      while |Queue| > 0 do
5:          p ← Queue.poll();
6:          S.add(p);
7:          for v ∈ nb(p) do
8:              if deg_G(v) ≥ k then
9:                  if |v, q| ≤ |p, q| then
10:                     S.add(v);
11:                 else if v ∉ T then
12:                     Queue.add(v);   T.add(v);
13:         if |S ∩ nb(q)| ≥ k ∧ |S ∩ nb(p)| ≥ k then
14:             if exist a k-core^ containing q in G[S] then
15:                 Φ ← this k-core^; break; //stop
16:     return Φ;
```

Algorithm 2 presents AppInc. First, it initializes four variables $Queue$, $S$, $T$ and $\Phi$: $Queue$ 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 $\Phi$ 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 $q$

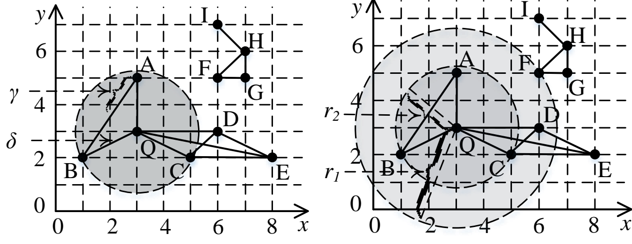



**Figure 4: Illustrating** AppInc. **Figure 5: Illustrating** AppFast.

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).

We illustrate AppInc using 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 $\Phi$ with members set $\{Q, A, B\}$. So $\gamma$=1.803 and $\delta$=$|Q, B|$=2.24. The actual approximation ratio is 1.803/1.5=1.202.*

COROLLARY 1. *If $q$ is the center of the MCC covering $\Psi$,* AppInc *finds the optimal solution, i.e., $\Phi$ equals to $\Psi$.*

PROOF. This can be proved directly by contradiction. □

COROLLARY 2. *The optimal solution $\Psi$ is in $O(q, 2\gamma)$.*

PROOF. By Lemma 3, we have $r_{opt} \le \gamma$. This implies that, for any $v \in \Psi$, we have $|q, v| \le 2 \times \gamma$. Thus, all the vertices of $\Psi$ are in $O(q, 2\gamma)$, and Corollary 2 holds. □

**Complexity.** In AppInc, the while loop is executed at most $n$ times, and each takes $O(m)$, as computing $k$-$\widehat{core}$ takes $O(m)$. So the total time cost of AppInc is $O(mn)$.

## 4.3 A (2+$\epsilon_F$)-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 $\epsilon_F$ is an arbitrary non-negative value.

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

Algorithm 3 presents AppFast. We denote the SAC returned by AppFast by $\Lambda$. $\epsilon_F$ is an input parameter. By following the two-step framework, it first computes the $k$-$\widehat{core}$ (line 2), and then finds $\Lambda$ from the $k$-$\widehat{core}$ (lines 3-14). Some variables such as $\Lambda$, $l$ and $u$ are initialized (line 3). In while loop (lines 4-14), it first finds an SAC $\Lambda'$ from $O(q, r)$ using breadth first search (BFS). If $\Lambda'$ does exist, it first updates $\Lambda$, since this solution has a smaller radius. It then checks whether the gap, i.e., $r - l$, is smaller than $\alpha$ (we will discuss how to set this gap later). If it is not larger than $\alpha$, then it returns $\Lambda$; otherwise, it updates $u$ as the maximum distance from $q$ to vertices in $\Lambda$, which ensures that the feasible solution found later has at least one less vertex than $\Lambda$. If $\Lambda'$ does not exist,

**Algorithm 3** Query algorithm: AppFast

1: **function** APPFAST($G$, $q$, $k$, $\epsilon_F$)
2: find the vertex list $X$ of the $k$-$\widehat{core}$, $\Lambda$, 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** $\Lambda' \ne \emptyset$ **then**
9: $\Lambda \leftarrow \Lambda'$;
10: **if** $r - l \le \alpha$ **then return** $\Lambda$;
11: $u \leftarrow \max_{v \in \Lambda} |q, v|$;
12: **else**
13: **if** $u - r \le \alpha$ **then return** $\Lambda$;
14: $l \leftarrow \min_{v \in \Lambda \wedge v \notin S} |q, v|$;

it returns $\Lambda$ if the gap, i.e., $u - r$, is small enough; otherwise, it updates $l$ as the minimum distance from $q$ to vertices in $\Lambda$, but not in $S$, which ensures that the set $S$ in the next iteration has at least one more vertex than current $S$.

