Efficient Closest Community Search over Large Graphs

Mingshen Cai[1] and Lijun Chang[2]

[1] Canva*, sam.cai@canva.com
[2] The University of Sydney, lijun.chang@sydney.edu.au

Abstract. This paper studies the closest community search problem. Given a graph G and a set of query vertices Q, the closest community of Q in G is the connected subgraph of G that contains Q, is most cohesive (*i.e.*, with the largest possible minimum vertex degree), is *closest* to Q, and is maximal. We show that this can be computed via a two-stage approach: (1) compute the maximal connected subgraph g_0 of G that contains Q and is most cohesive, and (2) iteratively remove from g_0 the vertex that is furthest to Q and subsequently also other vertices that violate the cohesiveness requirement. The last non-empty subgraph is the closest community of Q in G. We first propose baseline approaches for the two stages that run in $O(n + m)$ and $O(n_0 \times m_0)$ time, respectively, where n (resp. n_0) and m (resp. m_0) are the number of vertices and edges in G (resp. g_0). Then, we develop techniques to improve the time complexities of the two stages into $O(n_0 + m_0)$ and $O(m_0 + n_0 \log n_0)$, respectively. Moreover, we further design an algorithm CCS with the same time complexity as $O(m_0 + n_0 \log n_0)$, but performs much better in practice. Extensive empirical studies demonstrate that CCS can efficiently compute the closest community over large graphs.

1 Introduction

The graph model has been widely used to capture the information of entities and their relationships, where entities are represented by vertices and relationships are represented by edges [13]. With the proliferation of graph data, research efforts have been devoted to managing, mining and querying large graphs. In this paper, we study the problem of community search for a given set of query vertices, where a community is a group of vertices that are densely connected to each other [7, 9].

Traditionally, the problem of community detection has been extensively studied (*e.g.*, see the survey [7] and references therein), which aims to mine the community structures in a graph. Essentially, it partitions vertices of a graph into disjoint or overlapping groups such that each group represents one community. Community detection is a one-time task, and the result is the same set of communities for different users and thus does not reflect users' personalized information. To remedy the non-personalization issue of community detection, there is a growing interest to search communities for user-given query vertices which facilitates a user-centric personalized search (*e.g.*, see the tutorial [9] and references therein). This querying problem is known as the *community search* problem. In principle the total number of distinct communities that are discoverable by community search can be much larger than n — the number of vertices

* The work was done while Mingshen Cai was with The University of Sydney

in the data graph — and even may be exponential, while most of the community detection methods can only identify at most n distinct communities. As a result, community search has many applications [6, 8, 11], such as advertisements targeting, recommendation in online social networks, and metabolic network analysis.

Given a data graph $G = (V, E)$ and a set of one or more query vertices $Q \subset V$, the problem of community search aims to find a connected subgraph of G that contains all query vertices, and is (most) cohesive. In the literature, the cohesiveness of a subgraph is usually measured by its minimum vertex degree (aka *k-core*) [2, 6, 12, 14], minimum number of triangles each edge participates in (aka *k-truss*) [10], or edge connectivity [3]. Among them, the minimum vertex degree-based cohesiveness measure is popularly used due to its simplicity and easy computability. However, there could be an exponential number of subgraphs of G that contain Q and have the same cohesiveness (*i.e.*, minimum vertex degree). In light of this, Cui et al. [6] reports an arbitrary one satisfying the requirements as the result, while Sozio and Gionis [14] introduces a distance threshold τ such that all vertices in the reported subgraph should be within distance τ from the query vertices. For the former, it is obviously not a good idea to report an arbitrary one since vertices in the result could be far away from the query vertices, while for the latter it may not be an easy task to specify an appropriate distance threshold τ.

In this paper, we formulate the *closest community search* problem. Specifically, the closest community of Q in G is the *connected* subgraph of G that *contains all query vertices*, *is most cohesive* (*i.e.*, with the largest possible minimum vertex degree), *is closest to* Q, and *is maximal*. Here, the closeness of a subgraph is measured by the largest value among the shortest distances between query vertices and other vertices in the subgraph. Compared to [6], closest community only includes vertices that are close and thus relevant to the query vertices Q. Compared to [14], closest community search does not require end-users to input a distance threshold τ, but automatically finds the subgraph that satisfies the smallest τ.

We show that the closest community of Q in G can be computed via a two-stage approach: (1) stage-I computes the maximal connected subgraph g_0 of G that contains Q and is most cohesive, and (2) stage-II iteratively removes from g_0 the vertex that is furthest to Q and subsequently also other vertices that violate the cohesiveness requirement due to the removal of their neighbors. Then, the last non-empty subgraph will be the closest community of Q in G. We first propose baseline approaches for the two stages that run in $O(n + m)$ and $O(n_0 \times m_0)$ time, respectively, where n (resp. n_0) and m (resp. m_0) are the number of vertices and edges in G (resp. g_0). Then, we develop techniques to improve the time complexities of the two stages into $O(n_0 + m_0)$ and $O(m_0 + n_0 \log n_0)$, respectively. As a result, we have the IndexedLO algorithm whose time complexity is $O(m_0 + n_0 \log n_0)$; this is near-optimal in the worst case, since the closest community of Q could be g_0 itself whose size is $O(n_0 + m_0)$. Nevertheless, in practice the closest community of Q could be much smaller than g_0, as it is expected that the closest community of Q usually contains only a few vertices that are close to Q. Thus, we further develop an algorithm CCS that has the same time complexity as IndexedLO but performs much better in practice. Our contributions are as follows.

- We formulate the closest community search problem (Section 2), and develop a Baseline approach (Section 4.1).

– We develop techniques to improve the time complexity and obtain the IndexedLO algorithm that runs in $O(m_0 + n_0 \log n_0)$ time, which is near-optimal (Section 4.2).
– We design a CCS algorithm that has the same time complexity as IndexedLO but runs faster in practice (Section 4.3).
– We conduct extensive empirical studies to demonstrate the efficiency and effectiveness of our techniques (Section 5).

