Cohesiveness-aware Hierarchical Compressed Index for Community Search on Attributed Graphs

ACM SIGMOD Submission ID 646

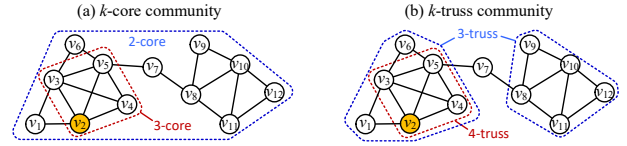
Abstract

Community search on attributed graphs (CSAG) is a fundamental research topic in graph data mining. Given an attributed graph G and a query node q, CSAG aims to find a structural- and attribute-cohesive subgraph from G that contains q. Exact methods based on graph traversal are time-consuming, especially for large graphs. Approximate methods improve the efficiency by pruning the search space with heuristic rules but still take hundreds of milliseconds to tens of seconds to respond, hindering the use of CSAG in time-sensitive applications. Moreover, one method's pruning strategies are tailored to its specific problem setting and community cohesiveness metrics, often being tightly coupled with its algorithmic paradigm. This makes them difficult to extend to other methods. This motivates us to study a general approach to accelerate various CSAG methods. We first present a proximity graph-based, cohesiveness-aware hierarchical index that accommodates different cohesiveness metrics. Then, we present two optimizations to enhance the index's navigability and reliability. Finally, we design a compressed storage structure for space-efficient indexing. We integrate the proposed index with mainstream CSAG methods and conduct experiments on real-world datasets. The results show that all methods using our index achieve an average 30.7× speedup while maintaining a comparable or even better attribute cohesiveness.

1 Introduction

Attributed graphs are widely used to model real-world information networks in diverse domains, such as social networks, collaboration networks, and knowledge graphs [18, 35, 51]. Nodes in these graphs represent entities with attributes, while edges signify relationships between entities. For instance, users in social networks have hobbies and other personal information; authors in collaboration networks have research topics, h-index, and # citations. Given an attribute graph G and a query node q, community search on attributed graphs (CSAG) aims to find a cohesive community $C \subseteq G$ that contains q, with cohesiveness measured in both structural and attribute aspects. Community search (CS) can be used in various applications like advertising [58], expert finding [53], and recommendation [23, 48].

CS on attributed graphs. Many efforts have been made in this research area, which can be categorized based on how they define community cohesiveness from both structural and attribute aspects. Structural cohesiveness. Numerous models like k-core [5, 16], k-truss [27, 29, 35], and k-clique [15, 43] are proposed. In terms of the ability to measure structural cohesiveness, [21] ranks them as k-core $\preceq$ k-truss $\preceq$ k-clique. Here, $A \preceq B$ indicates a model B is more cohesive than model A. In practice, k-core and k-truss are popular models in the literature. Figure 1 illustrates an attributed graph, where each node represents an author with several research topics (as textual attributes) and ⟨# publications, #h-index⟩ (as numerical attributes). Given a query node v_2, we show the 2-core (blue) and 3-core (red) communities of q in Figure 1(a). A k-core is a subgraph of which each node has a degree $\geq k$. Besides, a k-truss is a subgraph of which each edge is involved in at least $k-2$ triangles. We show the 3-truss (blue) and 4-truss (red) containing q in Figure 1(b). The larger the k, the more structural cohesiveness a community has.



Node	Attributes
v_5	<KG, NLP, DM> <131, 30>
v_3	<ML, DB, DM> <140, 31>

Node	Attributes
v_4	<IR, DB, DM> <138, 32>
v_2	<AI, DB, DM> <142, 33>

$Jdis(v_5,v_2) = 4/5$ $Edis(v_5,v_2) = 11.4$
$Jdis(v_4,v_2) = 1/2$ $Edis(v_4,v_2) = 4.12$
$Jdis(v_3,v_2) = 1/2$ $Edis(v_3,v_2) = 2.83$

Figure 1: Example of k-core and k-truss community models

Attribute cohesiveness. A community's attribute cohesiveness is defined based on the attribute distance of nodes. For textual attributes, metrics such as Jaccard distance, # shared attributes, and attribute coverage are often used [20, 28, 35, 50, 51, 58]. For numerical attributes, Euclidean distance and Manhattan distance [35, 50, 51] are often used. For instance, in Figure 1, v_5 is the most dissimilar to v_2 as it has the largest Jaccard distance in research topics and Euclidean distance in numerical attributes. Although v_3 and v_4 share the research topics {DB, DM} with v_2, v_3 is more similar to v_2 due to the smaller Euclidean distance in numerical attributes. One can define the attribute cohesiveness using these metrics individually or in combination [51], and form an optimization problem to find the most attribute-cohesive community. For example, the latest work SEA [51] defines attribute cohesiveness as the average attribute distance (weighted sum of Jacard distance and Manhattan distance) of all nodes to q and seeks to minimize the average attribute distance.

Motivation and challenges. Table 1 summarizes recent representative works of CS on attributed graphs. Regardless of how the CS problem is defined, the key idea for solving it is to design either a peeling-based or an expansion-based paradigm based on graph traversal. Since graph traversal is computational expensive, especially on large graphs, all works face efficiency challenges. Specifically, VAC [35], ATC [28], and SEA [51] are NP-hard problems, making exact algorithms impractical. So, approximate algorithms relying on pruning strategies are essential. Whether employing exact or approximate algorithms, their execution time ranges from hundreds of milliseconds to tens of seconds. Figure 2 shows the average runtime of four algorithms over 300 random queries across various datasets. In LiveJournal (a million-scale social network), VAC takes 16 secs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '25, June 22–27, 2018, Berlin, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Table 1: Summarization of four representative works

Methods ↓	Structure		Attribute		Hardness	Solution	Paradigm
	k-core	k-truss	Textual	Numerical			
VAC [35]		✓	✓	✓	NP-hard	Approx	Peeling
ATC [28]		✓	✓		NP-hard	Approx	Expansion
SEA [51]	✓		✓	✓	NP-hard	Approx	Sampling+Peeling
rKACS [58]		✓	✓		-	Exact	Expansion

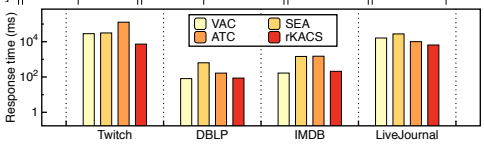


Figure 2: Runtime of existing methods on different datasets

to respond, even the fastest rKACS still requires 6 secs. This poses challenges for their use in time-sensitive, interactive applications, e.g., low-latency, high-frequency recommendation systems [31, 47] and community-enhanced GraphRAG-based Q&A systems [17].

Designing a general method to accelerate various algorithms for CS on attributed graphs is crucial and practically significant, but achieving this is non-trivial for two reasons. First, pruning strategies are typically designed to improve efficiency for specific problem settings and community cohesiveness metrics. Adapting one method's pruning strategies to other methods proves difficult. This suggests that designing a pruning strategy adaptable to diverse problems is challenging. Second, pruning strategies are often tightly coupled with the algorithmic paradigms, applying them to different algorithms requires refactoring of the algorithmic process, incurring high adaptation costs. This prompts us to seek a new approach: enhancing efficiency by reducing the search space through external mechanisms decoupled from the original algorithmic paradigm.

Our solution and contributions. Intuitively, if a node and the query node q belong to the same community, they must come from the same k-core or k-truss (i.e., with the same level of structural cohesiveness) and be similar to q in attributes (i.e., with a small attribute distance). Thus, running the algorithm on a small subgraph composed of promising nodes that share the same structural cohesiveness and similar attributes w.r.t. q, can efficiently yield a feasible community. This motivates us to present a Cohesiveness-aware Hierarchical Compressed Index (CHC-Index), serving as a general accelerator to speedup various algorithms by reducing search space in advance. We outline the framework of CHC-Index in **§3.1**, covering its construction (**§3.2**) and compressed storage (**§3.3**).

For CHC-Index's construction (**§3.2**), we leverage the monotonicity of structural cohesiveness, i.e., a k-core (k-truss) is contained in a $(k-1)$-core ($(k-1)$-truss), to maintain CHC-Index (denoted by Υ) as a hierarchical overlay with multiple layers (**§3.2.1**). For the k-th layer, we construct a sub-index Υ_k for all nodes belong to the k-core or k-truss, indexing them as a proximity graph (PG) where two nodes are connected in Υ_k if and only if they are close in terms of a specific attribute distance metric. In this way, the community model and attribute distance metric specific to different CS problems are considered parameters for CHC-Index construction, allowing flexible customization of indexes for various algorithms. We leverage the excellent retrieval capabilities of PG to return nodes with smaller attribute distances to q from the k-th layer sub-index Υ_k as candidates for searching the desired community. Despite the simplicity of the basic idea, implementing it involves two challenges related to the retrieval quality from Υ_k. (1) *How to enhance the navigability of the built index is crucial for retrieving high-quality candidate nodes. To achieve this,* we introduce an angular factor β to ensure that a node's neighbors in CHC-Index are spread across various directions in space by controlling the lower bound of the angles between the node and its neighbors (Theorem 1 in §3.2.2). This allows the search on CHC-Index to effectively converge from multiple directions. (2) *How to enhance the reliability of the built index is critical for retrieval quality. Note that, retrieving only attribute-similar nodes may miss nodes that are vital for maintaining community structure but have lower attribute similarity. To address this, we additionally* consider the structural locality and commonality of nodes in a community, integrate them with attribute distance to present a composite distance metric, and use it to optimize neighbor selection during CHC-Index construction (Theorem 2 in §3.2.2), thus refining the topology of the CHC-Index and enhancing the retrieval quality.

For CHC-Index's storage (**§3.3**), an intuitive method is to store each layer sub-index separately. While easy to implement, it increases the overall storage overhead, because nodes in higher layers exhibit significant redundancy in lower layers, due to the monotonicity of structural cohesiveness. *How to reduce the overhead of index storage is essential for improving usability.* To handle this, we design a compressed storage structure that preserves the topology of the hierarchical overlay of CHC-Index without losing information. We compress neighbors that appear repeatedly across different layers, ensuring redundant cross-layer neighbors are stored only once, thus reducing storage overhead. Furthermore, we prove that a node's neighbors can be correctly obtained by efficiently sequential reading from the compressed storage structure (Theorem 3, in §3.3).

The main contributions of this paper are summarized as follows.

- We propose a CHC-Index that incorporates the structural and attribute cohesiveness metrics essential for CS on attributed graphs, thereby adaptable to various CS algorithms (**§3.1**).
- We maintain CHC-Index as a hierarchical overlay with multiple layers, where each layer is organized as a proximity graph based on a specific attribute distance metric (**§3.2.1**). We refine CHC-Index's navigability and reliability by two optimizations (**§3.2.2**).
- We design a compressed structure that preserves the physical topology of CHC-Index without loss of information (**§3.3**) and outline the general search procedure using CHC-Index in **§4**.
- We conduct extensive experiments on five real-world datasets to evaluate from: efficiency and effectiveness (**§5.2-5.4**), ablation study (**§5.5**), index overhead (**§5.6**), scalability on dynamic graphs (**§5.7**), case study (**§5.8**), and parameter sensitivity (**§5.9**).

2 Preliminary and Problem

2.1 Preliminaries

Definition 1 (**Attributed Graph**). *An attributed graph is defined as $G = (V_G, E_G, A_G)$, where V_G (E_G) is the node (edge) set and A_G is the set of attributes associated with nodes. For each node $v \in V_G$, it has a set of attributes $A(v) = \{a_0, \ldots, a_m\}$, where $A(v) \subseteq A_G$ and $A(v)_i$ indicates the i-th attribute a_i of a node v.*

We next introduce the structural and attribute cohesiveness.

Structure cohesiveness. In the literature of community search (CS), k-core and k-truss are two most widely used community models, considering the trade-off between the computational complexity and their ability to measure structure cohesiveness [21].

Definition 2 (**k-core** [49]). *Given a graph G and an integer $k \geq 1$, a k-core of G is the largest subgraph $H_k \subseteq G$, such that $\forall v \in H_k$ has a degree at least k, i.e., $deg(v, H_k) \geq k$.*

The k-core model overlooks the strength of edges, which motivates k-truss model defined based on the concept of *support*.

Definition 3 (**Support** [57]). *Given an edge $e_{uv} \in E_G$, we define its support as the number of triangles containing e_{uv}, i.e., $sup(e_{uv}) = |\Delta_{uvw}|$, where w is a common neighbor of u and v. Given $N(\cdot)$ represents a node's neighbors, we have $sup(e_{uv}) = |N(u) \cap N(v)|$.*

The more triangles an edge involved, the more stable this edge becomes, indicating a strong connection between two nodes.

Definition 4 (**k-truss** [57]). *Given a graph G and an integer $k \geq 2$, a k-truss is defined as the largest subgraph $T_k \subseteq G$, such that $\forall e \in T_k$ has a support $sup(e, T_k) \geq k - 2$.*

The classic k-core and k-truss models do not require the subgraph to be connected, meaning that a k-core or k-truss may contain multiple connected components, which contradicts the inherent connectivity of communities. Therefore, some studies [28, 35, 42, 48, 51, 58] define the connected k-core (resp., k-truss) as a connected subgraph $\tilde{H}_k \subseteq H_k \subseteq G$ (resp., $\tilde{T}_k \subseteq T_k \subseteq G$), such that for each node $v \in \tilde{H}_k$, $deg(v, \tilde{H}_k) \geq k$ (resp., $\forall e \in \tilde{T}_k$, $sup(e, \tilde{T}_k) \geq k - 2$).

Example 1. *In Figure 1(b), the 3-truss T_3 is indicated by blue lines, which consists of two connected 3-truss, that are the left subgraph formed by nodes $v_1 \sim v_6$ and the right one formed by nodes $v_8 \sim v_{12}$.*

Attribute cohesiveness. Various methods are used to measure attribute distance of nodes. Without loss generality, we use $f(u, v)$ to represent the attribute distance between nodes u and v, and $F(f(u, v), C)$ to denote the attribute cohesiveness of a community C, measured based on a specific attribute distance $f(u, v)$ for $\forall u, v \in V_C$. Here, a community C is modeled as a k-core (resp., k-truss) H_k (resp., T_k) or its connected variants $\tilde{H}_k$ (resp., $\tilde{T}_k$). For instance, VAC [35] defines numerical attribute distance by Euclidean distance and assumes that a community's attribute cohesiveness is dominated by the worst pairwise attribute distance, i.e., $F(f(u, v), C) = \max\{f(u, v) | u, v \in V_C\}$, and aims to minimize it. SEA [51] measures numerical attribute distance by Manhattan distance and define $F(f(u, v), C) = \text{avg}\{f(u, q) | u, q \in V_C\}$ as the average attribute distance of all nodes to q, and aims to minimize it.

CS on attributed graphs (CSAG). We next provide the formalized CSAG problem, considering the aforementioned structural and attribute cohesiveness. Given a graph G, a community model C_{model} defined by (connected) k-core or k-truss, an attribute distance metric $f(u, v)$, a community attribute cohesiveness metric $F(f(u, v), C)$, a query node q, and a parameter $k \geq 1$ (resp., $k \geq 2$) for core-based (resp., truss-based) C_{model}. The CSAG problem is represented by a quadruple $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$, aiming to find a community $C \subseteq G$ satisfying the following properties:

- **Query participation.** The community C contains q, i.e., $q \in V_C$;
- **Structure cohesiveness.** The community C is modeled by a community model C_{model} with the input k, i.e., $C \in \{H_k, \tilde{H}_k, T_k, \tilde{T}_k\}$;
- **Attribute cohesiveness.** The community C has the optimal attribute cohesiveness $F(f(u, v), C)$ with opt $\in \{\max, \min\}$.

$$C = \underset{C \subseteq G}{\arg\text{opt}}\, F(f(u, v), C) \tag{1}$$

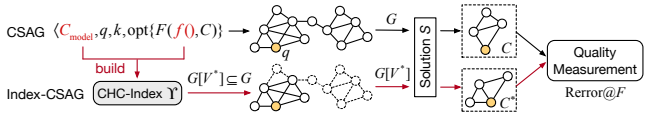


Figure 3: Pipeline of Index-CSAG

Example 2. *Recall VAC [35], which defines a community as C_{model} = "`connected k-truss`" and returns a community $C = \tilde{T}_k$ containing q, satisfying that the largest attribute distance $f(u, v)$ within C is minimized (opt = min), i.e., $C = \arg\min_{C \subseteq G}(\max\{f(u, v) | u, v \in V_C\})$.*

2.2 Problem Definition

Given the above definitions, we study the problem of designing a general index tailored for CSAG to enhance its performance.