We illustrate AppFast using Example 3.

EXAMPLE 3. *In Figure 5 (q=Q, k=2, $\epsilon_F$=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. $\Lambda$ is the same with $\Phi$.*

LEMMA 5. *In* AppFast, *the radius of the MCC covering $\Lambda$ has an approximation ratio of $(2 + \epsilon_F)$, if $\alpha$ is set as $\frac{r \times \epsilon_F}{2+\epsilon_F}$.*

PROOF. Consider the last loop in Algorithm 3 when returning $\Lambda$. Let the gap between the radii, which result in a feasible solution and no solution, be $\alpha$.

If $\Lambda'$ does exist (lines 8-10), the returned $\Lambda$ is contained in $O(q, r)$. We have $l \le \delta \le r \le u$ and $r - l \le \alpha$ (see Figure 6(a)). So we have $r \le \delta + \alpha$.

If $\Lambda'$ does not exist (lines 12-13), the returned $\Lambda$ is contained in $O(q, u)$. We have $l \le r \le \delta \le u$ and $u - r \le \alpha$ (see Figure 6(b)). So we have $r \le \delta + \alpha$.

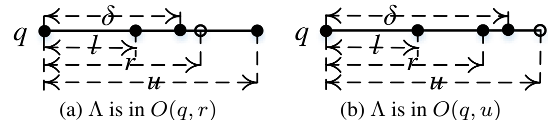
Therefore, we always have $r \le \delta + \alpha$. We denote the radius of the MCC covering $\Lambda$ by $r_\Lambda$. Considering Lemma 3, we have

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

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

$$\frac{r_\Lambda}{r_{opt}} \le \frac{2r_{opt}+\alpha}{r_{opt}} = 2 + \frac{\alpha}{r_{opt}} \le 2 + \frac{2\alpha}{r-\alpha}. \tag{3}$$

Let $\frac{2\alpha}{r-\alpha} \le \epsilon_F$, then we have $\frac{r_\Lambda}{r_{opt}} \le 2 + \epsilon_F$, if $\alpha$ is set as $\frac{r \times \epsilon_F}{2+\epsilon_F}$. Hence, Lemma 5 holds. □




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

**Figure 6: Illustrating the proof of Lemma 5.**

**Remark:** If $\epsilon_F$=0, the returned community $\Lambda$ is the same as $\Phi$.

COROLLARY 3. *The optimal solution $\Psi$ is in $O(q, 2r_\Lambda)$, where $r_\Lambda$ is the radius of the MCC containing $\Lambda$ in* AppFast.

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

**Complexity.** 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. Also, each loop takes $O(m)$. Thus, the total time cost of AppFast is $O(\min\{mn, m\log\frac{1}{\epsilon_F}\})$ if $\epsilon_F$>0, or $O(mn)$ if $\epsilon_F$=0.

## 4.4 A (1+$\epsilon_A$)-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 3, stated by Corollary 4:

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

PROOF. This can be proved directly by contradiction. □

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

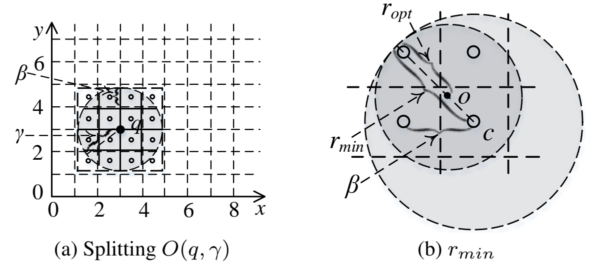
We consider the circle $O(c, r_{min})$, where $r_{min}$ is the minimum radius such that it contains a feasible solution, which is denoted by $\Gamma$. The value of $r_{min}$ is bounded by the following lemma.

LEMMA 6. $r_{min} \le r_{opt} + \frac{\sqrt{2}}{2}\beta$.

PROOF. We prove by contradiction. Suppose that $r_{min} > r_{opt} + \frac{\sqrt{2}}{2}\beta$. As mentioned before, we have $|o,c| \le \frac{\sqrt{2}}{2}\beta$. For any point $c'$ in $O(o, r_{opt})$, we have $|o,c'| \le r_{opt}$. By triangle inequality, we conclude that $|c,c'| \le |c,o| + |o,c'| \le r_{opt} + \frac{\sqrt{2}}{2}\beta$. This contradicts that $r_{min}$ is the minimum radius such that $O(c, r_{min})$ contains a feasible solution. Hence, Lemma 6 holds. □

By Lemma 6, we have $\frac{r_{min}}{r_{opt}} \le 1 + \frac{\sqrt{2}\beta}{2r_{opt}} \le 1 + \frac{\sqrt{2}\beta}{\delta}$. Thus, we can approximate $\Psi$ using $\Gamma$, and the approximation ratio is (1+$\epsilon_A$), if we let $\frac{\sqrt{2}\beta}{\delta} \le \epsilon_A$ $(0 < \epsilon_A < 1)$.