Proofs of all lemmas and theorems are omitted due to limit of space.

2 Preliminaries

For presentation simplicity, we focus our discussions on an undirected and unweighted graph $G = (V, E)$,[3] where V and E are the vertex set and edge set of G, respectively. We use n and m to denote the number of vertices and the number of edges of G, respectively. We denote the undirected edge between vertices u and v by (u, v). The set of neighbors of a vertex u is denoted by $N(u) = \{v \in V \mid (u, v) \in E\}$, and the degree of u is denoted by $deg(u) = |N(u)|$. A path between u and v is $(v_0, v_1, \ldots, v_l)$ such that $v_0 = u$, $v_l = v$ and $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq l$; the length of the path is l. The distance between u and v, denoted $\delta(u, v)$, is defined as the shortest length among all paths between u and v.

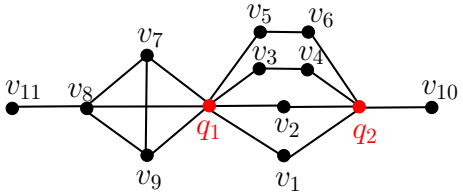


Fig. 1. An example graph

Given a set $Q \subset V$ of query vertices, the ***query distance*** of a vertex $v \in V$ is the maximum value among the distances between v and vertices of Q, *i.e.*, $\delta(Q, v) = \max_{u \in Q} \delta(u, v)$. For example, for the graph in Figure 1 and $Q = \{q_1, q_2\}$, $\delta(Q, v_1) = 1$, $\delta(Q, v_3) = 2$, and $\delta(Q, v_7) = 3$. Then, the query distance of a subgraph containing Q is the maximum query distance of its vertices. For example, the query distance of the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$ is 3, the query distance of the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_6\}$ is 2, and the query distance of the subgraph induced by vertices $\{q_1, q_2, v_1, v_2\}$ is 1.

Given a set Q of query vertices, we aim to find the closest community of Q in G. Intuitively, (1) the community should be connected and contain all query vertices, (2) the community should be cohesive such that the vertices are tightly connected, and (3) the community should be close to the query vertices such that it is relevant to the query. In this paper, for presentation simplicity we adopt the minimum vertex degree to measure the cohesiveness of a subgraph, while our techniques can be easily extended to other cohesiveness measures such as trussness [10] or edge connectivity [3]. We formally define the closest community as follows.

[3] The techniques we propose in this paper can be straightforwardly extended to directed graphs and weighted graphs.

Definition 1. *Given a graph $G = (V, E)$ and a set of query vertices $Q \subset V$, the **closest community** of Q in G is the **connected** subgraph g of G that contains Q and satisfies the following three conditions.*

1. ***Most Cohesive:** the minimum vertex degree of g is the largest among all connected subgraphs of G containing Q.*
2. ***Closest:** g has the smallest query distance among all subgraphs satisfying the above conditions.*
3. ***Maximal:** g is maximal.*

The closest community of $Q = \{q_1, q_2\}$ in Figure 1 is the subgraph induced by $\{q_1, q_2, v_1, v_2\}$, where the minimum vertex degree is 2 and the query distance is 1.

Problem Statement. Given a graph $G = (V, E)$ and a set of query vertices $Q \subset V$, we study the problem of efficiently computing the closest community of Q in G.

We assume that the input graph G is connected, and a tie-breaker (*e.g.*, vertex ID, or personalized PageRank values [2]) is introduced such that all vertices have different query distances. In the running examples, we use vertex ID for tie breaking.

3 General Idea

The general idea of our approaches is based on the concept of (k, d)-community.

Definition 2. *Given a graph $G = (V, E)$, a set of query vertices $Q \subset V$, and integers k and d, the (k, d)-**community** of Q in G is the **connected** subgraph g of G that contains Q and satisfies the following three conditions:*

1. ***Cohesive:** the minimum vertex degree of g is at least k.*
2. ***Close:** the query distance of g is at most d.*
3. ***Maximal:** g is maximal.*

It is easy to see that the closest community of Q in G is the (k, d)-community of Q in G that exists and has the largest k and the smallest d. Note that, the (k, d)-community of Q (if exists) is unique. Moreover, the (k, d)-communities of Q for different k values and different d values form hierarchical structures. That is, for a fixed d, the (k_1, d)-community of Q is a subgraph of the (k_2, d)-community of Q if $k_1 > k_2$; for a fixed k, the (k, d_1)-community of Q is a subgraph of the (k, d_2)-community of Q if $d_1 < d_2$. Thus, we can compute the closest community of Q by a two-stage framework.

Algorithm 1: TwoStageFramework

```
1 (k_Q, g_0) ← Stage-I(G, Q);
2 return Stage-II(g_0, Q, k_Q);
```

Stage-I. In the first stage, we compute the (k, ∞)-community of Q in G that has the largest k value. Denote this value of k as k_Q, and denote the (k_Q, ∞)-community of Q by g_0. Then, k_Q is the largest k value such that Q is in a connected component of the k-core of G, and g_0 is the connected component of the k_Q-core of G that contains Q. This is because, for any k, the (k, ∞)-community of Q is the connected component of the k-core that contains Q, where the *k-core* of a graph is the maximal subgraph g such that every

vertex in g has at least k neighbors in g. Note that, the k-core is unique. For the graph in Figure 1, the entire graph is a 1-core, the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$ is a 2-core, the subgraph induced by vertices $\{q_1, v_7, v_8, v_9\}$ is a 3-core, and there is no 4-core. Thus, for $Q = \{q_1, q_2\}$ in Figure 1, $k_Q = 2$ and the (k_Q, ∞)-community of Q is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$.