Index-based CSAG problem (Index-CSAG). Given a CSAG problem $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$ over a graph G, we aim to take a community model C_{model} and an attribute distance metric $f()$ as input to build an index Υ, based on which the CSAG problem can be solved by returning a result community C^* from a smaller search space $G[V^*]$, where $G[V^*] \subseteq G$ is a subgraph formed by a set of nodes V^* retrieved from the index Υ. The community C^* satisfies the same properties as original CSAG problem:

- **Query participation.** The community C^* contains q;
- **Structure cohesiveness.** The community C^* is modeled by a community model C_{model} with the input k;
- **Attribute cohesiveness.** The community C^* has the optimal attribute cohesiveness $F(f(u, v), C^*)$.

$$C^* = \underset{C^* \subseteq G[V^*]}{\arg\text{opt}}\, F(f(u, v), C^*) \tag{2}$$

Figure 3 illustrates the coarse-grained pipeline of Index-CSAG, indicated by red lines. Given a CSAG problem with a solution S aimed at finding a connected k-truss ($k = 3$) community containing the specific query node q (shown in yellow), Index-CSAG leverages the built index Υ to narrow down the search space from the entire G to a smaller subgraph $G[V^*] \subseteq G$, with the dashed part pruned. Here, V^* involves nodes retrieved from Υ that are likely to belong the same community as q. We then apply the original solution S to $G[V^*]$ to yield a connected 3-truss community C^*. In contrast, CSAG uses the entire G as input for S and returns a connected 3-truss community C. To measure the quality of C^* compared to C, we introduce the *relative error of attribute cohesiveness* $F()$ below.

Definition 5 (**Relative error of** $F()$). *Given two communities C and C^* with attribute cohesiveness of $F(f(u, v), C)$ and $F(f(u, v), C^*)$, we define the relative error of $F()$, denoted by* `Rerror@F`*, as Eq. 3.*

$$\texttt{Rerror@}F = \frac{F(f(u, v), C^*) - F(f(u, v), C)}{F(f(u, v), C)} \tag{3}$$

We avoid using the absolute value when calculating `Rerror@F` for a specific reason. Referring to Table 1, we observe that the solution S to a CSAG problem can be either exact or appropriate. When S is an approximate solution, maintaining the positive or negative `Rerror@F` clearly indicates whether the result has improved or deteriorated. For approximate solutions with opt = min, e.g., SEA [51], we consider C^* to be better (worse) than C if $F(f(), C^*) < F(f(), C)$ ($F(f(), C^*) > F(f(), C)$). So, a smaller (larger) negative (positive) `Rerror@F` implies a better (worse) result. While for ATC [28] with opt = max, a larger positive `Rerror@F` indicates better results.

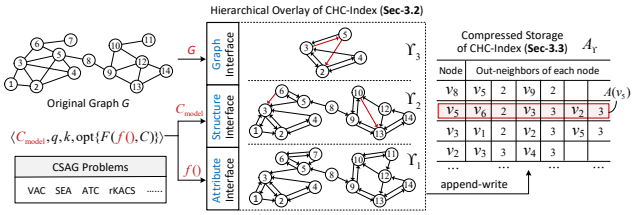


Figure 4: Framework of CHC-Index

3 The Proposed CHC-Index

3.1 Overview of CHC-Index

Figure 4 illustrates the framework of the proposed Cohesiveness-aware Hierarchical Compressed index (CHC-Index) Υ. Given a CSAG problem $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$ on a graph G, we use the community model C_{model} and attribute distance $f()$ to build and store Υ. C_{model} specifies the structural cohesiveness features for building Υ, ensuring nodes with the same structural cohesiveness as the query node q can be retrieved from Υ. While $f()$ offers insights into which attribute cohesiveness features to consider for building Υ, ensuring nodes with similar attributes as q are returned.

Logical view of Υ (§3.2). Since a k-core or k-truss (indicated by C_{model}) is contained in a $(k-1)$-core or $(k-1)$-truss, e.g., $H_k \subseteq H_{k-1}$, it makes sense to build Υ considering this structural cohesiveness monotonicity. In the logical view of CHC-Index Υ, nodes are organized in a hierarchical overlay with multiple layers. For the k-th layer, we construct a sub-index Υ_k for all nodes in the k-core or k-truss, indexing them as a proximity graph (PG) where two nodes are connected in Υ_k if and only if they are close in attribute distance $f()$. We expect to retrieve nodes with smaller attribute distances to q from the k-th layer sub-index Υ_k as candidates for searching a community of q, modeled as a k-core or k-truss with the given k. In Figure 4 (middle), we show the sub-indices for k-core model, where each Υ_k is used to index all nodes from a k-core of G.

Physical view of Υ (§3.3). Given the hierarchical overlay of Υ, an simple storage method is to store each layer sub-index separately. While easy to implement, this increases the overall storage overhead due to the redundancy, as nodes in higher layers also appear in the lower layers. To address this, we design a compressed storage structure that preserves the topology of the hierarchical overlay without losing information. Specifically, we compress each node's neighbors that appear repeatedly across different layers, storing redundant neighbors only once to reduce storage overhead.

Building process of Υ. Given a graph G, a community model C_{model}, and an attribute distance metric $f()$, Algorithm 1 outlines the process of building the CHC-Index Υ, with details in §3.2-3.3.

(1) Initialization. Given a graph G, we use core-decomposition [6] for core-based community model C_{model} or truss-decomposition method [44] for truss-based C_{model} to obtain the largest k value of k-core or k-truss of G, denoted by $k_{\max}$ (line 1). We then initialize $k_{\min}$ as 1 or 2 for core-based or truss-based C_{model} (line 2), respectively.

(2) Building sub-index Υ_k (§3.2). We build and store each layer sub-index Υ_k for $k \in [k_{\min}, k_{\max}]$ incrementally (lines 3-6). Specifically, the k-th layer sub-index Υ_k is constructed from the $(k-1)$-th layer sub-index Υ_{k-1} (line 4), given the initial sub-index $\Upsilon_{k_{\min}}$ is built as a proximity graph (PG) called Vamana [30]. In fact, Υ_k inherits

Algorithm 1: `buildIndex`$(G, C_{\text{model}}, f())$

Input: Graph G, community model C_{model}, attribute distance $f()$
Output: Physical storage structure of Υ, denoted by A_Υ

```
// Initialization
1 k_max ← decomposition(C_model, G) ; /* get the largest layer */
2 k_min ← 1 or 2 for core-based or truss-based C_model;
3 for ∀k ∈ [k_min, k_max] do
      // Build sub-index Υ_k
4     Υ_k ← buildSubIndex(k, f()) ;          /* Alg. 2 in §3.2 */
      // Record sub-index Υ_k
5     A_Υ ← physicalStore(A_Υ, Υ_k) ;        /* Alg. 8 in §3.3 */
6 end
7 return A_Υ;
```

a backbone structure from Υ_{k-1}, thereby preserving its indexing capabilities. It also incorporates local topology updates (e.g., red edges in Figure 4) to further enhance search performance over Υ_k.

(3) Physical storage of Υ_k (§3.3). For the k-th layer sub-index Υ_k, we employ an append-write strategy to store it in a *compressed adjacency list* A_Υ (line 5). Given a node $v \in V_{\Upsilon_k}$, we store it and its neighbors in an entry of A_Υ, denoted by $A(v) = \langle v, N_\Upsilon(v)\rangle$, where $N_\Upsilon(v)$ is a compressed structure maintaining all v's neighbors across different layers. This allows easy retrieval of v's neighbors at any Υ_k from $N_\Upsilon(v)$. Once a logical layer sub-index Υ_k is built, we update v's entry $A(v)$ by appending v's newly added neighbors in Υ_k to the end of $N_\Upsilon(v)$, while keeping existing neighbors unchanged.

Remarks. Before diving into the CHC-Index details in §3.2 and §3.3, it is worth mentioning the minimal requirements for the CSAG problem to which our CHC-Index is applicable. (1) *Hierarchical community model.* In the real world, communities inherently possess hierarchical properties [3, 13], which is beneficial for many applications, such as generative Q&A based on GraphRAG [17] and organized crime gang detection [46]. Our CHC-Index utilizes this hierarchy for efficient bottom-up index construction (§3.2), and it helps compress redundant nodes across layers to optimize storage space (§3.3). Besides k-core and k-truss, CHC-Index also supports k-clique-based community models, such as k-component [37] and kr-clique [33], provided they satisfy the hierarchical property. Since a k-clique belongs to a k-truss, trusses can serve as a fast filter for identifying cliques [8]. For example, we can build a CHC-Index with k ranging from 2 to $k_{\max}$ (the largest k of k-truss), and search within the k-th layer sub-index to find a k-clique-based community. (2) *Quantifiable attribute cohesiveness based on the attribute similarities of nodes.* As outlined in §2.1, the objective of a CSAG problem is to identify a community with optimized attribute cohesiveness. If the attribute cohesiveness can be quantified using attribute similarities of nodes, then we can leverage attribute similarity as a distance metric to construct each layer of CHC-Index as a proximity graph, enabling efficient retrieval of nodes highly similar to the query node. Since attribute cohesiveness is closely tied to similarity, these retrieved nodes are likely to contribute significantly to optimizing the community's attribute cohesiveness.

3.2 Hierarchical Overlay of CHC-Index

We take the core-based community model C_{model} as an example to demonstrate how to build the k-th layer sub-index Υ_k given an attribute distance $f()$. We first present a basic solution in §3.2.1 and enhance it with two optimizations in §3.2.2.

Algorithm 2: `buildSubIndex`(k, $f()$)

Input: The layer k and attribute distance metric $f()$ between nodes
Output: The k-th layer sub-index Υ_k

1 **if** $k == k_{\min}$ **then**
  `// Construction of` Υ_1 `from` G `for core-based` C_{model}
  `// Step 1. Initialization of` Υ_1 `from` G
2 $\Upsilon_k \leftarrow$ `randomGraph`(G, R) ; `/* directed` R`-regular graph */`
  `// Step 2. Refinement of` Υ_1 `(lines 3-14)`
3 **for** $\forall v \in V_{\Upsilon_0}$ **do**
4 [`Result`,`Visit`] $\leftarrow \Upsilon_k$.`greedySearch`$(s, v, 1, L, f())$;
    `/* find the top-1 closest (w.r.t.` $f()$`) node to` v `*/`
5 `Cand` $\leftarrow$ `Visit`$\cup N_{\text{out}}(v)$ and $N_{\text{out}}(v) \leftarrow \emptyset$;
6 Υ_k.`robustPrune`$(v$, `Cand`, $\alpha, R, f())$; `/* update out-neighbors` $N_{\text{out}}(v)$ `of each node` $v \in V_{\Upsilon_1}$ `*/`
7 **end**
8 **else**
  `// Construction of` Υ_k `from` Υ_{k-1}
  `// Step 1. Initialization of` Υ_k `from` Υ_{k-1}
9 $\Upsilon_k \leftarrow$ `inducedGraph`(Υ_{k-1}) ; `/* induced graph from` Υ_{k-1} `*/`
  `// Step 2. Refinement of` Υ_k
10 $\Upsilon_k \leftarrow$ `regularGraph`(Υ_k, R) ; `/*` R`-regular` Υ_k `*/`
11 **end**

3.2.1 *Basic Solution based on Vamana.*

The purpose of constructing the CHC-Index was to reduce the search space by retrieving nodes in the same k-core as the query node q and have similar attributes to q. This aligns with the goal of Approximate Nearest Neighbor Search (ANNS), which aims to find approximate top-n nodes close (in terms of a distance metric) to a query node from a well-structured Proximity Graph (PG). So, an intuitive approach is to introduce the concept of PG as the building block of CHC-Index. Since Vamana is a high-performance PG that plays an important role in the mainstream ANNS algorithm DiskANN [30], we propose a basic solution based on Vamana to build each layer sub-index Υ_k with $k \in [k_{\min}, k_{\max}]$ of CHC-Index.

Overview. Given a desired layer $k \in [k_{\min}, k_{\max}]$ and an attribute distance metric $f()$, Algorithm 2 outlines the process of building the sub-index Υ_k. We initialize the first layer $\Upsilon_{k_{\min}}$ as a Vamana graph (lines 1-7). For $k > k_{\min}$, we build Υ_k from Υ_{k-1} (lines 8-11).

Construction of $\Upsilon_{k_{\min}}$. For core-based community model C_{model}, we build the first layer Υ_1 ($k_{\min} = 1$) as a Vamana graph in two steps. (1) We initialize Υ_1 as a random directed R-regular graph (line 2), where each node $v \in V_{\Upsilon_1}$ has R random out-neighbors $|N_{\text{out}}(v)| = R$. (2) We refine the topology of Υ_1 through `greedySearch` (Algorithm 3) and `robustPrune` (Algorithm 7) to enhance its quality (lines 3-7).

(1) Initialization. We initialize Υ_1 via `randomGraph` as follows.

• `randomGraph`(G, R). For each node $v \in V_G$, we add directed edges from v to R randomly selected nodes in $V_G \setminus \{v\}$, until all nodes have R out-neighbors. Figure 5 (left) shows a 2-regular graph for the graph G illustrated in Figure 4. The degree bound R is empirically set on a logarithmic scale based on the size of G [38], i.e., $O(\ln |V_G|)$. Since the initial R-regular graph is formed randomly, its quality cannot be guaranteed, and lower quality can increase the time required for topology refinement. This suggests that constructing a good initial R-regular graph could improve the overall construction efficiency, which we will explore in future research.

(2) Refinement. Given a randomly formed R-regular graph, it is evident that a node's R out-neighbors may not be its optimal proximity

Algorithm 3: `greedySearch`(s, q, m, L, $f()$)

Input: Start node s of a specific sub-index, query node q, result size m, search list size $L \geq m$, and attribute distance metric $f()$
Output: top-m nodes in `Result` and all visited nodes in `Visit`

1 `Result` $\leftarrow \{s\}$ and `Visit` $\leftarrow \emptyset$;
2 **while** `Result` $\setminus$ `Visit` $\neq \emptyset$ **do**
3 $v \leftarrow \arg\min_{v \in \texttt{Result} \setminus \texttt{Visit}} f(v, q)$;
  `// Expand the search from the closest node to` q
4 $N_{\text{out}}(v) \leftarrow$ get out-neighbors of v from Υ_k;
5 `Result` $\leftarrow$ `Result` $\cup N_{\text{out}}(v)$ and `Visit` $\leftarrow$ `Visit` $\cup \{v\}$;
6 **if** |`Result`| $> L$ **then**
7 update `Result` to retain closest L nodes to q;
8 **end**
9 **end**
10 **return** [closest m nodes from `Result`, `Visit`]

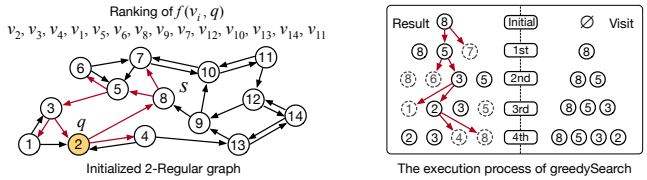


Figure 5: An example of `greedySearch` on a 2-Regular graph

neighbors, which necessitates refining each node's out-neighbors. Intuitively, when searching for a node v from a start node s in Υ_1, the nodes along the converging path, the nodes along the converging path from s to v are likely to be closer to v than those not on this path. So, it makes sense to use all visited nodes along this path to update v's out-neighbors. We take each node $v \in V_{\Upsilon_1}$ as a query q and apply `greedySearch`(s,v,m,L,$f()$) on Υ_1 to find the top-m ($m = 1$) approximate nearest neighbors of v from s. Typically, s is the medoid of V_{Υ_1} in terms of the attribute distance $f()$. All nodes along the converging path from s to v are stored in a set `Visit` and the top-m nodes closest to v are stored in `Result` (line 4). We then use `roubstPrune` to update v's out-neighbors with nodes from `Visit` (lines 5-6). Essentially, `greedySearch` identifies candidate nodes in `Visit` for updating out-neighbors, and `robustPrune` refines out-neighbors based on the candidates.