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 $\beta$ ($\epsilon_A$) is very small. To further improve the efficiency, we develop some optimization techniques.



(a) Splitting $O(q,\gamma)$ (b) $r_{min}$

**Figure 7: Illustrating AppAcc.**

Specifically, we assume that all the anchor points are organized into a region quadtree [13], where the root node[4] is a square, centered at $q$ with width $2\gamma$. 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 $\gamma$, 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 two effective pruning criteria:

- *Pruning1:* Consider a node (with center $p$), which intersects at the boundary of $O(q, r_{cur})$. Then we have $|p,q| \le r_{cur} + \frac{\sqrt{2}}{2}\beta$. 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.

Based on above analysis, we design AppAcc (see Algorithm 4). $\epsilon_A$ is an input parameter. It first runs AppFast ($\epsilon_F$=0), and obtains the $\widehat{k\text{-}core}$ in $O(q, 2\gamma)$ (lines 2-3), which contains $\Psi$ by Corollary 2. Then it initializes four variables: $\Gamma$ is the target SAC, $\beta$ equals to $\gamma$, $r_{cur}$ 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 $\Gamma$, 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 (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 $\beta$ and collect all the child nodes needed to be considered in the next level (lines 26-27). The loop is executed until $\beta$ is smaller than the threshold $\frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$ (we will discuss this threshold later). Finally, $\Gamma$ is returned (line 28).

LEMMA 7. *In* AppAcc, *if we set $\alpha' \le \frac{1}{4}\delta e_A$ and $\beta = \frac{\delta\epsilon_A}{\sqrt{2}(2+\epsilon_A)}$, where $0 < \epsilon_A < 1$, the radius of the MCC covering $\Gamma$ has an approximation ratio of (1+$\epsilon_A$).*

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

[4]To avoid ambiguity, we use word "node" for tree nodes.

**Algorithm 4** Query algorithm: `AppAcc`

```
1: function AppAcc(G, q, k, ε_A)
2:     obtain Φ, δ and γ using AppFast;
3:     S ← vertices of the k-ĉore, containing q, in O(q, 2γ);
4:     Γ ← Φ, β ← γ, r_cur ← γ, achList ← center points;
5:     while β ≥ δε_A / (√2(2+ε_A)) do
6:         map ← ∅;
7:         for each point p ∈ achList do
8:             if |p, q| ≤ r_cur + (√2/2)β then //Pruning1
9:                 Γ_p ← find an SAC in O(p, r_cur + (√2/2)β);
10:                if Γ_p ≠ ∅ then
11:                    u ← r_cur + (√2/2)β, l ← δ/2, map.put(p, l);
12:                    while u ≥ l do
13:                        r ← (l+u)/2;
14:                        Γ'_p ← find an SAC in O(p, r);
15:                        if Γ'_p ≠ ∅ then
16:                            Γ_p ← Γ'_p;
17:                            if r − l ≤ α' then break;
18:                            u ← max_{v∈Γ_q} |q, v|;
19:                        else
20:                            map.put(p, l);
21:                            if u − r ≤ α' then break;
22:                            l ← min_{v∈S∧v∉O(p,r)} |q, v|;
23:                    r ← radius of the MCC covering Γ_p;
24:                    if r < r_cur then r_cur ← r; Γ ← Γ_p;
25:        prune anchor points in map using Pruning2;
26:        β ← β/2;
27:        update anchor point list achList using map;
28:    return Γ;
```

solution. From the proof of Lemma 5, we can conclude that, $r \le r_p + \alpha'$ when the binary search stops. Then, we have

$$\frac{r}{r_p} \le 1 + \frac{\alpha'}{r_p} \le 1 + \frac{\alpha'}{r_{opt}} \le 1 + \frac{2\alpha'}{\delta}. \quad (4)$$

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

Consider the updated $r_{cur}$ after the binary search for all the anchor points. Then we have $r_{cur} \le \left(1 + \frac{\epsilon_A}{2}\right) r_{min}$. Let $r_\Gamma$ be the radius of the MCC covering $\Gamma$. By Lemmas 3 and 6, we have

$$\frac{r_\Gamma}{r_{opt}} \le \frac{r_{cur}}{r_{opt}} \le 1 + \frac{\epsilon_A}{2} + \frac{(2+\epsilon_A)\sqrt{2}\beta}{2\delta}. \quad (5)$$

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

**Complexity.** There are $O((\frac{2\gamma}{\beta})^2)$=$O((\frac{1}{\epsilon_A})^2)$ anchor points. Similar as 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 cost of `AppAcc` is $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\})$.