Stage-II. In the second stage, we compute the (k_Q, d)-community of Q that exists and has the smallest d value. As all vertices not in g_0 are guaranteed to be not in the (k_Q, d)-community of Q for any d, we can focus our computations on g_0. Thus, we iteratively reduce the graph g_0 to obtain the (k_Q, d)-community of Q with the next largest d value, and the final non-empty subgraph is the result. For example, the (k_Q, ∞)-community of $Q = \{q_1, q_2\}$ is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$. The next (k_Q, d)-communities that will be discovered are the subgraphs induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_8\}$, $\{q_1, q_2, v_1, v_2, \ldots, v_6\}$, $\{q_1, q_2, v_1, v_2, v_3, v_4\}$, and $\{q_1, q_2, v_1, v_2\}$, respectively, where the last one is the closest community of Q.

4 Our Approaches

We first propose a Baseline approach in Section 4.1, then improve its time complexity in Section 4.2, and finally improve its practical performance in Section 4.3.

4.1 A Baseline Approach

Baseline Stage-I: Baseline-S1**.** A naive approach for stage-I in Algorithm 1 would be iteratively computing the k-core of G for k values decreasing from n to 1, and stopping immediately if Q is contained in a connected component of the computed k-core. However, the worst-case time complexity will be quadratic to the input graph size, which is prohibitive for large graphs. To aim for a better time complexity, we propose to first compute the core number for all vertices, where the ***core number*** of a vertex u, denoted $\texttt{core}(u)$, is the largest k such that the k-core contains u. For the graph in Figure 1, $\texttt{core}(q_1) = \texttt{core}(v_7) = \texttt{core}(v_8) = \texttt{core}(v_9) = 3$, $\texttt{core}(v_{10}) = \texttt{core}(v_{11}) = 1$, and the core numbers of all other vertices are 2. Note that, the core number for all vertices in G can be computed by the *peeling algorithm* in linear time [1]. Then, the k-core of G is the subgraph induced by vertices whose core numbers are at least k [4]. Thus, we can compute k_Q, the largest k value such that Q is in a connected component of the k-core of G, by conducting a prioritized search from an arbitrary vertex of Q. That is, we grow the connected component from an arbitrary vertex of Q, and each time we include, into the connected component, the vertex that has the largest core number among all vertices that are connected to (a vertex of) the connected component. Once the connected component contains all vertices of Q, the minimum core number among all vertices of the connected component then is k_Q. The pseudocode of our baseline approach for stage-I is shown in Algorithm 2, denoted Baseline-S1.

Example 1. Consider $Q = \{q_1, q_2\}$ and the graph in Figure 1, and assume we conduct the prioritized search from q_1. The algorithm will first visit the vertices $\{q_1, v_7, v_8, v_9\}$ that have core numbers 3 and are connected to q_1. Then, the algorithm will visit a subset

Algorithm 2: Baseline-S1

Input: Graph $G = (V, E)$ and a set of query vertices $Q \subset V$
Output: k_Q and the (k_Q, ∞)-community of Q

```
1  Run the peeling algorithm of [1] to compute the core number for all vertices of G;
2  Initialize a priority queue Q to contain an arbitrary vertex of Q;
3  k_Q ← n;
4  while not all vertices of Q have been visited do
5      u ← pop the vertex with the maximum core number from Q;
6      Mark u as visited;
7      if core(u) < k_Q then k_Q ← core(u) ;
8      for each neighbor v ∈ N(u) do
9          if v is not in Q and has not been visited then Push v into Q;
10 g_0 ← the connected component of the k_Q-core of G that contains Q;
11 return (k_Q, g_0);
```

Algorithm 3: Baseline-S2

Input: A set of query vertices $Q \subset V$, an integer k_Q, and a graph g_0 that contains Q and has minimum vertex degree k_Q
Output: Closest community of Q

```
1  Compute the query distance for all vertices of g_0;
2  i ← 0;
3  while true do
4      u ← the vertex in g_i with the largest query distance;
5      g_{i+1} ← the connected component of the k_Q-core of g_i\{u} that contains Q;
6      if g_{i+1} = ∅ then break ;
7      else i ← i + 1 ;
8  return g_i;
```

of the vertices $\{v_1, v_2, q_2, v_3, v_4, v_5, v_6\}$ that have core numbers 2. Thus, $k_Q = 2$, and the (k_Q, ∞)-community of Q is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$.

The correctness of Baseline-S1 (Algorithm 2) can be verified from the definitions of k_Q and (k_Q, ∞)-community, and the property that the k-core of G is the subgraph induced by vertices whose core numbers are at least k. The time complexity of Baseline-S1 is proved by the theorem below.

Theorem 1. *The time complexity of* Baseline-S1 *is* $O(n + m)$ *where* n *and* m *are the number of vertices and the number of edges of* G*, respectively.*

Baseline Stage-II: Baseline-S2**.** In the second stage, we aim to iteratively reduce the graph g_0, obtained from the first stage, to compute the (k_Q, d)-community of Q with the next largest d value. Intuitively, the vertex that is furthest from the query vertices in g_0 will not be in the next (k_Q, d)-community; thus, we can remove this vertex from g_0 and then reduce the resulting graph to the connected component of the k_Q-core that contains Q. The final non-empty subgraph will be the closest community of Q.

The pseudocode of our baseline approach for stage-II is shown in Algorithm 3, denoted Baseline-S2. Line 1 computes the query distance for all vertices of g_0. Then, at Lines 4–5, we iteratively remove from g_i the vertex that is furtherest from the query vertices (*i.e.*, has the largest query distance), and compute the connected component g_{i+1} of the k_Q-core of the graph $g_i \backslash \{u\}$ that contains Q. If there is no such g_{i+1} (*i.e.*, $g_{i+1} = \emptyset$), then g_i is the closest community of Q and the algorithm terminates (Line 6). Otherwise, we increase i and continue the next iteration (Line 7).

Example 2. Continue Example 1. $k_Q = 2$ and g_0 is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$. v_9 is the vertex that has the largest query distance in g_0. Then, g_1 is computed as the connected component of the k_Q-core of $g_0 \backslash \{v_9\}$ that contains Q, which is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_8\}$. v_8 is the vertex that has the largest query distance in g_1, and g_2 is computed as the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_6\}$. Similarly, g_3 is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, v_3, v_4\}$, and g_4 is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2\}$. Now, v_2 is the vertex that has the largest query distance in g_4. After removing v_2 from g_4, the k_Q-core of $g_4 \backslash \{v_2\}$ does not contain all vertices of Q. Thus, the algorithm terminates, and g_4 is the closest community of Q.