• `greedySearch`(s,q,m,L,$f()$). Intuitively, searching along the nodes closest to the query node q helps to gradually approach the neighborhood of q. Algorithm 3 outlines the entire procedure of `greedySearch` on a sub-index Υ_k from a start node s. We initialize `Result` with $\{s\}$ and `Visit` with $\emptyset$. At each step of search expansion, we select an unvisited node v closest to q (i.e., with the smallest attribute distance $f(v, q)$) from `Result`\`Visit` as the next-hop (line 3), and update `Result` and `Visit` with $N_{\text{out}}(v)$ and v, respectively (lines 4-5). Here, we introduce a parameter $L \geq m$ to control the search width (lines 6-8). A larger L helps find more accurate top-m results but increases search time. Empirically, L is set within a range of (1,2] times m to balance accuracy and efficiency.

Example 3. *Figure 5 (right) illustrates the procedure for running* `greedySearch` *on the left 2-Regular graph, given* $s = v_8$, $q = v_2$, $m = 1$, *and* $L = 2$. *We provide the rankings of all nodes in ascending order of their attribute distances* $f(v, q)$ *to* q *(top left). Given the initial* `Result` $= \{v_8\}$ *and* `Visit` $= \emptyset$, *we expand the search from* v_8 *and update* `Result` $= \{v_8, v_5, v_7\}$ *with its two out-neighbors and* `Visit` $= \{v_8\}$. *Since* |`Result`| $> L$, *we retain the closest* L *nodes to* v_2 *as* `Result` $= \{v_8, v_5\}$ *(dashed circles indicate nodes removed from* `Result`*). We*

Algorithm 4: robustPrune(v, Cand, α, R, $f()$)

Input: Node v, candidate set Cand, distance factor α, degree bound R, and attribute distance metric $f()$
Output: v is updated with at most R new out-neighbors $N_{\text{out}}(v)$

1 **while** Cand $\neq \emptyset$ **do**
  // Step 1. Update $N_{\text{out}}(v)$ with the closest node to v
2 $u \leftarrow \arg\min_{w \in \text{Cand}} f(v, w)$;
3 $N_{\text{out}}(v) \leftarrow N_{\text{out}}(v) \cup \{u\}$;
4 **if** $|N_{\text{out}}(v)| == R$ **then**
5 **break**;
6 **end**
  // Step 2. Prune candidates that are dominated by u
7 **for** $w \in$ Cand **do**
8 **if** $\alpha f(u, w) \leq f(v, w)$ **then**
9 Cand$\leftarrow$ Cand$\setminus\{w\}$;
10 **end**
11 **end**
12 **end**

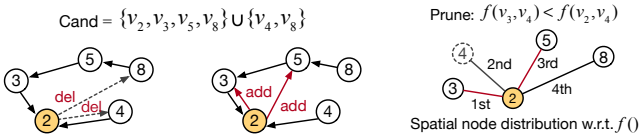


Figure 6: An example of robustPrune

next iteratively expand the search from the closest nodes v_5, v_3, and v_2 and update Result *and* Visit *accordingly, until all nodes in* Result *have been visited. As a result, we have* Result $= \{v_2, v_3\}$ *and* Visit$= \{v_8, v_5, v_3, v_2\}$, *and we finally return the closest node v_2 as the top-m closet node for $m = 1$.*

• roubstPrune(v,Cand,α,R,$f()$). Given all visited nodes Visit on the converging path from s to a node $v \in V_{\Upsilon_1}$, we prepare the candidates for out-neighbors refinement as Cand$\leftarrow$Visit$\cup N_{\text{out}}(v)$, and clear v's current out-neighbors (line 5 in Algorithm 2). We then call robustPrune(v,Cand,α,R,$f()$) to iteratively reassign R out-neighbors to v from Cand. In each iteration of Algorithm 4, we first select the closest node $u \in$ Cand to v as a new out-neighbor of v (lines 2-3), and then prune a candidate node $w \in$ Cand, if the pruning condition holds (lines 7-11). A node $w \in$ Cand is pruned if $\alpha f(u, w) \leq f(v, w)$, where the distance factor $\alpha \geq 1$. The intuition is that if w is sufficiently close to u, then adding a directed edge from v to w is unnecessary, as a search from v can still converge to w by approaching v's out-neighbor u. As a result, a node w satisfying $f(v, w) < \alpha f(u, w)$ remains in Cand for next iteration. We repeat the above until v has R new out-neighbors (lines 4-6).

Example 4. *Figure 6 provides the* robustPrune *process, refining the out-neighbors of node v_2 using* Visit $= \{v_8, v_5, v_3, v_2\}$ *from* greedySearch *in Example 3. For clarity, we show only the subgraph involved in this refinement. We initialize the candidates* Cand $=$ Visit $\cup N_{\text{out}}(v_2)$ *and delete the original out-neighbors* $N_{\text{out}}(v_2) = \{v_4, v_8\}$. *Given the spatial distribution of nodes in* Cand *shown on the right (provided with the rankings of attribute distance to v_2), we assign new out-neighbors for v_2 as follows. First, we select the closest node $v_3 \in$* Cand *as the first new out-neighbor and prune v_4 from* Cand *because $\alpha f(v_3, v_5) \leq f(v_2, v_5)$ (given $\alpha = 1$), which implies that a search from v_2 can converge to v_5's neighborhood by approaching v_3. Next, we select the closet $v_5 \in$* Cand *as the second out-neighbor.*

Algorithm 5: inducedGraph(Υ_{k-1})

Input: Sub-index Υ_{k-1} of the $(k-1)$-th layer
Output: Initialized sub-index Υ_k of the k-th layer

1 $D \leftarrow \{v \in V_{\Upsilon_{k-1}} | deg(v, G) < k\}$;
2 **while** $D \neq \emptyset$ **do**
3 $v \leftarrow$ select a node from D;
4 $D \leftarrow D \setminus \{v\}$;
5 $\Upsilon_{k-1} \leftarrow$ remove v and all edges involving v from Υ_{k-1};
6 **for** $\forall u \in N(v)$ **do**
7 $deg(u, G) = deg(u, G) - 1$;
8 $N(u) \leftarrow N(u) \setminus \{v\}$;
9 **if** $deg(u, G) < k$ **then**
10 $D \leftarrow D \cup \{u\}$;
11 **end**
12 **end**
13 **end**
14 **return** $\Upsilon_k \leftarrow \Upsilon_{k-1}$;

Algorithm 6: regularGraph(Υ_k, R)

Input: Initialized sub-index Υ_k and degree bound R
Output: Refined sub-index Υ_k

1 $M \leftarrow \{v \in V_{\Upsilon_k} | |N_{\text{out}}(v)| < R\}$;
2 **while** $M \neq \emptyset$ **do**
3 $v \leftarrow$ select a node from M;
4 $M \leftarrow M \setminus \{v\}$;
5 [Result,Visit] $\leftarrow \Upsilon_k$.greedySearch(s, v, 1, L, $f()$);
6 Cand $\leftarrow$ Visit$\setminus N_{\text{out}}(v)$;
7 Υ_k.robustPrune(v, Cand, α, R, $f()$);
8 **end**
9 **return** Υ_k;

Construction of Υ_k. We construct Υ_k from Υ_{k-1} with two steps in Algorithm 2. (1) Considering the structural cohesiveness monotonicity of k-core, we initialize Υ_k as a subgraph of Υ_{k-1} (line 9). (2) As the structural cohesiveness increases (i.e., $k-1 \rightarrow k$), some nodes in Υ_{k-1} would be excluded from Υ_k. This disrupts the R-regular property of Υ_k. As shown in Figure 4 (middle), nodes v_7, v_{11} are excluded from Υ_2 as they do not belong to a 2-core. The removal of v_7, v_{11} reduces the out-neighbors of nodes v_6, v_{10} in Υ_k to be $< R$, affecting the quality of Υ_k. Thus, we refine Υ_k by complementing out-neighbors for those nodes v with $N_{\text{out}}(v) < R$ (line 10).

(1) Initialization. We initialize Υ_k via inducedGraph as follows.

• inducedGraph(Υ_{k-1}). Given a sub-index Υ_{k-1}, we peel all nodes that do not belong to the k-core and remain the others in Υ_k. Specifically, we initialize a set D with all nodes having a degree $< k$ (in the original graph G), denoted by $D = \{v \in V_{\Upsilon_{k-1}} | deg(v, G) < k\}$ (line 1). Since they do not participate in the k-core, we iteratively delete each $v \in D$ from Υ_{k-1} (lines 3-5), update the degree of v's neighbor $u \in N(v)$ as $deg(u, G) - 1$ and remove v from u's neighbors, and add u to D if $deg(u, G) < k$ (lines 6-12). We repeat this until D is empty. As a result, all nodes in the k-core are remained in Υ_k.

(2) Refinement. We next refine Υ_k using regularGraph(Υ_k, R) to ensure its R-regularity. This R-regularity not only enhances the search performance over Υ_k, but is also critical to the compressed storage of CHC-Index Υ, which we will discuss in §3.3.

• regularGraph(Υ_k,R). We first maintain all nodes with # out-neighbors $< R$ in a set M for R-regularization (line 1). For each node $v \in M$, we adopt a similar approach to the Refinement step in Algorithm 2 to complete v's out-neighbors. The difference is

that Refinement step in Algorithm 2 reassigns R new out-neighbors for each node, whereas Algorithm 6 retains existing out-neighbors and only fills in the missing ones. For each node $v \in M$, we fist apply `greedySearch`(s,v,1,L,$f()$) to find the top-1 node close to v and return all visited nodes `Visit` during the search process (line 5). We next establish a set `Cand` by removing existing out-neighbors of v from `Visit`, denoted by `Cand`$\leftarrow$`Visit`$\setminus N_{\text{out}}(v)$ (line 6). Finally, we invoke `robustPrune`(v,`Cand`,α,R,$f()$) to select nodes from `Cand` as the completed out-neighbors until $N_{\text{out}}(v) = R$ (line 7).

Example 5. *Recalling the sub-index Υ_1 in Figure 4, we construct Υ_2 from Υ_1 as follows. Since v_7 and v_{11} do not belong to 2-core, they are excluded from the induced graph of Υ_1. Consequently, v_6 and v_{10} each have only one out-neighbor and need another out-neighbors to reach $R = 2$. Given $M = \{v_6, v_{10}\}$,* `regularGraph` *performs as follows. (1) It calls* `greedySearch` *with $q = v_6$, $s = v_8$, $m = 1$, and $L = 2$, which returns* `Visit` $= \{v_3, v_6, v_5, v_8\}$. *(2) It prepares* `Cand` $=$ `Visit` $\setminus \{v_5\} = \{v_3, v_6, v_8\}$. *(3) It invokes* `robustPrune` *to add the closest node v_3 as an out-neighbor of v_6. (4) This process repeats until $M = \emptyset$.*

3.2.2 Optimized solution.

In this section, we optimize the basic solution from two aspects: enhancing the search efficiency over the CHC-Index Υ, and improving the quality of nodes retrieved from Υ. Both aspects are critical for improving the performance of community search using Υ.

Optimization on `robustPrune`. In the original `robustPrune`, a distance factor $\alpha \geq 1$ is used to control the navigability of the Vamana graph, which is crucial for the search efficiency over Υ. We claim that relying solely on α is insufficient to achieve a stable search efficiency, motivating our first optimization on `robustPrune`.

Motivation. We explore the motivation by two cases: $\alpha = 1$ and $\alpha > 1$. Let us begin with a good property for $\alpha = 1$ (Lemma 2).

Lemma 1. *Given a node v, an out-neighbor $u \in N_{\text{out}}(v)$, and a candidate node $w \in$ Cand. If $f(v, w) < \alpha f(u, w)$, then w would remain in* Cand, *resulting an angle $\angle uvw > \arccos(\frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)})$.*

Proof. Consider a triangle composed of node u, v, w, we have the following derivation given $f(v, w) < \alpha f(u, w)$ [24].

$$\begin{aligned}\cos \angle uvw &= \frac{f(v,u)^2 + f(v,w)^2 - f(u,w)^2}{2f(v,u)f(v,w)} \\ &= \frac{f(v,u)}{2f(v,w)} + \frac{(1 - \frac{f(u,w)^2}{f(v,w)^2})f(v,w)}{2f(v,u)} \\ &< \frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)}\end{aligned}$$

Thus, we have $\angle uvw > \arccos(\frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)})$. □

Lemma 2. *Given three nodes v, $u \in N_{\text{out}}(v)$, and $w \in$ Cand. If $f(v, w) < \alpha f(u, w)$ and $\alpha = 1$, then we have $\angle uvw > 60°$.*

Proof. According to Lemma 1, we have $\angle uvw > \arccos \frac{f(v,u)}{2f(v,w)}$ for $\alpha = 1$. Since we select the closest node u from Cand as one of v's out-neighbor in each iteration of `robustPrune`, we have $f(v, w) \geq f(v, u)$ for each remained node $w \in$ Cand. Thus, $\frac{f(v,u)}{2f(v,w)} \leq \frac{1}{2}$ holds, indicating that $\angle uvw > \arccos \frac{f(v,u)}{2f(v,w)} \geq \arccos \frac{1}{2} = 60°$. □

Algorithm 7: `robustPrune++`(v, Cand, α, β, R, $f()$)

Input: Node v, candidate set Cand, distance factor α, angular factor $\beta \in [\sqrt{1-\alpha^{-2}}, 1]$, degree bound R, and distance metric $f()$
Output: v is updated with at most R new out-neighbors $N_{\text{out}}(v)$

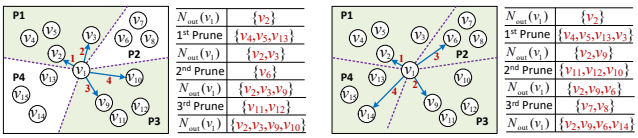
```
1  while Cand ≠ ∅ do
       // Step 1. Update N_out(v) with the closest node to v
2      u ← arg min_{w∈ Cand} f(v, w);
3      N_out(v) ← N_out(v) ∪ {u};
4      if |N_out(v)| == R then
5          break;
6      end
       // Step 2. Prune candidates that are dominated by u
7      for w ∈ Cand do
8          if αf(u, w) ≤ f(v, w) then
9              Cand← Cand\{w};
10         else if α > 1 && f(v,u)/f(v,w) ∉ [β ± √(β² − (1 − α⁻²))] then
11             Cand← Cand\{w};
12     end
13 end
```

Setting $\alpha = 1$ yields a favorable spatial distribution of v's out-neighbors [24], ensuring that the angle formed by v and its two consecutively added out-neighbors $u, w \in N_{\text{out}}(v)$ is $> 60°$, which improves the navigability of Υ_k in all directions. However, setting $\alpha = 1$ can lead to a proximity graph with a larger diameter, increasing the search convergence time on Υ_k [30] and partially negating the efficiency gains from improved navigability. So, a tunable $\alpha > 1$ is recommended in practice to reduce the diameter by decreasing the lower bound of $\angle uvw$. According to Lemma 1, the lower bound of $\angle uvw$ is affected by both α and $\frac{f(v,u)}{f(v,w)}$, allowing for any angle $> 0°$ when $\alpha > 1$. For example, given $\alpha = 3$ and $f(v, u) = f(v, w)$, we have $\angle uvw > \arccos(\frac{1}{2} + \frac{3}{8}) = 19.2°$. This implies that if nodes are pruned solely using the condition $\alpha f(u, w) \leq f(v, w)$ when $\alpha > 1$, the candidate set Cand may include nodes w with a small angle $\angle uvw$, negating the efficiency gains from the reduced diameter.