## 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 $\Psi$ 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 $\Psi$ 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.

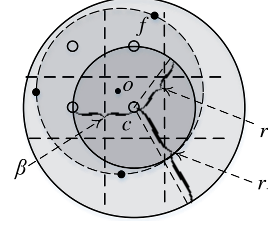



**Figure 8: Illustrating the annular region in `Exact+`.**

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

$$\frac{r_\Gamma}{1+\epsilon_A} \le r_{opt} \le r_\Gamma, \quad (6)$$

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

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

$$|f, c| \le |f, o| + |o, c| \le r_\Gamma + \frac{\sqrt{2}}{2}\beta, \quad (7)$$

$$|f, c| \ge |f, o| - |o, c| \ge \frac{r_\Gamma}{1+\epsilon_A} - \frac{\sqrt{2}}{2}\beta. \quad (8)$$

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 $\epsilon_A$ 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 above analysis, we design `Exact+` (Algorithm 5). It first runs `AppAcc` with a small value of $\epsilon_A$ (line 2). Note that $\Psi$ is initialized as $\Gamma$, 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 (7) and (8), 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, .

**Complexity.** `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 $O(m(\frac{1}{\epsilon_A})^2 \times \min\{n, \log\frac{1}{\epsilon_A}\} + m|F_1|^3)$. We will address the effect of $\epsilon_A$ on $|F_1|$ and the performance of Exact+ in the experiments.

**Algorithm 5** Query algorithm: Exact+

```
1:  function EXACT+(G, q, k, ε_A)
2:      run APPACC(G, q, k, ε_A);
3:      T ← anchor points in map of AppAcc;
4:      initialize F_1 ← ∅;
5:      for p ∈ T do F_1.add{v | r_- ≤ |p,v| ≤ r_+ ∧ v ∈ S};
6:      for v_1 ∈ F_1 do
7:          F_2 ← {v | √3 r_- ≤ |v_1,v| ≤ 2r_cur ∧ v ∈ F_1};
8:          for v_2 ∈ F_2 do
9:              F_3 ← {v | |v_1,v| ≤ |v_1,v_2| ∧ v ∈ F_1};
10:             for v_3 ∈ F_3 do
11:                 compute the MCC mcc of {v_1, v_2, v_3};
12:                 if mcc.radius < r_cur then
13:                     R ← a set of vertices in mcc;
14:                     if exist a k-ĉore in G[R] then
15:                         r_cur ← mcc.radius;
16:                         Ψ ← this k-ĉore;
17:     return Ψ;
```

# 5. EXPERIMENTAL RESULTS

We describe the setup in Section 5.1. Sections 5.2 and 5.3 report the effectiveness and efficiency results of SAC search.

## 5.1 Setup

**Datasets.** We consider four real datasets: Brightkite[5], Gowalla[5], Flickr[6] and Foursquare[7]. 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. With respect to Brightkite dataset, we consider the users' locations can be both static (Sections 5.2.1, 5.2.2 and 5.3) and dynamic (Section 5.2.3). The static location associated with a user is the place she checks in (or takes photos) most frequently. The Foursquare dataset [26] is extracted from Foursquare website, and the location of each user is her hometown position. Users without locations are shipped.

We have also performed experiments on synthetic datasets. We are not aware of any existing spatial graph data generators. Therefore, we create synthetic data in the following way. First, we use GTGraph[8], a well-known graph generator, to generate a (non-spatial) graph first. 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 follows a normal distribution with mean $\mu$ and standard deviation $\sigma$. We repeat this step for other vertices, starting from $v$'s neighbors, until every vertex is associated with a location. We set $\mu$=0.09 and $\sigma$=0.16; these values are derived from the Brightkite dataset. Following these settings, we create two spatial graphs of different sizes, namely Syn1 and Syn2.