The correctness of Baseline-S2 (Algorithm 3) is straightforward. The time complexity of Baseline-S2 is proved by the theorem below.

Theorem 2. *The time complexity of* Baseline-S2 *is* $O(n_0 \times m_0)$ *where* n_0 *and* m_0 *are the number of vertices and the number of edges of* g_0*, respectively.*

As a result, the total time complexity of Baseline that first runs Baseline-S1 and then runs Baseline-S2 is $O(n + m + n_0 \times m_0)$. Note that, n_0 and m_0 in the worst case can be as large as n and m, respectively. Thus, the time complexity of Baseline is quadratic to the input graph size in the worst case.

4.2 Improving the Baseline Approach

The Baseline approach proposed in Section 4.1 is too slow to process large graphs due to its quadratic time complexity $O(n + m + n_0 \times m_0)$. In this subsection, we propose techniques to improve the time complexity for the two stages of Baseline.

LinearOrder-S2: Improving Baseline-S2. As shown by our empirical studies in Section 5, Baseline-S2 takes more time than Baseline-S1 in Baseline. Thus, we first aim to reduce the time complexity of stage-II of Baseline, *i.e.*, Baseline-S2. The main cost of Baseline-S2 comes from Line 5 of Algorithm 3 that in each iteration computes the connected component of the k_Q-core of $g_i \backslash \{u\}$ that contains Q. To avoid this quadratic cost, we do not immediately search for the connected component of the k_Q-core that contains Q in each iteration. Instead, we separate the computation into two steps: step-1 builds the entire hierarchical structure for the (k_Q, d)-communities of Q for all different d values by ignoring the connectedness requirement, and step-2 searches for the connected (k_Q, d)-community of Q that has the smallest d value. This is based on the fact that the (k_Q, d_1)-community of Q is a subgraph of the (k_Q, d_2)-community of Q if $d_1 < d_2$.

To build the hierarchical structure for the (k_Q, d)-communities of Q for all different d values, we propose to compute a linear ordering for vertices of g_0; recall that g_0 is the (k_Q, ∞)-community of Q. Specifically, we encode the hierarchical structure by a linear ordering `seq` of vertices of g_0 and a subsequence `targets` of `seq`, such that *there is one-to-one correspondence between each* (k_Q, d)*-community for a different* d *value and each suffix of* `seq` *that starts from a vertex of* `targets`. Figure 2 shows such an example. Note that, to be more precise, we here refer to a variant of (k_Q, d)-community that does not necessarily to be connected, *i.e.*, we remove the connected requirement from Definition 2. To compute the linear ordering, we iteratively remove from g_0 the vertex that has the largest query distance (and add it to the end of `seq` and `targets`), and then subsequently remove from g_0 all vertices that violate the k_Q-core requirement (and add them to the end of `seq`).

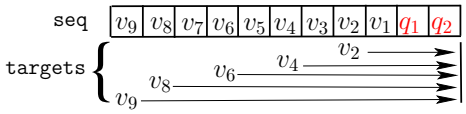


Fig. 2. Hierarchical structure of (k_Q, d)-communities for all different d values

Given `seq` and a vertex $u \in$ `seq`, let $\texttt{seq}_u$ denote the suffix of `seq` that starts from u. Then, the closest community of Q will be the connected component, of the subgraph induced by $\texttt{seq}_u$, containing Q, where u is the right-most vertex of `targets` such that Q is connected in the subgraph induced by $\texttt{seq}_u$. For example, in Figure 2, the closest community of $Q = \{q_1, q_2\}$ simply is the subgraph induced by $\texttt{seq}_{v_2} = \{v_2, v_1, q_1, q_2\}$. It is worth mentioning that, in general Q may be *disconnected* in the subgraph $\texttt{seq}_v$ where v is the last vertex of `targets`. This is because we do not check the connectedness of Q during the computation of `seq` and `targets` for the sake of time complexity. To get the closest community of Q from `seq` and `targets`, we can use a disjoint-set data structure [5] to incrementally maintain the connected components of the subgraphs of g_0 induced by vertices of suffices of `seq`.

The pseudocode of our improved algorithm for stage-II is shown in Algorithm 4, denoted LinearOrder-S2. Lines 1–14 compute the hierarchical structure for the (k_Q, d)-communities of Q for all different d values, and Lines 15–21 find the closest community of Q from the hierarchical structure. Note that, in order to efficient check whether Q is entirely contained in a single set of the disjoint-set data structure $\mathcal{S}$ at Line 21, we maintain a counter for each set recording the number of Q's vertices that are in this set. The counter can be maintained in constant time after each union operation of Line 20, and Line 21 can be tested in constant time; we omit the details.

Example 3. Reconsider Example 2. $k_Q = 2$ and g_0 is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, \ldots, v_9\}$. Firstly, v_9 is the vertex with the largest query distance, so v_9 is removed from the graph and is appended to both `seq` and `targets`; no other vertices are removed as a result of the k_Q-core requirement. Secondly, v_8 is the vertex with the largest query distance, and it is removed from the graph and is appended to both `seq` and `targets`; subsequently, v_7 is also removed from the graph and is appended to `seq` due to the violation of the k_Q-core requirement. So on so forth. The final results are $\texttt{seq} = (v_9, v_8, v_7, v_6, v_5, v_4, v_3, v_2, v_1, q_1, q_2)$ and $\texttt{targets} = (v_9, v_8, v_6, v_4, v_2)$ as shown in Figure 2. As the subgraph induced by $\texttt{seq}_{v_2} = \{v_2, v_1, q_1, q_2\}$ is connected and con-