Our solution. To further enhance the navigability of a sub-index, we introduce an angular factor $\beta \in [\sqrt{1-\alpha^{-2}}, 1]$ in a revised `robustPrune++` to ensure $\angle uvw > \arccos \beta$ for any $w \in$ Cand when $\alpha > 1$. In Algorithm 7, we first prune nodes following the same procedure as `robustPrune` (lines 8-9) for any $\alpha \geq 1$. Next, for any $\alpha > 1$, we keep refining the remained Cand by pruning those w that would cause a quite small angle $\angle uvw$ if another pruning condition holds (lines 10-11). The sufficient condition of $\angle uvw > \arccos \beta$ is $\frac{f(v,u)}{f(v,w)} \notin [\beta \pm \sqrt{\beta^2 - (1-\alpha^{-2})}]$ (Theorem 1).

Theorem 1. *Given a node v, an out-neighbor $u \in N_{\text{out}}(v)$, and a node $w \in$ Cand. If $\frac{f(v,u)}{f(v,w)} \in [\beta \pm \sqrt{\beta^2 - (1-\alpha^{-2})}]$ for $f(v, w) < \alpha f(u, w)$, and $\beta \in [\sqrt{1-\alpha^{-2}}, 1]$, then we have $\angle uvw > \arccos \beta$.*

Proof. According to Lemma 1, we have $\angle uvw > \arccos(\frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)})$ as $(v, w) < \alpha f(u, w)$. If $\frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)} < \beta$, then we have $\arccos(\frac{f(v,u)}{2f(v,w)} + \frac{(1-\alpha^{-2})f(v,w)}{2f(v,u)}) > \arccos \beta$, and $\angle uvw > \arccos \beta$ holds. Let us consider the term $\frac{f(v,u)}{f(v,w)}$ as a variable x, then the solution to the inequality $\frac{1}{2}(x + (1-\alpha^{-2})x^{-1}) < \beta$ is $x \in [\beta \pm \sqrt{\beta^2 - (1-\alpha^{-2})}]$. Besides, to ensure that the solution x is a real number, the range of β needs to be within $[\sqrt{1-\alpha^{-2}}, 1]$. □



(a) robustPrune (b) robustPrune++

Figure 7: Example of out-neighbors selection. We present the selected out-neighbors and pruned nodes in the table

Example 6. *Figure 7 illustrates the out-neighbor selection for v_1 using* `robustPrune` *and* `robustPrune++`. `robustPrune` *first selects the closest node v_2 as an out-neighbor and prunes v_4, v_5, v_{13}, as they are close enough to v_2 and can be reached through v_2. In contrast,* `robustPrune++` *also prunes v_3, as it prioritizes selecting a node that forms a larger angle* $> \arccos \beta$ *with v_1, v_2. So, in the second iteration,* `robustPrune++` *selects v_9. Finally,* `robustPrune` *selects four out-neighbors across only two regions P1 and P3, while* `robustPrune++` *covers four regions, achieving better navigability in all directions.*

Optimization on the distance metric. We assign out-neighbors based on attribute distance $f()$. However, relying solely on $f()$ is problematic sometimes, motivating our second optimization.

Motivation. In a k-core, some nodes are crucial for maintaining the community structure, and their absence would cause the k-core to collapse [60]. If these nodes have a large attribute distance from the query node q, they may be distant from other community members in Υ_k. Thus, a large m is required to ensure these crucial nodes are retrieved in the top-m closest nodes to q, which increases search time. This motivates us to consider *structural closeness* alongside attribute distance when building a sub-index Υ_k, to avoid overlooking nodes that are structurally important to the community.

Our solution. We first define the structural closeness $g(u, v)$ between two nodes u and v as complementary to attribute distance $f(u, v)$, then we combine them together as a composite metric $\eta(u, v)$ to measure the distance of two nodes. Finally, we replace the distance metric $f()$ with the composite $\eta()$ in `greedySearch` (Algorithm 3) and `robustPrune++` (Algorithm 7) to construct Υ_k. We next present structural closeness based on two observations.

• *Structural locality of a community.* According to the small world theory [32], two nodes in the same community exhibit strong access locality [10, 25, 28, 48, 51], meaning they are more likely to belong to the same community if they are within each other's localized space. The hops between two nodes is usually used to reflect their structural locality. Given three nodes u, v, w, if the hops from u to v is less than that from u to w, i.e., $hop(u, v) < hop(u, w)$, then v is more likely to belong to the same community as u compared to w.

• *Structural commonality of a community.* As discussed in [29, 44, 57], nodes that share more common neighbors exhibit greater structural commonality. Given two nodes u, v from G and one of their common neighbor $w \in N(u) \cap N(v)$, they form a triangle, showing a stable relationship among them. The more common neighbors $|N(u) \cap N(v)|$ they have, the stronger their relationship is, and the more likely they belong to the same community [39, 56].

We define the structural closeness $g(u, v)$, considering both structural locality and commonality. In Eq. 4, $g()$ consists of two parts: (1) The first part reflects the effect of structural locality of u, v, where a shorter distance between v and u results in a smaller $g()$. (2) The second part reflects the effect of structural commonality of v and u, where the more common neighbors they have leads to a smaller $g()$. Additionally, we introduce a scale factor $\gamma > 0$ to balance the two parts. Generally, when $\gamma < 1$, smaller values of γ place more importance on structural commonality. Conversely, when $\gamma > 1$, larger values of γ place more importance on structural locality. When $\gamma = 1$, both aspects exhibit the same importance.

$$g(u, v) = \gamma \cdot (hop(u, v) - 1) + (1 - \frac{|N(u) \cap N(v)|}{|N(u)|}) \tag{4}$$

Proposition 1. *Given two nodes u and v with $hop(u, v) \geq 3$, their structural closeness $g(u, v)$ is determined solely by structural locality.*

It is evident to see that u and v have no common neighbors as $hop(u, v) \geq 3$, and we have $g(u, v) = \gamma \cdot (hop(u, v) - 1) + 1$.

Corollary 1. *Given two nodes v, w that are three hops away from u, if $hop(u, v) < hop(u, w)$, then v is more structural close to u.*

Proof. Since $hop(u, v) \geq 3$ and $hop(u, w) \geq 3$, we have $g(u, v) - g(u, w) = \gamma \cdot (hop(u, v) - hop(u, w))$ based on Proposition 1. Then, $g(u, v) < g(u, w)$ holds for $hop(u, v) < hop(u, w)$. □

This corollary somewhat mirrors real-world situations. In social networks where connections are based on friendships, if two users are more than three hops apart, it suggests they are neither direct friends nor have common friends. As hops increase, the relationship weakens as the involvement of more intermediary users.

Proposition 2. *Given two nodes u and v with $hop(u, v) \leq 2$, their $g(u, v)$ is determined by both the structural locality and commonality.*

Recalling social networks, evaluating whether users within two hops of each other are friends and share more common friends is crucial for determining if they belong to the same social circle.

Corollary 2. *Given two nodes v and w within two hops of u and satisfies $hop(u, v) \neq hop(u, w)$. We say v is more structural close to u, if $hop(u, v) < hop(u, w)$ and $|N(u) \cap N(w)| < |N(u) \cap N(v)| + \gamma \cdot |N(u)|$, or $hop(u, v) > hop(u, w)$ and $|N(u) \cap N(v)| > |N(u) \cap N(w)| + \gamma \cdot |N(u)|$.*

Proof. Since v and w are within two hops of u and $hop(u, v) \neq hop(u, w)$, we have two cases of $hop(u, v) < hop(u, w)$ ($hop(u, v) = 1$ and $hop(u, w) = 2$) and $hop(u, v) > hop(u, w)$ ($hop(u, v) = 2$ and $hop(u, w) = 1$). For the first case, we have $g(u, v) - g(u, w) = \frac{|N(u) \cap N(w)|}{|N(u)|} - \frac{|N(u) \cap N(v)|}{|N(u)|} - \gamma$. Then, $g(u, v) < g(u, w)$ holds if $|N(u) \cap N(w)| < |N(u) \cap N(v)| + \gamma \cdot |N(u)|$. Similarly, for the second case, we have $g(u, v) - g(u, w) = \frac{|N(u) \cap N(w)|}{|N(u)|} - \frac{|N(u) \cap N(v)|}{|N(u)|} + \gamma$. Then, $g(u, v) < g(u, w)$ holds if $|N(u) \cap N(v)| > |N(u) \cap N(w)| + \gamma \cdot |N(u)|$. □

This corollary reveals that within a two-hop range, it is not easy to determine whether two nodes have a better structural closeness based solely on single aspect. For instance, in a social network, considering a user v is a friend of u but they have only a few common friends, while for another user w is not a direct friend of u but shares more common friends with u. Both them demonstrate a certain structural closeness with u. To assert that v is more structurally close to u, it must be ensured that the structural commonality between w and u does not exceed a certain threshold (Corollary 2).

Given the structural closeness $g()$ and an attribute distance $f()$, we next define a composite metric $\eta() = g() + f()$ (Eq. 5).

$$\eta(u,v) = \gamma \cdot (hop(u,v) - 1) + (1 - \frac{|N(u) \cap N(v)|}{|N(u)|}) + f(u,v) \quad (5)$$

The parameter γ is used to control the effect of structural locality on the composite distance. We adjust γ based on Theorem 2.

Theorem 2. *Given three nodes u, v, and w, where $hop(u,v) = m$ and $hop(u,w) = n$. If $n \geq m + \frac{2}{\gamma}$, then we have $\eta(u,v) \leq \eta(u,w)$.*

Proof. Considering the composite distance $\eta(u,v)$, it has an upper bound as $\eta(u,v) \leq \gamma \cdot (m-1) + 2$ when u and v have none structural commonality ($|N(u) \cap N(v)| = 0$) and totally different attributes ($f(u,v) = 1$). While for $\eta(u,w)$, it has a lower bound as $\eta(u,w) \geq \gamma \cdot (n-1)$ when they share the most common neighbors and have the same attributes. Then, we have $\eta(u,v) - \eta(u,w) \leq \gamma \cdot (m-1) + 2 - \gamma \cdot (n-1) = \gamma \cdot (m-n) + 2$. If we have $n \geq m + \frac{2}{\gamma}$, then $\gamma \cdot (m-n) + 2 < 0$ holds, thereby $\eta(u,v) \leq \eta(u,w)$ holds. □

For example, when updating out-neighbors for a node u using `robustPrune++` with $\gamma = 1$, nodes that are 2-hops ($m = 2$) away from u have a smaller $\eta()$ than that more than 4 hops away from u ($n \geq m + \frac{2}{\gamma} = 4$), making them more likely to be selected as u's out-neighbors. As a result, nodes within a 2-hop range of a given node are more likely to be recalled than those outside 4-hop range. While for nodes within the 4-hop range, their recall depends on their attribute distance, structural locality, and structural commonality.

3.2.3 Complexity Analysis.

Complexity of basic solution (§3.2.1). The time complexity of basic solution comprises two parts: constructing the first layer sub-index $\Upsilon_{k_{\min}}$ and constructing the k-th layer sub-index for $k \in (k_{\min}, k_{\max}]$. Since $\Upsilon_{k_{\min}}$ is based on Vamana [30], and the complexity of Vamana is derived following the analysis in NSG [24], we conduct our complexity analysis based on [24, 30] as follows.

Complexity of constructing $\Upsilon_{k_{\min}}$. We initialize $\Upsilon_{k_{\text{mix}}}$ as a random R-regular graph via `randomGraph`, which costs a time of $O(nR)$ as it requires to process all nodes $n = |V_G|$ in a graph G. We next iteratively invoke `greedySearch` and `robustPrune` to refine each node's R out-neighbors. From the analysis of [24], the complexity of `greedySearch` is $O(\frac{Rn^{1/d}\log n^{1/d}}{\Delta r})$, where d is the # dimensions of attribute, Δr is a constant defined as $\Delta r = \min\{|f(u,v) - f(u,w)|, |f(u,v) - f(v,w)|, |f(u,w) - f(v,w)|\}$ for all possible non-isosceles triangles composed by any three nodes u, v, and w in G. According to [30], the complexity of `robustPrune` is $O(lR)$, where $l = |\texttt{Cand}|$ is the number of candidate nodes. Since we need to refine out-neighbors for all the n nodes in the worst case, the total time of out-neighbor refinement is $O(n \cdot (\frac{Rn^{1/d}\log n^{1/d}}{\Delta r} + lR))$. Thus, the total time for constructing $\Upsilon_{k_{\min}}$ is $O(nR + \frac{Rn^{1+1/d}\log n^{1/d}}{\Delta r} + nlR)$.

Complexity of constructing Υ_k for $k \in (k_{\min}, k_{\max}]$. We initialize Υ_k as an induced graph of Υ_{k-1} via `inducedGraph`. We denote the number of nodes removed in Υ_k as m_k. So, the complexity of `inducedGraph` is $O(m_k R)$. We next ensure the R-regularization of Υ_k by calling `regularGraph`, which employs `greedySearch` and `robustPrune` to refine a node's out-neighbors at each step. Suppose that Υ_k has n_k nodes, then refining out-neighbors for only one node of Υ_k requires time of $O(\frac{Rn_k^{1/d}\log n_k^{1/d}}{\Delta r} + lR)$. Since a node's removal would result in an average R nodes losing one out-neighbor, `regularGraph` requires to refine out-neighbors for $m_k R$ nodes on average, yielding the total time of $O(\frac{m_k R^2 n_k^{1/d}\log n_k^{1/d}}{\Delta r} + m_k lR^2)$. So, the time for constructing Υ_k is $O(m_k R + \frac{m_k R^2 n_k^{1/d}\log n_k^{1/d}}{\Delta r} + m_k lR^2))$. Because we need to construct Υ_k for all $k \in (k_{\min}, k_{\max}]$, the total time is $O(\sum(m_k R + \frac{m_k R^2 n_k^{1/d}\log n_k^{1/d}}{\Delta r} + m_k lR^2))$. Sine $\sum \frac{m_k R^2 n_k^{1/d}\log n_k^{1/d}}{\Delta r} < \int \frac{R^2 x^{1/d}\log x^{1/d}}{\Delta r} dx = \frac{R^2 n^{1+1/d}\log n^{1/d}}{\Delta r(1+1/d)}$, we have $\sum(m_k R + m_k lR^2) < nR + nlR^2$ as $\sum m_i < n$, which is dominated by $\frac{R^2 n^{1+1/d}\log n^{1/d}}{\Delta r(1+1/d)}$. So, the time of constructing all Υ_k is $O(\frac{R^2 n^{1+1/d}\log n^{1/d}}{\Delta r(1+1/d)})$.

Overall complexity. Since the time of constructing all Υ_k dominates that of $\Upsilon_{k_{\min}}$, the overall complexity is $O(\frac{R^2 n^{1+1/d}\log n^{1/d}}{\Delta r(1+1/d)})$.

Complexity of optimized solutions (§3.2.2). We provide the time complexity for each optimized solution respectively.

Complexity of solution with `robustPrune++`. We introduce an additional pruning condition to `robustPrune` without changing its input size, maintaining the same complexity as the basic solution.