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

**Table 4: Datasets used in our experiments.**

| Type | Name | Vertices | Edges | $\widehat{d}$ |
|---|---|---|---|---|
| Real | Brightkite | 51,406 | 197,167 | 7.67 |
| | Gowalla | 107,092 | 456,830 | 8.53 |
| | Flickr | 214,698 | 2,096,306 | 19.5 |
| | Foursquare | 2,127,093 | 8,640,352 | 8.12 |
| Synthetic | Syn1 | 30,000 | 300,000 | 20 |
| | Syn2 | 400,000 | 4,000,000 | 20 |

**Table 5: Parameter settings.**

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

The statistics of each dataset are summarized in Table 4, where $\widehat{d}$ is the average degree. Without loss of generality, we normalize all the locations of each dataset into the unit square $[0,1]^2$.

**Parameters.** We consider 5 parameters: $\epsilon_F$ (the parameter of AppFast), $\epsilon_A$ (the parameter of AppAcc), $k$ (denoting the minimum degree), $\theta$ (the parameter of $\theta$-SAC search), and the percentage of vertices $n$. The ranges of the parameters and their default values are shown in Table 5. The default values of $\epsilon_F$ and $\epsilon_A$ 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% 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.

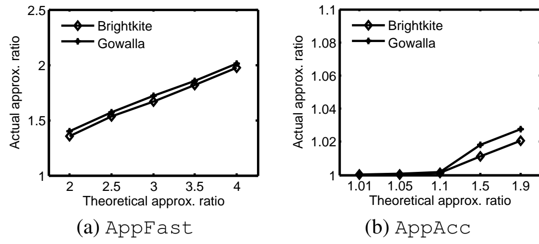
**Queries.** For each dataset, we randomly select 200 query vertices with core numbers of 4 or more. Such a core number constraint ensures a meaningful community (at least 4-$\widehat{core}$) containing the query vertex. In the results reported in the following, each data point is the average result for these 200 queries. We use the term "AppFast($\epsilon$)" ("AppAcc($\epsilon$)") to denote the algorithm AppFast (AppAcc) with the parameter $\epsilon_F$=$\epsilon$ ($\epsilon_A$=$\epsilon$). 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.

## 5.2 Effectiveness Evaluation

In this section, we first study the approximation ratios of approximation algorithms, then compare SAC search with the state-of-art methods, and finally show results on dynamic graphs.

### 5.2.1 *Approximation Ratio*

In Figure 9, we report the theoretical and actual approximation ratios of AppFast and AppAcc on Brightkite and Gowalla datasets. Note that if we set $\epsilon_F$=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 $\epsilon_F$=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 9(b).




(a) AppFast (b) AppAcc

**Figure 9: Approximation ratio.**

### *5.2.2 Comparison with the State-of-the-Arts*

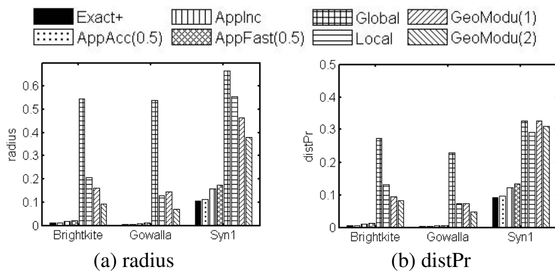
In this subsection, we show that SAC search returns communities with higher spatial cohesiveness compared with the state-of-the-art community retrieval methods: `Global` [29], `Local` [7] and `GeoModu` [4]. The first two methods are CS methods designed for non-spatial graphs, while `GeoModu` is a CD method for spatial graphs. We also compare it with $\theta$-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 $\mu$ (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.
- $\theta$-`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 $\mu$ 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 $G_q$.

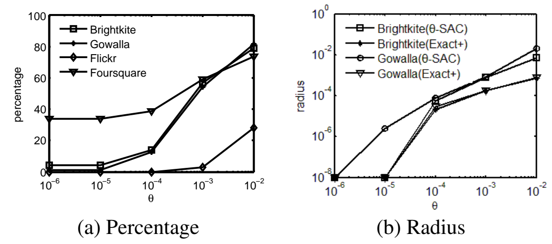
Intuitively, lower values of these metrics for a community imply that it achieves higher spatial cohesiveness. To compare these methods, we consider both exact and approximation algorithms. We search communities using these algorithms and compute the average values of above metrics for these communities. We report the results on Brightkite and Gowalla datasets in Figure 10.