Algorithm 4: LinearOrder-S2

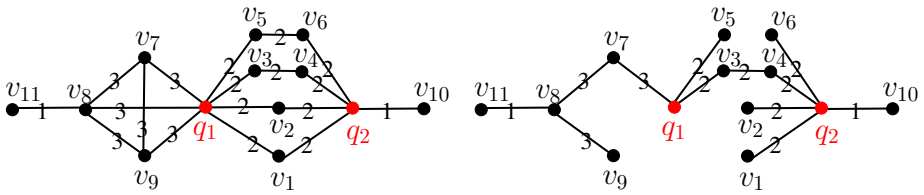
```
   /* Compute the hierarchical structure for the (k_Q, d)-communities    */
 1 Compute the query distance for all vertices of g_0;
 2 Sort vertices of g_0 in decreasing order with respect to their query distances;
 3 seq ← ∅; targets ← ∅;
 4 g' ← g_0; deg(u) ← the degree of u in g' for all vertices u ∈ g';
 5 while g' is not empty do
 6     u ← the vertex in g' with the largest query distance;
 7     if Q ∩ seq = ∅ then Append u to targets;
 8     Q ← {u};                          /* Q is a queue */;
 9     while Q ≠ ∅ do
10         Pop a vertex v from Q, and append v to seq;
11         for each neighbor w of v in g' do
12             deg(w) ← deg(w) − 1;
13             if deg(w) = k_Q − 1 then Push w into Q ;
14         Remove v from g';
   /* Search for the closest community of Q                               */
15 Initialize an empty disjoint-set data structure S;
16 for each vertex u ∈ targets in the reverse order do
17     for each vertex v ∈ seq between u (inclusive) and the next target vertex (exclusive) do
18         Add a singleton set for v into S;
19         for each neighbor w of v in g_0 do
20             if w ∈ S then Union v and w in S ;
21     if Q is entirely contained in a single set of S then break ;
22 return all vertices in the set of S that contains Q;
```

tains both q_1 and q_2, the closest community of $Q = \{q_1, q_2\}$ is the subgraph induced by vertices $\texttt{seq}_{v_2}$.

Theorem 3. *The time complexity of* LinearOrder-S2 *is* $O(m_0 + n_0 \log n_0)$.

Indexed-S1**: Improving** Baseline-S1**.** By improving Baseline-S2 to LinearOrder-S2 which has a time complexity of $O(m_0 + n_0 \log n_0)$, stage-I (*i.e.*, Baseline-S1), which processes the entire input graph and takes $O(n + m)$ time, now becomes the bottleneck. Thus, in the following we propose to utilize an index structure to improve Baseline-S1.

Baseline-S1 computes two things: k_Q and g_0 where g_0 is the connected component of the k_Q-core of G that contains Q. We first discuss how to efficiently get g_0 from G based on an index structure if k_Q is known. Recall that, for any k, the k-core of G is the subgraph induced by vertices whose core number are at least k. Thus, *in the index structure, we precompute and store the core number for all vertices of G, and moreover we sort the neighbors of each vertex in the graph representation in decreasing order with respect to their core numbers*. Thus, to search for g_0, we can conduct a pruned breath-first search which starts from an arbitrary vertex of Q and visits only vertices whose core numbers are at least k_Q. It can be verified that, the vertices and edges visited during the pruned breath-first search form the g_0.



(a) Weighted graph (b) Maximum spanning tree

Fig. 3. Weighted graph and maximum spanning tree

Secondly, to efficiently compute k_Q, we further maintain a maximum spanning tree of the edge-weighted graph of G where the weight of edge (u, v) equals $\max\{\texttt{core}(u), \texttt{core}(v)\}$. For example, the weighted graph and the maximum spanning tree for the graph in Figure 1 are shown in Figure 3(a) and Figure 3(b), respectively. It can be verified by a similar argument as in [3] that k_Q equals the minimum weight among all edges in the paths between q_1 and q_i for $2 \leq i \leq |Q|$ in the maximum spanning tree, where $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$. For example, the path between q_1 and q_2 in Figure 3(b) is (q_1, v_3, v_4, q_2) and $k_{\{q_1,q_2\}} = 2$. Note that, by further processing the maximum spanning tree using the techniques in [3], k_Q can be computed in $O(|Q|)$ time; we omit the details.

Algorithm 5: Indexed-S1

1 Compute k_Q based on the index $\mathcal{I}$;
2 Conduct a pruned breadth-first search on G by starting from an arbitrary vertex of Q and visiting only vertices whose core numbers are at least k_Q;
3 $g_0 \leftarrow$ the subgraph of G induced by vertices visited at Line 2;
4 **return** (k_Q, g_0);

The pseudocode of our index-based algorithm for stage-I is shown in Algorithm 5, which is self-explanatory.

Theorem 4. *The time complexity of* Indexed-S1 *is* $O(n_0 + m_0)$.

By invoking Indexed-S1 for stage-I and LinearOrder-S2 for stage-II, we get an algorithm that computes the closest community of Q in $O(m_0 + n_0 \log n_0)$ time; denote this algorithm as IndexedLO.

4.3 The CCS Approach

The time complexity of IndexedLO is near-optimal in the worst case, because the closest community of Q could be g_0 itself whose size is $O(n_0 + m_0)$. Nevertheless, the closest community of Q could be much smaller than g_0 in practice, as it is expected that the closest community of Q usually contains only a few vertices that are close to Q. Motivated by this, in this section we propose a CCS approach to improve the performance of IndexedLO in practice. The general idea of CCS follows the framework of [2]. That is, instead of first computing g_0 — the connected component of the k_Q-core of G that contains Q — and then shrinking g_0 to obtain the closest community of Q as shown in Algorithm 1, we start from working on a small subgraph containing Q and then progressively expand it by including next few further away vertices. As the

vertices are added to the working subgraph in increasing order according their query distances, once the working subgraph has a connected k_Q-core that contain all vertices of Q, the closest community of Q can be computed from the working subgraph by invoking LinearOrder-S2.