Complexity of solution with composite distance. The composite distance mainly affects the time for pairwise distance calculations. In the basic solution, computing attribute distance $f()$ has $O(1)$ complexity. However, using composite distance involves extra computations: calculating the hops between two nodes by BFS ($O(n)$) and determining the common neighbors by set intersection ($O(\bar{d}_G)$, where $\bar{d}_G$ is the average degree of G). So, the complexity of the second optimized solution is $O((n + \bar{d}_G)\frac{R^2 n^{1+1/d} logn^{1/d}}{\Delta r(1+1/d)})$.

3.3 Compressed Storage of CHC-Index

We introduce a space-efficient *compressed adjacency list* for record each node and its out-neighbors across various layers of CHC-Index Υ (§3.3.1). We then introduce the physical storage procedure and provide the effectiveness analysis in §3.3.2. We also discuss how CHC-Index perform deletes, inserts, and updates in §3.3.3.

3.3.1 Space-efficient Data Structure of CHC-Index.

The key of compressing multi-layer sub-indices is to store only one copy of out-neighbors that appear repeatedly across multiple layers. Considering the hierarchy of CHC-Index, an intuitive approach is to record the *Maximum Co-occurrence Layer* (MCL), denoted by k_{mcl}, between a node and its out-neighbors, and maintain only one copy of out-neighbors across sub-indices Υ_k for $\forall k \leq k_{\text{mcl}}$. Generally, MCL can be defined based on the concept of coreness or trussness for core-based or truss-based C_{model}. We next take core-based C_{model} as an example to show the definition of MCL.

Definition 6 (Coreness [42, 48]). *Given a node $v \in V_G$, the coreness of v is the largest k of the k-core contains v, denoted by $c(v)$.*

The coreness $c(v)$ indicates that v belongs to every k-core with $k \leq c(v)$, i.e., $c(v)$ is the maximum layer that v occurs. So, we define the *Maximum Co-occurrence Layer* between two nodes as follows.

$A_\Upsilon = \{A(v_1), A(v_2), \cdots, A(v_n)\}$

	Node	1st element		2nd element		3rd element		···
	v_8	v_5	$k_{\text{mcl}}(v_8, v_5) = 2$	v_9	$k_{\text{mcl}}(v_8, v_9) = 2$			
$A(v_5)$	v_5	v_6	$k_{\text{mcl}}(v_5, v_6) = 2$	v_3	$k_{\text{mcl}}(v_5, v_3) = 3$	v_2	$k_{\text{mcl}}(v_5, v_2) = 3$	
		$E_{56} : \langle v_6, 2\rangle$		$E_{53} : \langle v_3, 3\rangle$		$E_{52} : \langle v_2, 3\rangle$		

Figure 8: Compressed adjacency list of CHC-Index Υ

Definition 7 (**Maximum Co-occurrence Layer (MCL)**). *Given two nodes u, v in the CHC-Index Υ, we define the MCL of u and v as $k_{\text{mcl}}(u, v) = \min\{c(u), c(v)\}$, which is simplified as k_{mcl}.*

Lemma 3. *Given two nodes u and v with $k_{\text{mcl}}(u, v)$, then they cannot possibly be mutual out-neighbors at any Υ_k for $\forall k > k_{\text{mcl}}(u, v)$.*

Since u and v has an MCL of k_{mcl}, it means the largest layer sub-index that include both u and v is $\Upsilon_{k_{\text{mcl}}}$. Hence, they cannot possibly be out-neighbors of each other at any Υ_k for $\forall k > k_{\text{mcl}}(u, v)$.

Based on Lemma 3, we can easily decide the boundary for compressed storage, i.e., the layer beyond which two nodes are not mutual out-neighbors. So, we integrate MCL with adjacency list to present the compressed adjacency list of Υ, denoted by A_Υ. As shown in Figure 8, $A_\Upsilon = \{A(u_1) \cdots A(u_n)\}$ consists of $n = |V_G|$ entries, each $A(u_i) = \langle u_i, \{E_{ij} : \langle v_j, k_{\text{mcl}}\rangle\}\rangle$ records u_i's out-neighbors across different layers as a set of elements $\{E_{ij} : \langle v_j, k_{\text{mcl}}\rangle\}$, where $v_j \in N_{\text{out}}(u_i)$ and k_{mcl} is the MCL between u_i and v_j.

Example 7. *Figure 8 illustrates two entries of nodes v_5 and v_8 from the hierarchical overlay of Υ in Figure 4 (middle part). Note that, v_3 is an out-neighbor of v_5 that appeared three times in $\Upsilon_1 \sim \Upsilon_3$ and has a MCL with v_5 as $k_{\text{mcl}} = 3$. To save the space for storing v_3, we record v_3 only once as an element $E_{53} = \langle v_3, 3\rangle$.*

3.3.2 Physical Storage Procedure.

We adopt an append-write strategy to update the compressed adjacency list A_Υ layer-by-layer. Algorithm 8 outlines the procedure for storing a sub-index Υ_k in A_Υ, where A_Υ is initialized as $\emptyset$. Since some nodes in Υ_k are inherited from Υ_{k-1} with out-neighbors unchanged, we only consider nodes whose out-neighbors have changed in Υ_k and update their entries in A_Υ. For each node $u_i \in V_{\Upsilon_k}$, we do the followings. (1) We first identify nodes with changed out-neighbors in Υ_k, i.e., out-neighbors from $N^k_{\text{out}}(u_i) \setminus N^{k-1}_{\text{out}}(u_i)$, where $N^k_{\text{out}}(u_i)$ are u_i's out-neighbors in Υ_k. Since u_i's out-neighbors in Υ_{k-1} have already in A_Υ, we access A_Υ via `getNeighbors` (Algorithm 9) to get $N^{k-1}_{\text{out}}(u_i)$ (line 2), while out-neighbors in Υ_k are directly obtained from the input Υ_k. (2) For each out-neighbor $v_j \in N^k_{\text{out}}(u_i) \setminus N^{k-1}_{\text{out}}(u_i)$, we compute the MCL of u_i and v_j as k_{mcl} and append a new element $E_{ij} : \langle v_j, k_{\text{mcl}}\rangle$ to the end of entry $A(u_i)$ (lines 3-6). (3) We finally update A_Υ with the refined $A(u_i)$ (line 7).

Example 8. *Recall the Υ with three layers in Figure 4 (middle). Let us consider node v_5 in Υ_1, after invoking* `physicalStore`*(A_Υ, Υ_1), we have an entry of v_5 with two elements as $A(v_5) = \{v_6, \{E_{56} : \langle v_6, 2\rangle, E_{53} : \langle v_3, 3\rangle\}\}$. Since v_5 and its out-neighbors in Υ_2 are inherited from Υ_1 with unchanged, we do not update $A(v_5)$. But in Υ_3, due to the removal of v_6, we add a new out-neighbor v_2 to v_5 via* `regularGraph` *(Algorithm 6). So, we append a new element $E_{52} : \langle v_2, 3\rangle$ to the end of $A(v_5)$ and update it as $A(v_5) = \{v_5, \{E_{56}, E_{53}, E_{52}\}\}$.*

Given a node u_i, a desired layer k, and the compressed adjacent list A_Υ, Algorithm 9 obtains u_i's out-neighbors in Υ_k from A_Υ as

Algorithm 8: `physicalStore`(A_Υ, Υ_k)

Input: Compressed adjacency list A_Υ, sub-index Υ_k
Output: Updated compressed adjacency list A_Υ

1 **for** $\forall u_i \in V_{\Upsilon_k}$ **do**
2 $\quad N^{k-1}_{\text{out}}(u_i) \leftarrow$ `getNeighbors`($A_\Upsilon, k-1, u_i$);
$\quad$ `// Update` A_Υ `using` u_i`'s new out-neighbors in` Υ_k
3 $\quad$ **for** $\forall v_j \in N^k_{\text{out}}(u_i) \setminus N^{k-1}_{\text{out}}(u_i)$ **do**
4 $\quad\quad k_{\text{mcl}} \leftarrow \min\{c(u_i), c(v_j)\}$; `/* compute MCL of` u_i, v_j `*/`
5 $\quad\quad A(u_i) \leftarrow A(u_i) \cup \langle v_j, k_{\text{mcl}}\rangle$;
6 $\quad$ **end**
7 $\quad A_\Upsilon \leftarrow$ update A_Υ with $A(u_i)$;
8 **end**
9 **return** A_Υ;

Algorithm 9: `getNeighbors`(A_Υ, k, u_i)

Input: Compressed adjacency list A_Υ, desired layer k, query u_i
Output: Out-neighbors of u_i in k-th layer sub-index Υ_k

1 $N^k_{\text{out}}(u_i) \leftarrow \emptyset$;
2 $A(u_i) \leftarrow$ get the entry of u_i from A_Υ;
3 **for** $\forall E_{ij} : \langle v_j, k_{\text{mcl}}\rangle \in A(u_i)$ retrieved in order **do**
4 $\quad$ **if** $k_{\text{mcl}} \geq k$ **then**
5 $\quad\quad N^k_{\text{out}}(u_i) \leftarrow N^k_{\text{out}}(u_i) \cup v_j$;
6 $\quad$ **end**
7 $\quad$ **if** $|N^k_{\text{out}}(u_i)| == R$ **then**
8 $\quad\quad$ **break**;
9 $\quad$ **end**
10 **end**
11 **return** $N^k_{\text{out}}(u_i)$;

follows. (1) We initialize u_i's out-neighbors in Υ_k as $\emptyset$ and get the entry $A(u_i)$ from A_Υ (lines 1-2). (2) We sequentially access each element $E_{ij} : \langle v_j, k_{\text{mcl}}\rangle$ in $A(u_i)$ and check if the MCL of v_j and u_i is greater than or equal to the desired layer k, i.e., $k_{\text{mcl}} \geq k$. If yes, we update $N^k_{\text{out}}(u_i)$ with v_j (lines 4-6). Otherwise, v_j is not an out-neighbor of u_i (Lemma 3) and is ignored. (3) Since Υ_k is R-regular graph, each node has R out-neighbors. So, we repeat above until R nodes with $k_{\text{mcl}} \geq k$ are retrieved (lines 7-9) and return the final $N^k_{\text{out}}(u_i)$. We next prove that the retrieved R nodes from $A(u_i)$ with $k_{\text{mcl}} \geq k$ are exactly all the out-neighbors of u_i in Υ_k.

Lemma 4. *Given a node u, suppose v is one of u's out-neighbor in Υ_k but not an out-neighbor in Υ_{k-1}, i.e., $v \in N^k_{\text{out}}(u) \setminus N^{k-1}_{\text{out}}(u)$, then* `physicalStore`*($A_\Upsilon, \Upsilon_k$) would append a new element $\langle v, k_{\text{mcl}}\rangle$ with $k_{\text{mcl}}(u, v) \geq k$ to the end of entry $A(u)$.*

Proof. We complete the proof by contradiction. Suppose that `physicalStore`(A_Υ, Υ_k) appends an element $\langle v, k_{\text{mcl}}\rangle$ with $k > k_{\text{mcl}}$ to $A(u)$. From Lemma 3, nodes u and v are not mutual out-neighbors in Υ_k because $k > k_{\text{mcl}}$. This contradicts the assumption of $v \in N^k_{\text{out}}(u)$. Thus, $k_{\text{mcl}}(u, v) \geq k$ holds. □

Lemma 5. *After performing the physical storage of Υ_k, for $\forall u \in V_{\Upsilon_k}$, its entry $A(u) \in A_\Upsilon$ must contain R elements $\{\langle v, k_{\text{mcl}}\rangle\}$ of which each v and u has an MCL satisfying $k_{\text{mcl}}(u, v) \geq k$.*

Proof. Suppose a node u in Υ_k has $m \leq R$ out-neighbors inherited from Υ_{k-1}, then these m out-neighbors have an MCL $k_{\text{mcl}} \geq k$ based on Lemma 3. Moreover, since we maintain Υ_k as a R-regular graph, u should have another exclusive $R - m$ out-neighbors in Υ_k. So, according to Lemma 4, these $R - m$ out-neighbors must have an MCL $k_{\text{mcl}} \geq k$. Therefore, a node u in Υ_k has R out-neighbors with

$k_{\text{mcl}} \geq k$, indicating that $A(u)$ must contain R elements $\{\langle v, k_{\text{mcl}}\rangle\}$ of which each v and u has an MCL $k_{\text{mcl}}(u, v) \geq k$. □

Theorem 3. *Using Algorithm 9 ensures that all of the R out-neighbors of a node u in Υ_k can be correctly returned.*

Proof. According to Lemma 5, there must exist R elements in u's entry $A(u)$ after performing `physicalStore`(A_Υ, Υ_k), representing the R out-neighbors of u in Υ_k. Since we adopt append-write strategy to update A_Υ layer-by-layer, elements added after k-th layer with $k_{\text{mcl}} \geq k$ will appear after these existing R elements, without changing the positions of these R elements. Hence, in Algorithm 9, the R nodes with $k_{\text{mcl}} \geq k$ retrieved by sequentially accessing $A(u)$ are exactly the R out-neighbors of u in Υ_k. □

3.3.3 Maintenance of CHC-Index.

In real-world applications, graphs often change dynamically. We briefly introduce how CHC-Index supports these changes, including deletions, insertions, and updates. Similar to [38, 54], we handle each operation as follows. (1) *Deletion.* We employ a lazy deletion strategy. when a node $v \in V_G$ is deleted, we mark its entry $A(v) \in A_\Upsilon$ invalid, as well as the elements in other node entries where v is an out-neighbors. We then introduce a post-verification step to skip all invalid nodes after the search. (2) *Insertion.* For each new node v, we treat it as a query node q and run `greedySearch` (Algorithm 3) to identify all visited nodes on the converging path to q as candidates for neighbor refinement. We then run `robustPrune++` (Algorithm 7) to assign R out-neighbors to v. (3) *Update.* Updates are managed as a combination of deletion followed by insertion. These strategies are effective for incremental maintenance with minor graph changes. However, if the graph changes significantly, reconstructing the index is necessary to ensure efficiency and effectiveness. In §5.7, we study CHC-Index's scalability in dynamic scenarios.

3.3.4 Complexity Analysis.

Since we construct the lowest layer sub-index as an R-regular graph, the sapce complexity for storing n nodes ($n = |V_G|$) and their R out-neighbors in the compressed adjacency list A_Υ is $O(nR)$. When storing Υ_k for $k \in (k_{\text{mix}}, k_{\text{max}}]$, we invoke `regularGraph` to complete out-neighbors for an average of $m_k R$ nodes (m_k represents the number of nodes that belong to the $(k-1)$-core community but excluded from k-core community), resulting in $m_k R$ new elements added to A_Υ. Therefore, the space complexity for storing all Υ_k is $O(\sum m_i R) < O(nR)$. Hence, the total complexity of A_Υ is $O(nR)$.

4 Community Search with CHC-Index

Given a graph G, a CSAG problem with an original solution S, and a CHC-Index Υ that is stored as a compressed adjacency list A_Υ, the community search with index involves two steps: (1) *high-quality candidate nodes retrieval from A_Υ* and (2) *community search against retrieved nodes using S to return a desired community from G.* We also provide the implementation details of the integration of CHC-Index with existing mainstream CSAG problems in Appendix A.