(a) radius (b) distPr

**Figure 10: 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.



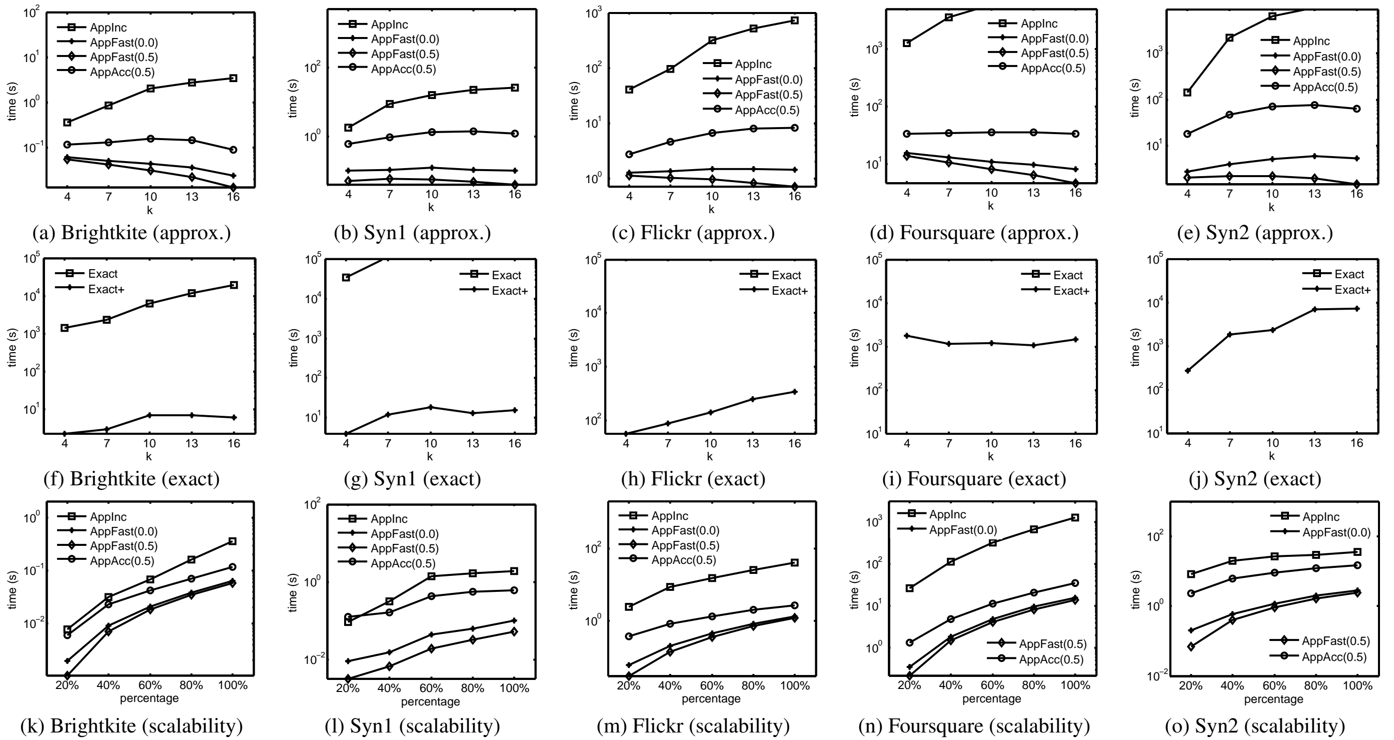

(a) Percentage (b) Radius

**Figure 11: Results of $\theta$-SAC search.**

**2. CD comparison.** Since `GeoModu` considers both links and locations, the returned communities achieve better spatial cohesiveness than `Global` and `Local`. While the average radius and distPr values of `GeoModu` are larger than those of SAC search, 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 [4], which has no reference to the query vertices. Thus, SAC search achieves higher structure and spatial cohesiveness than `GeoModu`.

**3. Comparison with $\theta$-SAC search.** We vary the value of $\theta$ in $\theta$-SAC search, and compute the percentage of queries returning non-empty subgraphs. Figure 11(a) reports the results. Notice that the percentage is low when $\theta$ 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 $\theta$ is not easy. In contrast, SAC search does require the specification of $\theta$, 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 11(b) compares their results. We observe that the average radius of MCCs covering SACs found by $\theta$-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 $\theta$-SAC.

In addition, we have tried another approach by simply extracting vertices within $O(q, \theta)$ as a community, in which there is no structure cohesiveness requirement. We vary the value of $\theta$ and compute the average degree of vertices in the communities. The results show that, the average degree is very low. For example, the average values on Brightkite dataset are 0.36 and 0.39, when $\theta$ is $10^{-6}$ and $10^{-5}$ respectively. This implies that the community may not be a connected subgraph. Thus, only using locations is insufficient for identifying communities. In contrast, SAC search always guarantees that each vertex in an SAC has a minimum degree of $k$ or more.