Algorithm 6: CCS

Input: Graph $G = (V, E)$, a set of query vertex Q, and an index $\mathcal{I}$
Output: Closest community of Q

```
1  Compute k_Q based on the index I;
2  h_0 ← the subgraph of G induced by Q;
3  i ← 0; g ← ∅;
4  while true do
5      g' ← the connected component of the k_Q-core of h_i that contains Q;
6      g ← LinearOrder-S2(Q, k_Q, g');
7      if g = ∅ then
8          i ← i + 1; h_i ← h_{i−1};
9          while h_i ≠ G and the size of h_i is less than twice of h_{i−1} do
10             Get the next vertex u that has the smallest query distance;
11             Add to h_i the vertex u and its adjacent edges to existing vertices of h_i;
12     else break;
13 return g;
```

The pseudocode of CCS is shown in Algorithm 6. We first compute k_Q based on the index $\mathcal{I}$ (Line 1), and initialize the working subgraph h_0 to be the subgraph of G induced by Q (Line 2). Then, we go to iterations (Lines 5–12). In each iteration, we try to compute the closest community of Q in h_i by invoking LinearOrder-S2 on the connected component of the k_Q-core of h_i that contains Q (Lines 5–6). Let g be the result. If g is not empty, then it is guaranteed to be the closest community of Q in G (Line 12). Otherwise, the current working subgraph h_i does not include all vertices of the closest community of Q, and we need to grow the working subgraph (Lines 8–11). To grow the working subgraph, we (1) include vertices in increasing order according to their query distances, and (2) grow the working subgraph exponentially at a rate of two. Here, the size of a graph is measured by the summation of its number of vertices and its number of edges. We will prove shortly that the time complexity of this strategy will be $O(m_0 + n_0 \log n_0)$ in the worst case. Note that, if we grow the working subgraph at the rate of adding one vertex, then it is easy to see that the time complexity would be quadratic (*i.e.*, $O(n_0 \times m_0)$).

Example 4. Reconsider $Q = \{q_1, q_2\}$ and the graph in Figure 1, and recall that the vertices in increasing query distance order are $q_1, q_2, v_1, v_2, \ldots, v_{11}$. $k_Q = 2$. The initial working subgraph h_0 consists of vertices q_1 and q_2 and is of size 2, as shown in Figure 4. The second working subgraph h_1 is of size 5, as shown in Figure 4. h_1 does not have a 2-core, and we continue growing the working subgraph. The third working subgraph h_2 is the subgraph induced by vertices $\{q_1, q_2, v_1, v_2, v_3\}$, and g is computed as the subgraph induced vertices $\{q_1, q_2, v_1, v_2\}$ which is the closest community of Q. Thus, the algorithm terminates and reports g as the closest community of Q.

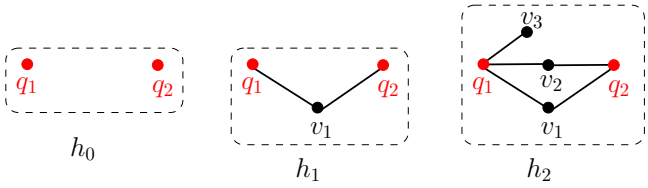


Fig. 4. Running example of CCS

Although CCS may need to process many subgraphs of g_0, we prove in the theorem below that its worst-case time complexity is $O(m_0 + n_0 \log n_0)$.

Theorem 5. *The worst-case time complexity of* CCS *is* $O(m_0 + n_0 \log n_0)$.

5 Experiments

In this section, we conduct extensive empirical studies to evaluate the performance of our algorithms on real-world graphs. We evaluate the following four algorithms.

- Baseline, which invokes Baseline-S1 (Algorithm 2) for stage-I and Baseline-S2 (Algorithm 3) for stage-II.
- LinearOrder, which invokes Baseline-S1 (Algorithm 2) for stage-I and LinearOrder-S2 (Algorithm 4) for stage-II.
- IndexedLO, which invokes Indexed-S1 (Algorithm 5) for stage-I and LinearOrder-S2 (Algorithm 4) for stage-II.
- CCS (Algorithm 6).

All the algorithms are implemented in C++.

Datasets. We use six real graphs that are downloaded from the Stanford Network Analysis Platform[4] in our evaluation. Statics of these graphs are shown in Table 1, where $\texttt{core}_{max}$ denotes the maximum core number among vertices in a graph.

Table 1. Statistics of Real Graphs

Graphs	n	m	$\texttt{core}_{max}$
Email	36,692	183,831	43
Amazon	334,863	925,872	6
DBLP	317,080	1,049,866	113
Youtube	1,134,890	2,987,624	51
LiveJournal	3,997,962	34,681,189	360
Orkut	3,072,441	117,185,083	253

Setting. We compare the performance of the algorithms by measuring their query processing time. The reported time includes all the time that is spent in computing the closest community for a query, except the I/O time for reading the graph from disk to main memory. All experiments are conducted on a machine with 2.9 GHz Intel Core i7 CPU and 16GB main memory.

[4] http://snap.stanford.edu/

5.1 Experimental Results

In this testing, the query vertices for a graph are randomly selected from its 5-core. The total running time of the four algorithms on the six graphs is shown in Figure 5. We can see that the algorithms in sorted order from slowest to fastest are Baseline, LinearOrder, IndexedLO, and CCS. This results align with our theoretical analysis. That is, the time complexities of these four algorithms are $O(n+m+n_0 \times m_0)$, $O(n+m+m_0+n_0 \log n_0)$, $O(m_0+n_0 \log n_0)$, and $O(m_0+n_0 \log n_0)$, respectively, where n_0 usually is much smaller than n, and m_0 usually is much smaller than m. Baseline cannot finish with 10 minutes, except for the two small graphs Email and Amazon. The improvement of CCS over LinearOrder is up-to 607 times. The improvement of CCS over IndexedLO is up-to 148 times, despite having the same worst-case time complexity.