Candidate nodes retrieval. Given an input cohesiveness parameter k and a query node q, we first leverage `greedySearch`(s,q,m,L,$\eta()$) to recall top-m candidate nodes V^* that are similar to q in terms of the composite distance $\eta()$. Here, s is the start node of sub-index Υ_k

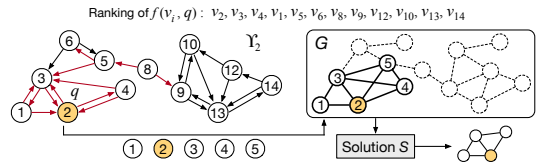


Figure 9: An example of community search with CHC-Index

and L is the search width. During the construction of CHC-Index, we compute s as the medoid of each sub-index Υ_k and maintain all pairs of $\langle k, s\rangle$ in a map, from which we can directly obtain the required start node of Υ_k. Unlike the use of `greedySearch` during CHC-Index construction, we invoke `greedySearch` here with some differences. First, we set m as a fraction r of all nodes in a specific Υ_k, denoted by $m = r \cdot |V_{\Upsilon_k}|$. A large r increases the scale of candidate nodes, thus extending the community search time. In our experimental study, we show the effect of r on the community search's performance. Second, we invoke `getNeighbors` (Algorithm 9) to get the out-neighbors (at the k-th layer) of the closest node to q from the compressed adjacency list A_Υ at line 4 of `greedySearch`, rather than getting out-neighbors from a specific sub-index Υ_k.

Community search against retrieved nodes. If the retrieved nodes V^* are of sufficiently high quality, possibly including majority or even all members of the desired community of q, then the induced graph $G[V^*] \subseteq G$ formed by V^* is likely to preserve the structure of the desired community. By applying the original solution S to the given CSAG problem on $G[V^*]$, we expect to find a community C^* similar to or exactly the same as the desired community. To achieve this, we first extract $G[V^*]$ from G as the source graph for running the original solution S. Then, we take $G[V^*]$, query node q, and cohesiveness parameter k as inputs to S to find a feasible community C^*. Since $G[V^*]$ is smaller than G, running S on $G[V^*]$ is reasonably more efficient than that on G. As long as we can get the high-quality V^* from A_Υ, it is possible to enhance the efficiency of community search while ensuring community quality.

Example 9. *Given the graph G in Figure 4, we use v_2 as query node to find a connected k-core ($k = 2$) community from G that contains v_2. Figure 9 illustrates the process of CS with CHC-Index. First, we retrieve candidate nodes from the 2-nd layer sub-index Υ_2 (illustrated in Figure 4) given the start node $s = v_8$ and $r = 40\%$. Since Υ_2 includes 12 nodes, we need to retrieve $m = 40\% \times 12 \approx 5$ nodes. As a result, 5 nodes $V^* = \{v_1, v_2, v_3, v_4, v_5\}$ are returned by* `greedySearch` *given the ranking of all nodes in ascending order of the attribute distance $f()$ to v_2 (top). Next, we obtain the induced graph of V^* from G, which serves as the input for the original solution S of a CSAG problem, ultimately obtaining a result community. In §5.8, we provide a detailed case study on Facebook to demonstrate how CHC-Index accelerates VAC [35].*

Effectiveness analysis. We strive for the community search utilizing the CHC-Index to yield results that are either comparable to or surpass those of the original algorithm. It is evident to see that the effectiveness of CS with CHC-Index depends on the quality of m nodes V^* retrieved by `greedySearch` in the first step. The higher the quality of V^*, the better the community obtained using the original solution S against on the induced graph $G[V^*]$. According to [45], the search width L and # retrieval nodes m ($L \geq m$) are crucial for the quality of retrieved nodes. A larger L and m usually result in higher quality of retrieved nodes, however, at the cost of

Table 2: Statistics of datasets

Datasets ↓	#Nodes	#Edges	Density	$d_{\max}$	d_{avg}	$k^c_{\max}$	$k^t_{\max}$
Facebook	4,039	88,234	21.85	1,045	43.69	117	96
Twitch	168,114	6,797,557	40.43	35,259	80.86	149	125
DBLP	682,819	1,951,209	2.86	345	3.75	113	114
IMDB	2,875,685	9,705,602	3.38	591	4.42	353	187
LiveJournal	3,997,962	34,681,189	8.67	14,815	17.34	321	315

increased search time. In §5.4, we analyze the trade-off between the efficiency and effectiveness of community search using CHC-Index.

5 Experiments

We conducted an experimental study to answer the following questions. Our code [14] was implemented in C++ and run on a Linux server with 2.1 GHz CPU and 64 GB memory.

Q1. How do CSAG algorithms perform in terms of efficiency and effectiveness when using and not using CHC-Index? (**§5.2-5.4**)
Q2. What is the effect of two optimizations on CHC-Index? (**§5.5**)
Q3. What are the time and space overhead of CHC-Index? (**§5.6**)
Q4. How does CHC-Index performs in dynamic scenarios? (**§5.7**)
Q5. How does CHC-Index help to find feasible communities in real-world datasets? (case study in **§5.8**)
Q6. How is the parameter sensitivity of CHC-Index? (**§5.9**)

5.1 Experimental Setup

Datasets. Table 2 summarizes some statistics from five real-world datasets, including the density, maximum (average) degree $d_{\max}$ (d_{avg}), the largest k of k-core ($k^c_{\max}$) and k-truss ($k^t_{\max}$) in a graph. (1) Facebook [36], (2) Twitch [40] and (3) LiveJournal [55] are social networks. (4) DBLP [1] provides relationships among authors, papers, and venues, with attributes such as research interests, # publications, h-index and # citations for authors. (5) IMDB [2] provides relationships among actors, directors, and movies, with attributes like category, # movies for actors; genres, ratings for movies.

Methods. We integrate CHC-Index with four representative CSAG methods VAC [35], ATC [28], rKACS [58], and SEA [51] to obtain four Index-CSAG methods Index-VAC, Index-ATC, Index-rKACS, and Index-SEA. In our experiments, we implement the fastest versions of VAC, ATC, and rKACS, that are ATruss, LocATC, and Incremental-rKACS. For simplicity, we still refer them as VAC, ATC, and rKACS. Besides, SEA supports both homogeneous and heterogeneous graphs, while others are only for homogeneous graphs. To perform methods except SEA on a heterogeneous graph, we follow [51] to a homogeneous graph given a meta-path and run original methods on the converted graph to find desired communities.

Parameters and queries. We evaluate all methods using appropriate parameters, including β that controls the index's navigability, the fraction r of nodes retrieved from CHC-Index, and the desired structural cohesiveness level k. Similar to [35], we randomly select 300 query nodes for each dataset and report the average result. We highlight the best (suboptimal) results in bold (with underlines).

Metrics. We measure efficiency using response time and evaluate efficiency improvement with CHC-Index using speedup. We assess the effectiveness using attribute cohesiveness metric $F()$ and evaluate the effectiveness improvement with CHC-Index using `Rerror@F` (Definition 5). For Facebook, which includes 10 human-annotated ground-truth communities, we also follow [28, 51] to evaluate the effectiveness using F_1-score w.r.t. ground-truth. Given a returned community C and a ground-truth community $C^\star$, the F_1-score w.r.t. ground truth is $F_1 = \frac{2P \cdot R}{P+R}$, where $P = \frac{|V_C \cap V_{C^\star}|}{|V_C|}$ represents the precision and $R = \frac{|V_C \cap V_{C^\star}|}{|V_{C^\star}|}$ represents the recall.

Remark. VAC and SEA are approximate methods with `opt` = min, a negative `Rerror@F` further from zero indicates better results. rKACS is an exact method with `opt` = max, a negative `Rerror@F` closer to zero shows better results. ATC is an approximate method with `opt` = max, a larger positive `Rerror@F` indicates better results. For clarity, in §5.3, we highlight `Rerror@F` in red for better or equal attribute cohesiveness $F()$ and in blue for worse $F()$.

5.2 Efficiency Evaluation

We build the CHC-Index using default parameters of $\alpha = 1.2$ and $\beta = 0.8$, and evaluate the efficiency of all methods across various datasets with a default $k = 15$. Table 3 shows the results obtained by setting an appropriate $r \in [0.1, 0.5]$ that achieves a balance between speedup and `Rerror@F`. *Note that the index-based method achieves an average speedup of 96.3×, 2.6×, 21.9×, and 2.1× across datasets for VAC, ATC, SEA, and rKACS, respectively.* This primarily stems from CHC-Index's capability to retrieve a set of high-quality nodes that belong to the same k-core or k-truss community as the query q and have similar attributes to q, thus reducing the search space for community search. *Moreover, efficiency gains are noticeable in larger and denser datasets. For instance, Index-VAC achieves a speedup of 142 × and 222.9 × on the largest LiveJournal and densest Twitch, which are greater than others.* This is because larger and denser graphs have larger search spaces. So, even filtering out a small number of nodes by CHC-Index can well-reduce the search space.

5.3 Effectiveness Evaluation

Table 4 shows the effectiveness results, with the relative error of the attribute cohesiveness `Rerror@F` highlighted in red for better or equal attribute cohesiveness $F()$ and in blue for worse $F()$. *All index-based methods achieve a comparable effectiveness to original methods.* Especially, Index-VAC on DBLP, Index-ATC on LiveJournal, and Index-SEA on Twitch yield much better communities with improved $F()$. For Index-rKACS on DBLP, the same community as the exact rKACS is found. This is because our CHC-Index considers both attribute and structural cohesiveness, ensuring high-quality node retrieval for community search. We also report F_1-score w.r.t. ground-truth on 10 ego-networks of Facebook, where index-based methods still achieve comparable or better results.

5.4 Trade-off of Effectiveness and Efficiency

Experiments in §5.3-5.2 illustrate how CHC-Index affects the community search's effectiveness and efficiency under default parameters. While in this section, we study the trade-off between effectiveness and efficiency. By jointly adjusting the search width L and the number of retrieval nodes m in `greedySearch`, we retrieve candidate nodes of varying quality from the CHC-Index, based on which we run community search and collect data in the form of ⟨`Rerror@F`, speedup⟩. To enhance data reliability, we run multiple tests for each parameter configuration and present the results for three methods on DBLP (yellow), IMDB (red), and Twitch (blue), along with a 95% confidence interval (shaded area) in Figure 11.

Table 3: Efficiency: response time (ms) and speedup w.r.t. response time for different methods

Methods →	VAC vs. Index-VAC			ATC vs. Index-ATC			SEA vs. Index-SEA			rKACS vs. Index-rKACS		
	VAC	Index-VAC	Speedup	ATC	Index-ATC	Speedup	SEA	Index-SEA	Speedup	rKACS	Index-rKACS	Speedup
Facebook	251.44	37.81	6.96	596.69	426.56	1.32	109.66	45.41	2.49	74.93	62.39	1.22
Twitch	28462.92	178.30	**222.92**	127453.26	34585.47	3.68	31290.41	1680.69	18.76	7293.32	1960.55	**3.87**
DBLP	81.14	2.28	55.86	166.70	88.32	2.04	638.37	46.58	**62.63**	87.94	72.26	1.41
IMDB	167.10	3.91	53.98	1504.90	1114.18	1.21	1458.98	172.06	11.86	210.96	148.68	1.42
LiveJournal	16159.89	115.29	142.00	10081.11	1923.55	**4.74**	27545.46	1658.46	13.80	6499.27	2880.56	2.26

Table 4: Effectiveness: community attribute cohesiveness $F()$ and Rerror@F (red for better or equal $F()$ and blue for worse $F()$)

Methods →	VAC vs. Index-VAC			ATC vs. Index-ATC			SEA vs. Index-SEA			rKACS vs. Index-rKACS		
	VAC	Index-VAC	Rerror@F	ATC	Index-ATC	Rerror@F	SEA	Index-SEA	Rerror@F	rKACS	Index-rKACS	Rerror@F
Facebook	0.84	0.80	-0.05	875	881	0.02	0.30	0.29	-0.02	6.17	6.10	-0.01
Twitch	0.41	0.36	-0.10	136944	134151	0.02	0.09	0.08	**-0.08**	20.07	19.95	-0.0022
DBLP	0.87	0.71	**-0.17**	1955	1377	0.18	0.43	0.44	0.02	1.82	1.82	**0.00**
IMDB	0.02	0.02	-0.16	1442	1147	-0.01	0.020	0.019	-0.01	5.12	5.06	-0.0021
LiveJournal	0.98	0.82	-0.16	24087	23836	**0.30**	0.37	0.36	-0.03	8.43	8.43	**0.00**

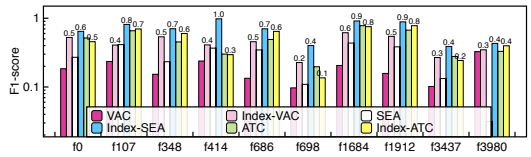


Figure 10: F_1-score on each ego-network of Facebook

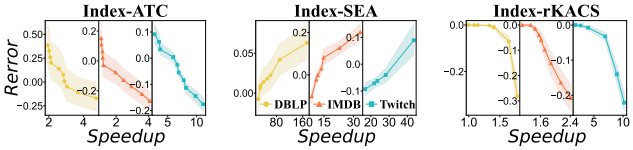


Figure 11: Trade-off of effectiveness and efficiency

The X-axis represents the speedup and the Y-axis represents the Rerror@F. Since ATC is an approximate method with opt = max, a larger positive Rerror@F indicates that Index-ATC achieves a better attribute cohesiveness F than that of original ATC (the upper the Y-axis value, the better). In contrast, SEA is an approximate method with opt = min, a negative Rerror@F further from zero indicates that Index-SEA has a better attribute cohesiveness than the original SEA (the lower the Y-axis value, the better). While for rKACS, it is an exact method with opt = max, a negative Rerror@F closer to zero shows better results (the upper the Y-axis value, the better). From the results, we found that achieving better effectiveness requires more time to retrieve high-quality nodes, which in turn reduces the speedup introduced from CHC-Index, and vice versa.

5.5 Ablation Study

To verify the effect of two optimizations on CHC-Index, we implement three versions of VAC: Index-VAC, Index-VAC w/o O2, and Index-VAC w/o O1+O2, where O1 and O2 are optimizations on robustPrune and distance metric, respectively. In Figure 12, each bar shows the runtime with two parts: lower part is the time of candidates retrieval from CHC-Index and upper part is the time of community search from candidates. *We found that both O1 and O2 contribute to performance improvements. In particular, the effect of O2 is superior to that of O1.* This is because O2 considers both structural closeness and attribute distance, allowing for the retrieval of high-quality candidates with a smaller r. Regarding O1, it enhances the navigability of CHC-Index, thereby optimizing the converge time during the candidate nodes retrieval phase. In Figure 12, the time for candidate nodes retrieval increases when O1 is removed.

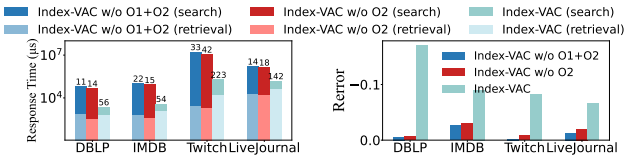


Figure 12: Ablation results of Index-VAC

5.6 Index Overhead

Table 5 shows the index construction and storage overhead. The larger the graph, the more the time is required for constructing the first layer sub-index $\Upsilon_{k_{\min}}$ (i.e., the Vamana graph). While for other layer sub-index Υ_k with $k \in (k_{\min}, k_{\max}]$, we perform robustPrune++ on a small subgraph induced from Υ_{k-1}, which is more efficient. The value of $k_{\max}$ also affects the efficiency. The larger the $k_{\max}$, the more the time is required to run robustPrune++ for all Υ_k. Moreover, it is expected that larger graphs require more storage space. We also present the ratio of the index size to the original data in the last row, also demonstrating the space efficiency.