(a) Brightkite (approx.) (b) Syn1 (approx.) (c) Flickr (approx.) (d) Foursquare (approx.) (e) Syn2 (approx.)

(f) Brightkite (exact) (g) Syn1 (exact) (h) Flickr (exact) (i) Foursquare (exact) (j) Syn2 (exact)

(k) Brightkite (scalability) (l) Syn1 (scalability) (m) Flickr (scalability) (n) Foursquare (scalability) (o) Syn2 (scalability)

**Figure 12: Efficiency evaluation.**

### *5.2.3 Adaptability to Location Changes*

We now study the adaptability of location changes of SAC search. We focus on "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.

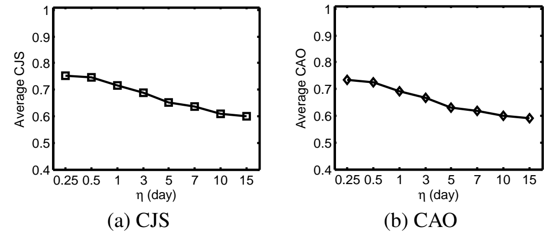
To evaluate the adaptability of location changes for SAC search, we first go through checkin records in $R_1$ and update users' locations according to their latest checkin timestamps. Then, for each user $q \in Q$, we do the same operation for records in $R_2$, and if the record was generated by $q$,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

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

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

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

**Figure 13: 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 $\eta$, and compute their CJS and CAO values. We report their average results in Figure 13, where $\eta$ varies from 0.25 day to 15 days. From Figure 13(a), we can observe that, the CJS decreases as the time threshold increases. For example, after 6 hours, the CJS decreases to 75%. In Figure 13(b), similar results can be observed for CAO. In addition, we plot the SACs of two users in Figure 2. Thus, these results well confirm that, SAC search has high adaptability to location changes.

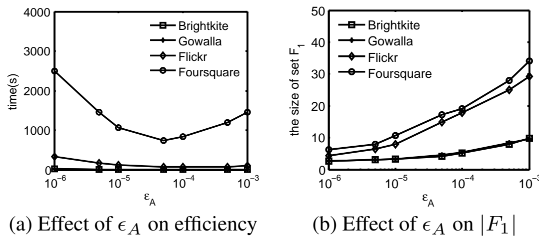
## 5.3 Efficiency Evaluation

**1. Effect of $k$ for approximation algorithms.** Figures 12(a)-(e) report the results. We skip the results on Gowalla, due to the space limitation. We can see that `AppFast` runs consistently faster than `AppInc` and `AppAcc`. For example, for the largest 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. The reason is that, 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 $\epsilon_A$. Since $\epsilon_A$ 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 14(a) shows that the efficiency of `Exact+` is not very sensitive to $\epsilon_A$ on all real datasets except Foursquare. Figure 14(b) shows $|F_1|$ increases with $\epsilon_A$; that is, fewer vertices are pruned as $\epsilon_A$ grows. Recall in Sec. 4.5 that `Exact+` is composed of two phases. When $\epsilon_A$ is small, the cost of Phase (1) dominates the overall cost; as $\epsilon_A$ increases, the cost of Phase (2) grows. Thus, there is a local minimum in Figure 14(a). In practice, a sensitivity test can be done to choose $\epsilon_A$.




(a) Effect of $\epsilon_A$ on efficiency (b) Effect of $\epsilon_A$ on $|F_1|$

**Figure 14: Effect of $\epsilon_A$ on the efficiency of `Exact+`.**

Figures 12(f)-(j) show the results of exact algorithms ($\epsilon_A$=$10^{-4}$). We skip the results of `Exact`, if a single query takes more than 10 hours. We can see that, `Exact` performs extremely slow, even on the smallest dataset. This because it adopts an exhaustive search to find $\Psi$, which enumerates all the three vertex combinations. Hence, it is really worth the effort to study more efficient algorithms.

`Exact+` is at least four orders of 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 with the annular regions, the numbers of fixed vertices left may be different, but in general larger datasets have more fixed vertices, and thus more time cost is needed. In addition, since `Exact+` runs `AppAcc` in the first step, it is slower than the approximation algorithms, but it takes reasonable time, i.e., few seconds, on moderate-size graphs like Brightkite and Syn1.