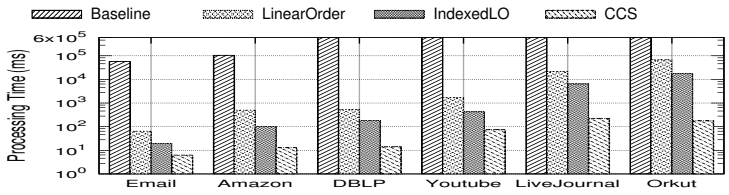


Fig. 5. Total running time of the algorithms (ms)

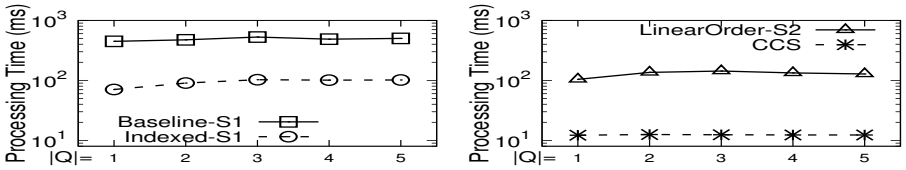
To get a more detailed analysis of the algorithms, we separate the total running time into the running time of stage-I and the running time of stage-II, for each algorithm. The results are shown in Table 2. Recall that the algorithm for stage-I of LinearOrder is the same as that of Baseline, and the algorithm for stage-II of IndexedLO is the same as that of LinearOrder. We can see that Indexed-S1 (used in stage-I of IndexedLO) significantly improves upon Baseline-S1 (used in stage-I of Baseline and LinearOrder) as a result of the index-based approach, and the improvement is more than one order of magnitude. Regarding stage-II, we can see that LinearOrder-S2 (used in LinearOrder and IndexedLO) significantly improves upon Baseline-S2 (used in Baseline) due to the improved time complexity from quadratic (specifically, $O(n_0 \times m_0)$) to near-linear (specifically, $O(m_0 + n_0 \log n_0)$).

Table 2. Stage-I and stage-II time of the algorithms (ms)

Graphs	Baseline		LinearOrder		IndexedLO		CCS
	Stage-I	Stage-II	Stage-I	Stage-II	Stage-I	Stage-II	Total
Email	56.01	56,322	56.10	12.66	7.48	12.22	6.31
Amazon	445.14	104,581	445.03	68.52	35.89	68.24	12.85
DBLP	493.02	>10 min	493.95	120.84	59.5	120.77	14.00
Youtube	1,385	>10 min	1,385	291	128.49	291	74.07
LiveJournal	19,343	>10 min	19,343	4,178	2374.39	4,178	226.74
Orkut	58,815	>10 min	58,815	13,620	4295.06	13,620	176.63

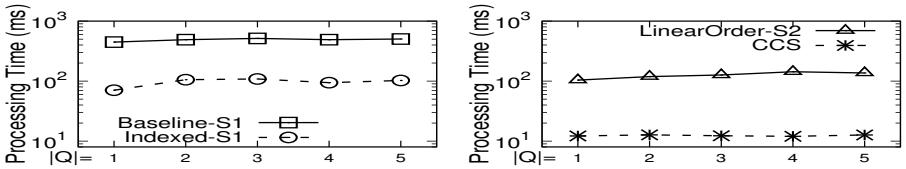
Now, let's compare the two stages within each algorithm. We can see that for Baseline, stage-II (Baseline-S2 with time complexity $O(n_0 \times m_0)$) dominates stage-I (Baseline-S1 with time complexity $O(n+m)$) due to the quadratic time complexity of stage-II, and stage-II takes more than 10 minutes for graphs DBLP, Youtube, LiveJournal, and Orkut. This motivates us to improve Baseline-S2 to LinearOrder-S2 that runs

in $O(m_0 + n_0 \log n_0)$ time, which leads to our second algorithm LinearOrder. Due to the improved time complexity of LinearOrder-S2, we can see that stage-I of LinearOrder (*i.e.*, Baseline-S1) now dominates due to processing the entire input graph. This motivates us to utilize an index structure that is built offline to improve the online query processing time, which results in our third algorithm IndexedLO that has a time complexity of $O(m_0 + n_0 \log n_0)$. The time complexity of IndexedLO is near-linear to the size of the initial graph g_0, which increases along with the input graph size. Thus, the processing time of IndexedLO increases significantly for large graphs (*e.g.*, Orkut), which motivates us to design the CCS algorithm. We can see that CCS significantly outperforms both stages of IndexedLO, and the processing time of CCS on large graphs increases much slower than that of IndexedLO.



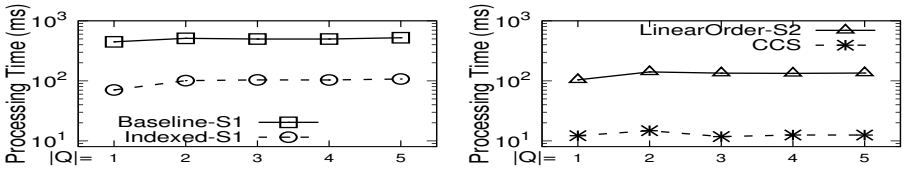
(a) Stage-I time (vary $|Q|$) (b) Stage-II time (vary $|Q|$)

Fig. 6. DBLP with distance 1 among query vertices



(a) Stage-I time (vary $|Q|$) (b) Stage-II time (vary $|Q|$)

Fig. 7. DBLP with distance 2 among query vertices



(a) Stage-I time (vary $|Q|$) (b) Stage-II time (vary $|Q|$)

Fig. 8. DBLP with distance 3 among query vertices

Vary Query Size. In this testing, we evaluate the impact of the number of query vertices on the performance of the algorithms. In particular, we separately consider the algorithms for stage-I and for stage-II. For stage-I, we compare Indexed-S1 with Baseline-S1, and for stage-II, we compare CCS with LinearOrder-S2. Note that, (1) we do not include Baseline-S2 because it is too slow as shown in Table 2, and (2) we compare CCS with LinearOrder-S2 although the reported time of CCS is its total processing time. We vary the number of query vertices $|Q|$ from 1 to 5. For each query size, we generate three sets of queries such that the distances among the query vertices are 1, 2, and 3, respectively. The results on DBLP are shown in Figure 6, Figure 7 and Figure 8. We can see that the processing time for both stages increases slightly when the number of query vertices increases. Nevertheless, this is not significant, and CCS still significantly outperforms the other algorithms.