5.7 Scalability Analysis on Dynamic Graphs

In this section, we conduct an experiment to study the scalability of CHC-Index on dynamic graphs with the following experimental settings. First, we divide a graph G into two parts: a subgraph $G_i \subseteq G$ with $X\%$ nodes of G and the remaining $(100 - X)\%$ part $\tilde{G}_i = G \setminus G_i$. We vary the fraction of nodes $X \in \{1, 2, 4, 8, 16, 32\}$, generating six subgraphs $\{G_1, \cdots, G_6\}$. For example, given $X = 2\%$, we have a subgraph $G_1 \subseteq G$ with 1% nodes of G and another subgraph $\tilde{G}_1 = G \setminus G_1$ with the remaining 99% nodes. For deletions, we build an index Υ for G, then delete $X\%$ nodes and update the index as Υ' following the process in §3.3.3. Additionally, we build an index $\tilde{\Upsilon}$ for $\tilde{G}_i$. We perform the same queries on both Υ' and $\tilde{\Upsilon}$, and report Rerror@F in Table 6. We expect that the effectiveness of using Υ' can be comparable with that of using $\tilde{\Upsilon}$. Similarly, for insertions, we update $\tilde{\Upsilon}$ as $\tilde{\Upsilon}'$ by inserting $X\%$ nodes following the process in §3.3.3. Then, we perform the same queries on both $\tilde{\Upsilon}'$ and Υ, and report Rerror@F in Table 7. We expect that the effectiveness of using $\tilde{\Upsilon}'$ can be comparable with that of using Υ. We highlight Rerror@F in red for better or equal attribute cohesiveness and in blue for worse. From the results, we observed that the updated index achieves comparable effectiveness w.r.t. the reconstructed index for small graph changes, but gradually deteriorates as the

Table 5: The construction (sec) and storage (MB) overhead of CHC-Index for different methods

Metrics ↓	Facebook				Twitch				DBLP				IMDB				LiveJournal			
	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS
Time (s)	2	2	2	2	1201	1014	1865	1014	174	150	150	150	598	756	437	756	6540	6793	7920	6793
Index size	0.22	0.20	0.22	0.20	18	18	20	18	26	26	26	26	54	55	55	55	493	480	505	480
Dataset size	36	36	36	36	205.6	205.6	205.6	205.6	322.6	322.6	322.6	322.6	752.7	752.7	752.7	752.7	2719	2719	2719	2719
Ratio	0.61%	0.56%	0.61%	0.56%	8.75%	8.75%	9.73%	8.75%	8.06%	8.06%	8.06%	8.06%	7.17%	7.31%	7.31%	7.31%	18.1%	17.7%	18.6%	17.7%

Table 6: `Rerror`@F on dynamic graphs (deletion)

Graph ↓	DBLP				Twitch			
	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS
$G \setminus G_1 \rightarrow \tilde{G}_1$	-1.4×10^{-3}	0.01	3.1×10^{-4}	-1.4×10^{-4}	-6.3×10^{-3}	0.02	6.3×10^{-4}	0.00
$G \setminus G_2 \rightarrow \tilde{G}_2$	-5.3×10^{-4}	-0.03	4.3×10^{-3}	-1.3×10^{-3}	-2.2×10^{-3}	-0.01	4.2×10^{-3}	-2.7×10^{-4}
$G \setminus G_3 \rightarrow \tilde{G}_3$	1.3×10^{-3}	-0.07	0.01	-3.4×10^{-3}	-5.9×10^{-4}	-0.02	6.1×10^{-3}	-2.7×10^{-4}
$G \setminus G_4 \rightarrow \tilde{G}_4$	0.01	-0.16	0.01	-0.01	2.4×10^{-3}	-0.08	0.01	-3.4×10^{-3}
$G \setminus G_5 \rightarrow \tilde{G}_5$	0.02	-0.29	0.03	-0.02	0.01	-0.22	0.01	-0.01
$G \setminus G_6 \rightarrow \tilde{G}_6$	0.07	-0.55	0.03	-0.03	0.04	-0.54	0.06	-0.04

Table 7: `Rerror`@F on dynamic graphs (insertion)

Graph ↓	DBLP				Twitch			
	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS
$\tilde{G}_1 \cup G_1 \rightarrow G$	-1.1×10^{-3}	-0.01	1.5×10^{-3}	0.00	-0.01	-0.02	3.6×10^{-3}	0.00
$\tilde{G}_2 \cup G_2 \rightarrow G$	0.00	-0.03	0.01	-1.4×10^{-4}	1.5×10^{-3}	-0.04	0.01	-3.1×10^{-4}
$\tilde{G}_3 \cup G_3 \rightarrow G$	8.4×10^{-3}	-0.09	0.01	-0.01	2.1×10^{-3}	-0.07	0.02	-4.9×10^{-4}
$\tilde{G}_4 \cup G_4 \rightarrow G$	0.01	-0.18	0.01	-0.01	0.02	-0.13	0.04	-6.7×10^{-4}
$\tilde{G}_5 \cup G_5 \rightarrow G$	0.03	-0.32	0.03	-0.02	0.03	-0.25	0.06	-3.6×10^{-3}
$\tilde{G}_6 \cup G_6 \rightarrow G$	0.08	-0.61	0.08	-0.05	0.16	-0.42	0.08	-0.03

Table 8: Time (sec) for index maintenance (insertion)

Graph ↓	DBLP				Twitch			
	VAC	ATC	SEA	rKACS	VAC	ATC	SEA	rKACS
$\tilde{G}_1 \cup G_1 \rightarrow G$	13	14	14	14	11	13	11	13
$\tilde{G}_2 \cup G_2 \rightarrow G$	28	29	28	29	24	25	22	25
$\tilde{G}_3 \cup G_3 \rightarrow G$	35	38	37	38	37	33	36	33
$\tilde{G}_4 \cup G_4 \rightarrow G$	77	77	75	77	72	70	71	70
$\tilde{G}_5 \cup G_5 \rightarrow G$	156	153	155	153	137	178	173	178
$\tilde{G}_6 \cup G_6 \rightarrow G$	297	312	300	312	247	240	244	240

scale of deletions and insertions increases. This suggests that when the graph undergoes significant changes, reconstructing the index would be a viable solution. Since we adopt lazy deletion strategy that only requires marking nodes as invalid, the complexity is $O(1)$, making the update overhead negligible. In contrast, the insertion operation involves calling `greedySearch` and `robustPrune`, which incurs some index maintenance overhead. So, we report only the index maintenance time for insertions in Table 8. The results show that this overhead remains modest with minor graph changes but increases as more changes are made to G.

5.8 Case Study

We conducted a case study on Facebook given a query node q (red) with ID = 20 and $k = 4$. We marked community members returned by VAC (implemented as the approximate version ATruss) and Index-VAC in purple. Nodes marked in blue and purple represent the search space for both algorithms. For VAC, it is the maximum 4-truss containing q, while for Index-VAC, it is the induced graph formed by 25 nodes retrieved from Υ_4. Since the search space of VAC (with 260 nodes) is much larger than that of Index-VAC (with 26 nodes), VAC needs to iteratively remove more nodes (215 nodes) than that of Index-VAC (15 nodes) to find a feasible community, resulting in Index-VAC achieves a 40× speedup compared to VAC. Moreover, the average attribute distance of all nodes to q for VAC and Index-VAC are 0.64 and 0.59, respectively. The community attribute cohesiveness $F(f(), C)$ (i.e., the largest attribute distance $f()$ of the most dissimilar nodes marked in yellow) for VAC and Index-VAC are 0.93 and 0.87, respectively. Both results

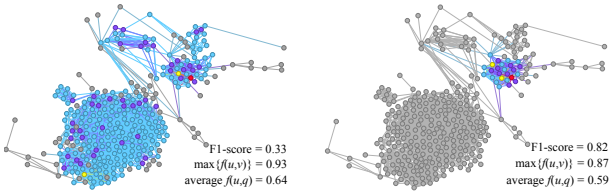


Figure 13: Case study: VAC vs. Index-VAC on Facebook

prove that our community is more cohesion in attributes than VAC (`Rerror`@$F = -0.06$). Additionally, regarding the F_1-score w.r.t. ground-truth (a 9 nodes community) of Facebook, VAC and Index-VAC achieve a F_1-score of 0.33 and 0.82, respectively.

5.9 Parameter Sensitivity

We varied k, r, and β to study their effect. We show results of Index-VAC and Index-SEA on DBLP and LiveJournal in Figure 14-16. Since VAC and SEA are approximate methods with `opt` = min, a smaller negative `Rerror`@F indicates a better attribute cohesiveness $F()$.

Effect of k. In Figure 14, all methods achieve a `Rerror`@F (< 0), indicating our index's capability to retrieve good candidate nodes, thereby yielding better communities. The speedup increases as k increases, as a larger k means fewer nodes need to be retrieved from a small Υ_k for the search, therefore benefiting the speedup.

Effect of r. Figure 15 shows that speedup decreases as r increases since retrieving more nodes from index diminishes the efficiency of index-based methods. Additionally, `Rerror`@F worsens as r increases, since index-based methods gradually deteriorates back to the one without index, resulting in reduced effectiveness gains. We recommend setting a larger r for sparse graphs compared to dense graphs, for example, the optimal r for VAC on DBLP (density: 2.86) and LiveJournal (density: 8.67) are 0.45 and 0.25, respectively. This is because communities exhibit strong locality in dense graphs, allowing a smaller r to recall more community members within a limited search space. In contrast, a larger r is needed to cover more community members spread across the sparse graph.

Effect of β. From the results, we found that β does not significantly affect the performance, for example, the speedup fluctuates within a relatively small range from 52 to 56 for VAC on DBLP. But when β is large enough, it would reduce the lower bound of the angle, thus affecting navigability. We recommend setting $\beta = 0.8$ for optimal performance according to the experimental results in normal scenarios. While for resource-constrained scenarios (with limited memory), the degree bound R is typically reduced to decrease the # neighbors for each node in the index, thus reducing the size of the index at the expense of navigability. To counteract this, we recommend reducing $beta$ to increase the lower bound of the angle, helping to mitigate the negative effects of reducing R.

6 Related Work

CS on unattributed graphs. The key point of CS on unattributed graphs is how to define a community's structural cohesiveness. For

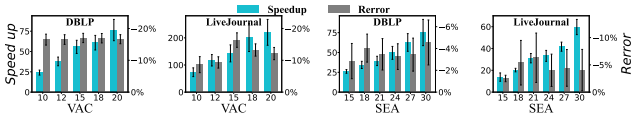


Figure 14: Effect of the structural cohesiveness parameter k

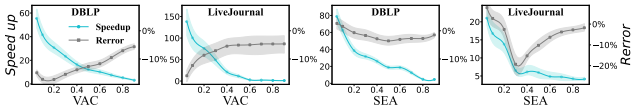


Figure 15: Effect of fraction r of nodes retrieved from index

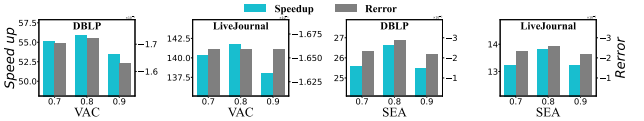


Figure 16: Effect of the angular factor β

homogeneous graphs, many works model the cohesive community based on k-core [5–7, 16, 22, 41], k-truss [4, 27, 29, 34], k-clique [15, 43, 59], k-edge [9, 26], and query-biased density model [52]. To extend them to heterogeneous scenarios, the concept of meta-path, denoted by $\mathcal{P}$, is often used to define some new community models, e.g., $(k, \mathcal{P})$-core [23], $(k, \mathcal{P})$-Btruss and $(k, \mathcal{P})$-Ctruss [57]. Many follow-up works use them for various downstream applications, i.e., expert finding in [50, 53] and influential CS via skyline influence vectors in [62]. Compared to unattributed graphs, attributed graphs excel in depicting node characteristics and are more widely used in practice. This motivates the study of CS on attributed graphs.

CS on attributed graphs. In this scenario, both structural and attribute cohesiveness should be considered in one model. Various studies define different attribute cohesiveness metrics and integrates them with structural cohesiveness for CS on homogeneous or heterogeneous attributed graphs [11, 12, 18–20, 28, 35, 49, 51, 61]. For example, [28] measure a community's attribute cohesiveness as the weighted sum of each attribute's coverage, with coverage computed as the ratio of nodes with exactly matched attributes. Similarly, [20] uses # shared attributes to measure cohesiveness, relying on equality matching too. [35] aims to minimize the maximum attribute distance in a community, but overlooks the similarity of nodes to the query q. [49] defines attributed cohesiveness as the average attribute distance across all pairs of nodes, while [51] relaxes it to the average attribute distance of all nodes to q.

CS on attributed graphs holds considerable research value, but both exact and approximate algorithms face efficiency challenges, limiting their practical applications. This motivates us to study a general cohesiveness-aware index to accelerate existing algorithms.

7 Conclusions

In this paper, we present a cohesiveness-aware hierarchical compressed index (CHC-Index) for community search over attributed graphs (CSAG), serving as a general accelerator to speedup existing algorithms of CSAG. We first present a proximity graph-based CHC-Index construction method, which supports various structure and attribute cohesiveness metrics. Then, we enhance its navigability and reliability through two optimizations. Finally, a compressed storage structure is designed for space-efficient storage. Integrating our index with mainstream CSAG algorithms and testing on real-world datasets demonstrates an average 30.7× speedup while maintaining a comparable or even better attribute cohesiveness.

References

[1] 2023. DBLP. http://dblp.uni-trier.de/xml/.
[2] 2023. IMDB. https://www.imdb.com/interfaces/.
[3] Yong-Yeol Ahn, James P Bagrow, and Sune Lehmann. 2010. Link Communities Reveal Multiscale Complexity in Networks. *nature* 466, 7307 (2010), 761–764.
[4] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: A Truss-equivalence Based Indexing Approach. *PVLDB* 10, 11 (2017), 1298–1309.
[5] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and Effective Community Search. *Data Min. Knowl. Discov.* 29, 5 (2015), 1406–1433.
[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) Algorithm for Cores Decomposition of Networks. *arXiv* cs.DS/0310049 (2003).
[7] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.
[8] Paul Burkhardt, Vance Faber, and David G. Harris. 2020. Bounds and Algorithms for Graph Trusses. *J. Graph Algorithms Appl.* 24, 3 (2020), 191–214. https://doi.org/10.7155/JGAA.00527
[9] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based Optimal Algorithms for Computing Steiner Components with Maximum Connectivity. In *SIGMOD*. 459–474.
[10] Jiazun Chen, Jun Gao, and Bin Cui. 2023. ICS-GNN$^+$: Lightweight Interactive Community Search via Graph Neural Network. *VLDBJ* 32, 2 (2023), 447–467.
[11] Lu Chen, Chengfei Liu, Kewen Liao, Jianxin Li, and Rui Zhou. 2019. Contextual Community Search Over Large Social Networks. In *ICDE*. 88–99.
[12] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-located Community Search in Large Scale Social Networks. *PVLDB* 11, 10 (2018), 1233–1246.
[13] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. 2008. Hierarchical Structure and The Prediction of Missing Links in Networks. *Nature* 453, 7191 (2008), 98–101.
[14] Code and datasets. 2023. Code and datasets. https://anonymous.4open.science/r/CHC-INDEX-0FEE/README.md.
[15] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online Search of Overlapping Communities. In *SIGMOD*. 277–288.
[16] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local Search of Communities in Large Graphs. In *SIGMOD*. 991–1002.
[17] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. *arXiv* (2024).
[18] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and Efficient Attributed Community Search. *VLDBJ* 26, 6 (2017), 803–828.
[19] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *PVLDB* 10, 6 (2017), 709–720.
[20] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.
[21] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A Survey of Community Search over Big Graphs. *VLDBJ* 29, 1 (2020), 353–392.
[22] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2019. Effective and Efficient Community Search Over Large Directed Graphs. *IEEE Trans. Knowl. Data Eng.* 31, 11 (2019), 2093–2107.
[23] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and Efficient Community Search over Large Heterogeneous Information Networks. *PVLDB* 13, 6 (2020), 854–867.
[24] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474.
[25] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: Lightweight Interactive Community Search via Graph Neural Network. *PVLDB* 14, 6 (2021), 1006–1018.
[26] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On Minimal Steiner Maximum-Connected Subgraph Queries. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2455–2469.
[27] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-truss Community in Large and Dynamic Graphs. In *SIGMOD*. 1311–1322.
[28] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *PVLDB* 10, 9 (2017), 949–960.
[29] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *PVLDB* 9, 4 (2015), 276–287.
[30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point