**3. Scalability.** We vary the percentage of vertices in all the datasets to study the scalability of approximation algorithms. The results are reported in Figures 12(k)-(o). We can see that, `AppInc`, `AppFast` and `AppAcc` generally scale well with $n$. Their performance trends are similar with those discussed before. In addition, `AppFast(0.0)` scales slightly better than `AppInc`.

# 6. CONCLUSIONS

In this paper, we study online SAC search algorithms. We propose two exact algorithms, and three efficient approximation algorithms. Our experiments show that SAC search achieves higher effectiveness than the state-of-the-art CD and CS algorithms. In the future, we will examine other spatial cohesiveness measures (e.g., pair-wise vertex distances). We will study how to support batch processing for SAC search. We will also develop a system prototype and collect user statistics, to perform qualitative comparison among communities generated by different solutions.

## Acknowledgments

Reynold Cheng, Yixiang Fang, Xiaodong Li, Siqiang Luo, and Jiafeng Hu were supported by the Research Grants Council of HK (Project HKU 17205115 and 17229116) and HKU (Projects 102009508 and 104004129). We would like to thank Dr. Mauro Sozio for his insightful comments.

# 7. REFERENCES

[1] N. Armenatzoglou et al. A general framework for geo-social query processing. In *PVLDB*, volume 6, pages 913–924, 2013.
[2] M. Barthélemy. Spatial networks. *Physics Reports*, 499(1):1–101, 2011.
[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *arXiv*, 2003.
[4] Y. Chen et al. Finding community structure in spatially constrained complex networks. *IJGIS*, 29(6):889–911, 2015.
[5] Y. Chon, N. D. Lane, F. Li, H. Cha, and F. Zhao. Automatically characterizing places with opportunistic crowdsensing using smartphones. In *UbiComp*, pages 481–490, 2012.
[6] W. Cui et al. Online search of overlapping communities. In *SIGMOD*, pages 277–288, 2013.
[7] W. Cui et al. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.
[8] D. J. Elzinga and D. W. Hearn. The minimum covering sphere problem. *Management science*, 19(1):96–104, 1972.
[9] J. Elzinga and D. W. Hearn. Geometrical solutions for some minimax location problems. *Transportation Science*, 6(4):379–394, 1972.
[10] P. Expert et al. Uncovering space-independent communities in spatial networks. *PNAS*, 108(19):7663–7668, 2011.
[11] Y. Fang et al. Effective community search for large attributed graphs. In *PVLDB*, pages 1233–1244, 2016.
[12] Y. Fang et al. Scalable algorithms for nearest-neighbor joins on big trajectory data. *TKDE*, 28(3):785–800, 2016.
[13] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.
[14] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.
[15] M. Girvan et al. Community structure in social and biological networks. *PNAS*, 99(12):7821–7826, 2002.
[16] D. Guo. Regionalization with dynamically constrained agglomerative clustering and partitioning (redcap). *IJGIS*, 22(7):801–823, 2008.
[17] T. Guo et al. Efficient algorithms for answering the m-closest keywords query. In *SIGMOD*, pages 405–418. ACM, 2015.
[18] J. Hu et al. Querying minimal steiner maximum-connected subgraphs in large graphs. In *CIKM*, pages 1241–1250. ACM, 2016.
[19] X. Huang et al. Approximate closest community search in networks. In *PVLDB*, pages 276–287, 2015.
[20] A. Lancichinetti et al. Limits of modularity maximization in community detection. *Phy. Rev. E*, 84(6):066122, 2011.
[21] R.-H. Li et al. Influential community search in large networks. In *PVLDB*, volume 8, pages 509–520, 2015.
[22] K. M. MacQueen et al. What is community? an evidence-based definition for participatory public health. *AJPH*, 91(12):1929–1938, 2001.
[23] P. Manchanda et al. Social dollars: the economic impact of customer participation in a firm-sponsored online customer community. *Marketing Science*, 34(3):367–387, 2015.
[24] N. Megiddo. Linear-time algorithms for linear programming in r3 and related problems. In *FOCS*, pages 329–338, 1982.
[25] M. Newman et al. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026–113, 2004.
[26] M. Sarwat et al. Lars*: a scalable and efficient location-aware recommender system. *TKDE*, 26(6):1384–1399, 2014.
[27] J. Scott. *Social network analysis*. Sage, 2012.
[28] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
[29] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, pages 939–948, 2010.
[30] W. Zhang, J. Wang, and W. Feng. Combining latent factor model with location features for event-based group recommendation. In *KDD*, pages 910–918, 2013.