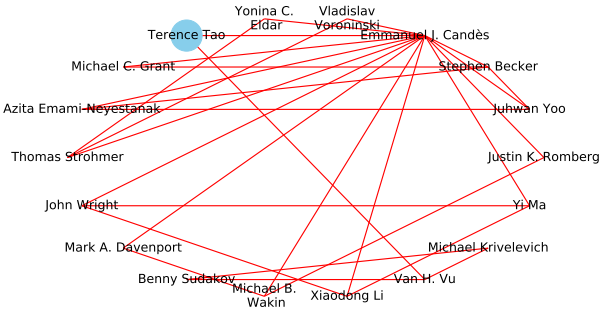


Fig. 9. Closest community search for "Terence Tao"

Case Study. We conduct a case study for the closest community search on the DBLP coauthor graph, which is built based on the dataset *BigDND: Big Dynamic Network Data*[5] extracted from DBLP. The dataset includes all author publication information stored in DBLP up-to October 2014. In our coauthor graph, each vertex represents one author, and there is an edge between u and v if they have published at least 3 papers together. The final coauthor graph has $367,202$ vertices and $821,205$ edges.

In the case study, we search for the closest community of "Terence Tao", an Australian-American mathematician who is one of the Fields Medal recipient in 2006. The result is shown in Figure 9, which has 18 authors. Terence Tao has published more than 3 papers together with Van H. Vu and Emmanuel J. Candes. The most cited paper of Tao's is *Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information*, which is a collaborated work with Emmanuel J. Candes and Justin Romberg. Emmanuel J. Candes also has over 80 papers recorded in our dataset, so there are a lot of scholars that coauthor with him as well, as shown in Figure 9, there are 13 nodes that represent his coauthors.

6 Related Works

Community Search. Given a set of one or more query vertices Q, community search aims to find cohesive subgraphs that contain Q. In the literature, the cohesiveness of a subgraph is usually measured by minimum degree (aka *k-core*) [6, 14], minimum number of triangles each edge participates in (aka *k-truss*) [10], or edge connectivity [3]. In this paper, we use the minimum degree-based cohesiveness measure in our closest community search problem. The technique of [6] cannot be used for closest community search as it inherently ignores the distance between vertices. Although the technique of [14] can be extended to compute the closest community which corresponds to our Baseline approach, it is infeasible for large graphs as shown by our experiments. On the other hand, the closest community search problem is recently studied in [10] which uses the trussness-based cohesiveness measure, the general idea of the algorithm in [10] is similar to our combination of Indexed-S1 and Baseline-S2. We have shown that Baseline-S2 cannot process large graphs due to its quadratic time complexity. In

[5] http://projects.csail.mit.edu/dnd/

order to process large graphs, heuristic techniques (such as bulk deletion and local exploration) are used in [10] which destroys the *exactness*; that is, the computed result may be not the closest community. It will be an interesting future work to extend our implementation to handle the query of [10].

Influential Community Search. The problem of influential community search is recently investigated in [2, 12]. Influential community search does not have query vertices but considers a vertex-weighted input graph, and aims to find top subgraphs that have minimum vertex degree k and have largest minimum vertex weight. Due to not having query vertices and not aiming for most cohesive subgraph, the algorithms in [2, 12] cannot be used to process closest community search queries.

7 Conclusion

In this paper, we formulated the closest community search problem based on the minimum degree-based cohesiveness measure. We firstly developed a Baseline algorithm, and then progressively improved it to IndexedLO, and CCS. We theoretically analyzed their time complexities, and conducted extensive empirical studies to evaluate the efficiency and effectiveness of the algorithms.

References

1. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. CoRR **cs.DS/0310049** (2003)
2. Bi, F., Chang, L., Lin, X., Zhang, W.: An optimal and progressive approach to online search of top-k influential communities. PVLDB **11**(9), 1056–1068 (2018)
3. Chang, L., Lin, X., Qin, L., Yu, J.X., Zhang, W.: Index-based optimal algorithms for computing steiner components with maximum connectivity. In: Proc. of SIGMOD'15 (2015)
4. Chang, L., Qin, L.: Cohesive Subgraph Computation over Large Sparse Graphs. Springer Series in the Data Sciences (2018)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
6. Cui, W., Xiao, Y., Wang, H., Wang, W.: Local search of communities in large graphs. In: Proc. of SIGMOD'14. pp. 991–1002 (2014)
7. Fortunato, S.: Community detection in graphs. Physics reports **486**(3-5), 75–174 (2010)
8. Guimerà, R., Nunes Amaral, L.A.: Functional cartography of complex metabolic networks. Nature **433**(7028), 895–900 (2 2005)
9. Huang, X., Lakshmanan, L.V.S., Xu, J.: Community search over big graphs: Models, algorithms, and opportunities. In: Proc. of ICDE'17. pp. 1451–1454 (2017)
10. Huang, X., Lakshmanan, L.V.S., Yu, J.X., Cheng, H.: Approximate closest community search in networks. Proc. VLDB Endow. **9**(4), 276–287 (Dec 2015)
11. Li, J., Wang, X., Deng, K., Yang, X., Sellis, T., Yu, J.X.: Most Influential Community Search over Large Social Networks. In: Proc. of ICDE'17. pp. 871–882 (2017)
12. Li, R.H., Qin, L., Yu, J.X., Mao, R.: Influential community search in large networks. Proceedings of the VLDB Endowment **8**(5), 509–520 (1 2015)
13. Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly Media, Inc. (2013)
14. Sozio, M., Gionis, A.: The community-search problem and how to plan a successful cocktail party. In: Proc. of KDD'10. p. 939 (2010)