Nearest Neighbor Search on a Single Node. In *NeurIPS*, Vol. 32.
[31] Barrie Kersbergen, Olivier Sprangers, and Sebastian Schelter. 2022. Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale. In *SIGMOD*. 150–159.
[32] Jon M. Kleinberg. 2000. Navigation in A Small World. *Nature* 406 (2000), 845–845.
[33] Jianxin Li, Xinjue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most Influential Community Search over Large Social Networks. In *ICDE*. 871–882.
[34] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *SIGMOD*. 2183–2197.
[35] Qing Liu, Yifan Zhu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. VAC: Vertex-Centric Attributed Community Search. In *ICDE*. 937–948.
[36] Julian J. McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NIPS*. 548–556.
[37] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society. *Nature* 435, 7043 (2005), 814–818.
[38] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *SIGMOD* 1, 1 (2023), 54:1–54:27.
[39] Arnau Prat-Pérez, David Dominguez-Sal, Josep M. Brunat, and Josep Lluís Larriba-Pey. 2016. Put Three and Three Together: Triangle-Driven Community Detection. *ACM Trans. Knowl. Discov. Data* 10, 3 (2016), 22:1–22:42.
[40] Benedek Rozemberczki and Rik Sarkar. 2021. Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings. *arXiv* abs/2101.03091 (2021).
[41] Mauro Sozio and Aristides Gionis. 2010. The Community-search Problem and How to Plan a Successful Cocktail Party. In *KDD*. 939–948.
[42] Longxu Sun, Xin Huang, Ronghua Li, Byron Choi, and Jianliang Xu. 2020. Index-based Intimate-Core Community Search in Large Weighted Graphs. *IEEE Trans. Knowl. Data Eng.* (2020).
[43] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. 2013. Denser than the Densest Subgraph: Extracting Optimal Quasi-cliques with Quality Guarantees. In *KDD*. 104–112.
[44] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.
[45] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.
[46] Tong Wang, Cynthia Rudin, Daniel Wagner, and Rich Sevieri. 2013. Learning to Detect Patterns of Crime. In *ECML-PKDD*, Vol. 8190. 515–530.
[47] Weiqing Wang, Hongzhi Yin, Zi Huang, Qinyong Wang, Xingzhong Du, and Quoc Viet Hung Nguyen. 2018. Streaming Ranking Based Recommender Systems. In *SIGIR*. 525–534.
[48] Yuxiang Wang, Xiaoxuan Gou, Xiaoliang Xu, Yuxia Geng, Xiangyu Ke, Tianxing Wu, Zhiyuan Yu, Runhuai Chen, and Xiangying Wu. 2024. Scalable Community Search over Large-scale Graphs based on Graph Transformer. In *SIGIR*.
[49] Yuxiang Wang, Chengjie Gu, Xiaoliang Xu, Xinjun Zeng, Xiangyu Ke, and Tianxing Wu. 2024. Efficient and Effective (*k*,*P*)-Core-Based Community Search over Attributed Heterogeneous Information Networks. *Information Sciences* (2024), 120076.
[50] Yuxiang Wang, Jun Liu, Xiaoliang Xu, Xiangyu Ke, Tianxing Wu, and Xiaoxuan Gou. 2023. Efficient and Effective Academic Expert Finding on Heterogeneous Graphs through (k,P)-Core Based Embedding. *ACM Trans. Knowl. Discov. Data* 17, 6, Article 85 (mar 2023), 35 pages.
[51] Yuxiang Wang, Shuzhan Ye, Xiaoliang Xu, Yuxia Geng, Zhenghe Zhao, Xiangyu Ke, and Tianxing Wu. 2024. Scalable Community Search with Accuracy Guarantee on Attributed Graphs. In *ICDE*.
[52] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *PVLDB* 8, 7 (2015), 798–809.
[53] Xiaoliang Xu, Jun Liu, Yuxiang Wang, and Xiangyu Ke. 2022. Academic Expert Finding via (K,P)-Core based Embedding over Heterogeneous Graphs. In *ICDE*. 338–351.
[54] Zhaozhuo Xu, Weijie Zhao, Shulong Tan, Zhixin Zhou, and Ping Li. 2022. Proximity Graph Maintenance for Fast Online Nearest Neighbor Search. *arXiv* abs/2206.10839 (2022).
[55] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*. 745–754.
[56] Jaewon Yang and Jure Leskovec. 2015. Defining and Evaluating Network Communities based on Ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213.
[57] Yixing Yang, Yixiang Fang, Xuemin Lin, and Wenjie Zhang. 2020. Effective and Efficient Truss Computation over Large Heterogeneous Information Networks. In *ICDE*. 901–912.
[58] Junhao Ye, Yuanyuan Zhu, and Lu Chen. 2023. Top-r Keyword-Based Community Search in Attributed Graphs. In *ICDE*. 1652–1664.
[59] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2018. Index-Based Densest Clique Percolation Community Search in Networks. *IEEE Trans. Knowl. Data Eng.* 30, 5 (2018), 922–935.
[60] Fan Zhang, Conggai Li, Ying Zhang, Lu Qin, and Wenjie Zhang. 2020. Finding Critical Users in Social Communities: The Collapsed Core and Truss Problems. *IEEE Trans. Knowl. Data Eng.* 32, 1 (2020), 78–91.
[61] Zhiwei Zhang, Xin Huang, Jianliang Xu, Byron Choi, and Zechao Shang. 2019. Keyword-Centric Community Search. In *ICDE*. 422–433.
[62] Yingli Zhou, Yixiang Fang, Wensheng Luo, and Yunming Ye. 2023. Influential Community Search over Large Heterogeneous Information Networks. *PVLDB* 16, 8 (2023), 2047–2060.

A Integration with Existing Methods

We now take several representative CSAG problems, such as VAC [35], ATC [28], SEA [51], and rKACS [58], as examples to demonstrate how to build CHC-Index for them. After constructing the dedicated CHC-Index, we can apply the general pipeline of community search (§4) to find feasible communities using such an index.

A.1 Integration with VAC

Problem formalization. VAC aims to find a connected k-truss community C within a graph G that contains the query node q, with the goal of minimizing the attribute score of C, denoted by $\texttt{Ascore}(C)$. In VAC, $\texttt{Ascore}(C)$ is defined as the worst attribute distance $f(u, v)$ among any two nodes in a community C.

$$\texttt{Ascore}(C) = \max_{u,v \in V_C} f(u, v) \tag{6}$$

Here, $f(u, v)$ is defined as Euclidean distance or Jaccard distance for numerical or textual attributes, respectively. So, we formalize the problem as $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$, where C_{model}="`connected k-truss`", $F(f(), C)$ is actually the $\texttt{Ascore}(C)$ defined based on $f()$ over C, and the optimization goal as opt = min. Thus, VAC is expected to return a community C satisfying the following criteria.

$$C = \arg\min_{C \subseteq G} \texttt{Ascore}(C) \tag{7}$$

Building Υ. We next take C_{model} and $\eta() = f() + g()$ (Eq. 5) as input of Algorithm 1 to build the CHC-Index Υ using the method introduced in §3. Notably, each layer of sub-index Υ_k is built to maintain all nodes belonging to a k-truss community. Specifically, we begin by applying truss-decomposition method to get the largest k of the k-truss of G, denoted by $k_{\max}$, which indicates the highest layer of Υ. Then, we invoke Algorithm 2 to build each sub-index Υ_k for $\forall k \in [1, k_{\max}]$ layer by layer, followed by invoking Algorithm 8 to store each built Υ_k in the compressed adjacency list A_Υ.

Support VAC using Υ. VAC provides an exact solution S_e and an approximate solution S_a. The primary objective of them is to iteratively remove an edge e_{uv} between nodes u and v that have the largest $f(u, v)$, thereby reducing the overall $\texttt{Ascore}(C)$. The difference lies in their edge removal strategies: S_e iteratively removes the edge with the globally largest $f()$ (the largest $f()$ between any two nodes), while S_a iteratively removes the edge with the locally largest $f()$ (the largest $f()$ w.r.t. q). According to [35], both solutions tend to remove edges between nodes whose $f()$ deviates from the distribution of node attribute distances, making the attribute distance between nodes within the community as close as possible. This implies that the attribute distance from any node to q will also tend to be similar, which which is the intuition behind the approximate solution S_a. To support both two solutions, we first return a set of nodes similar to q from Υ, denoted by V^*. Then, we take the induced subgraph $G[V^*] \subseteq G$ as the input of S_e and S_a. If

S_a is used, it removes the edge with the globally largest $f()$. If S_a is used, it removes the edge with the locally largest $f()$. Since the size of $G[V^*]$ is smaller than the original G, efficient search is achieved.

A.2 Integration with SEA

Problem formalization. SEA [51] aims to find a connected k-core community C that contains the query node q, satisfying that the attribute distance of C, denoted by $\delta(C)$, is minimized. In SEA, $\delta(C)$ is defined as the average value of the attribute distance $f(u,q)$ between each node u in a community C and the query node q.

$$\delta(C) = \frac{\sum_{u \in V_C \setminus q} f(u,q)}{|V_C| - 1} \tag{8}$$

Here, $f(u,q)$ is defined as a composite distance of the Jaccard distance $f^t(u,q)$ regarding to textual attributes and the Manhattan distance $f^{\#}(u,q)$ regarding to numerical attributes, denoted by $f(u,q) = \gamma f^t(u,q) + (1-\gamma) f^{\#}(u,q)$. Therefore, we formalize the problem as $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$, where C_{model}="`connected k-core`", $F(f(), C)$ is $\delta(C)$ defined based on $f()$ over C, and the optimization goal as opt = min. Thus, SEA is expected to return a community C satisfying the following criteria.

$$C = \arg\min_{C \subseteq G} \delta(C) \tag{9}$$

Building Υ. Since the community is modeled as a connected k-core, we apply core-decomposition method in Algorithm 1 to get the largest k of the k-core of G, denoted by $k_{\max}$, as the highest layer of Υ. Then, we invoke Algorithm 2 (using $\eta() = g() + f()$) to build each sub-index Υ_k for $\forall k \in [1, k_{\max}]$ layer by layer, followed by invoking Algorithm 8 to store each built Υ_k in A_Υ.

Support SEA using Υ. SEA presents an approximate solution following a sampling-estimation manner. It first collect a set of nodes from the entire graph G that are similar to q in terms of $f()$ as the random sample. Then, it iteratively finds a connected k-core community C from the sample and estimate its $\delta(C)$ until the relative error of estimated $\delta(C)$ w.r.t. the exact one is small enough. Because SEA aims primarily to minimize the attribute distance between community nodes and the query node q, leveraging the index Υ to enhance efficiency is highly intuitive. This involves replacing the original graph G with an induced graph $G[V^*]$ constructed from nodes V^* returned by Υ that are similar to q, thereby serving as the overall population for the sampling-estimation process of SEA.

A.3 Integration with ATC

Problem formalization. ATC aims to find a connected k-truss community C that contains the query node q, satisfying that the the attribute score of C w.r.t. the given attributes W_q ($W_q \subseteq A(q)$ is a subset attributes of q), denoted by $\texttt{Ascore}(C, W_q)$, is maximized. In ATC, $\texttt{Ascore}(C, W_q)$ is defined based on the attribute coverage within C as follows, where V_w denotes the node set that includes all nodes containing an attribute w and $\frac{|V_w \cap V_C|}{|V_C|}$ is the attribute coverage of w within C, i.e., the fraction of nodes that contain w.

$$\texttt{Ascore}(C, W_q) = \sum_{w \in W_q} \frac{|V_w \cap V_C|^2}{|V_C|} \tag{10}$$

Intuitively, the more nodes within C share more attributes from W_q, the higher the $\texttt{Ascore}(C, W_q)$. This implicitly implies that $\texttt{Ascore}(C, W_q)$ is strongly dominated by the number of shared attributes between two nodes $u, v \in V_C$, based on which we define the attribute distance $f(u,v)$ as the Jaccard distance. Therefore, we formalize the problem as $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$, where C_{model}="`connected k-truss`", $F(f(), C)$ is $\texttt{Ascore}(C, W_q)$ defined based on $f()$ over C, and the optimization goal as opt = max. Thus, ATC returns a community C satisfying the following criteria.

$$C = \arg\max_{C \subseteq G} \texttt{Ascore}(C, W_q) \tag{11}$$

Building Υ. Since the community is modeled as a connected k-truss, we apply truss-decomposition method in Algorithm 1 to get the largest k of the k-core of G, denoted by $k_{\max}$. Then, we iteratively build each sub-index Υ_k and store it in A_Υ by invoking Algorithm 2 and Algorithm 8, respectively.

Support ATC using Υ. ATC presents a solution called `LocATC`, which adopts a bottom-up expansion strategy to find target communities with two steps. It first obtains candidate nodes based on a heuristic BFS method. Then, it iteratively removes nodes that can increase the overall $\texttt{Ascore}(C, W_q)$ until $\texttt{Ascore}(C, W_q)$ cannot further be optimized. When the quality of candidate nodes is low, continuous expansion of candidate nodes is necessary. The more expansion rounds, the longer the execution time. To support ATC, we replace LocATC's first step with returning candidate nodes V^* from the built CHC-Index Υ and take $G[V^*]$ as input of the second step, which is expected to enhance the quality of candidate nodes, thereby reducing the number of expansion rounds and significantly improving the overall efficiency.

A.4 Integration with rKACS

Problem formalization. rKACS aims to find top-r connected k-truss community $\{C_1, \cdots, C_r\}$ that each node in C_i contains at least one attribute in the query attributes W_q, satisfying that the semantic quality of C, denoted by $\texttt{score}(C, W_q)$, is maximized. In rKACS, $\texttt{score}(C, W_q)$ is defined as the worst score between any two nodes in a community C, denoted by $\texttt{score}(u, v, W_q)$.

$$\texttt{score}(C, W_q) = \min_{u,v \in V_C} \texttt{score}(u, v, W_q) \tag{12}$$

$$\texttt{score}(u, v, W_q) = (|A(u) \cap W_q| + |A(v) \cap W_q|) \times \frac{|A(u) \cap A(v)|}{|A(u) \cup A(v)|} \tag{13}$$

Since the attribute score between two nodes depends on their Jaccard similarity in attributes, we define the attribute distance $f(u,v)$ as the Jaccard distance (i.e., 1-Jaccard similarity). Therefore, we formalize the problem as $\langle C_{\text{model}}, q, k, \text{opt}\{F(f(), C)\}\rangle$, where C_{model}="`connected k-truss`", $F(f(), C)$ is $\texttt{score}(C, W_q)$ defined based on $f()$ over C, and the optimization goal as opt = max. Thus, rKACS returns r communities satisfying the following criteria.

$$\{C_1, \cdots, C_r\} = \arg\max_{C_1, \cdots, C_r \subseteq G} \sum \texttt{score}(C_i, W_q) \tag{14}$$

Building Υ. Since the community is modeled as a connected k-truss, we apply truss-decomposition in Algorithm 1 to get the largest layer of $k_{\max}$. Then, we iteratively build each sub-index Υ_k and store it in A_Υ by invoking Algorithm 2 and Algorithm 8, respectively.

Support rKACS using Υ. rKACS first removes nodes with pairwise scores $\texttt{score}(u, v, W_q) = 0$, resulting in a smaller graph G'. It then enumerates feasible communities on G', using score-based pruning to accelerate the search process. Since G' only excludes a small number of nodes with scores of 0, the remaining number of nodes is still large, leading to considerable time costs for subsequent enumeration and pruning. To support rKACS, we utilize the CHC-Index to retrieve a set of nodes similar to q, which not only excludes nodes with scores of 0 but also those with small non-zero scores. This significantly reduces the size of G', thereby decreasing the space required for feasible community enumeration and pruning, ultimately improving the efficiency of rKACS.