The VLDB Journal
https://doi.org/10.1007/s00778-023-00834-9

REGULAR PAPER

Identifying similar-bicliques in bipartite graphs

Kai Yao[1] · Lijun Chang[1] · Jeffrey Xu Yu[2]

Received: 2 June 2023 / Revised: 23 November 2023 / Accepted: 19 December 2023

Abstract

Bipartite graphs have been widely used to model the relationship between entities of different types, where vertices are partitioned into two disjoint sets/sides. Finding dense subgraphs in a bipartite graph is of great significance and encompasses many applications. However, none of the existing dense bipartite subgraph models consider similarity between vertices from the same side, and as a result, the identified results may include vertices that are not similar to each other. In this work, we formulate the notion of similar-biclique which is a special kind of biclique where all vertices from a designated side are similar to each other and aim to enumerate all similar-bicliques. The naive approach of first enumerating all maximal bicliques and then extracting all maximal similar-bicliques from them is inefficient, as enumerating maximal bicliques is already time consuming. We propose a backtracking algorithm MSBE to directly enumerate maximal similar-bicliques and power it by vertex reduction and optimization techniques. In addition, we design a novel index structure to speed up a time-critical operation of MSBE, as well as to speed up vertex reduction. Efficient index construction algorithms are developed. To handle dynamic graph updates, we also propose algorithms and optimization techniques for maintaining our index. Finally, we parallelize our index construction algorithms to exploit multiple CPU cores. Extensive experiments on 17 bipartite graphs as well as case studies are conducted to demonstrate the effectiveness and efficiency of our model and algorithms.

Keywords Maximal similar-biclique enumeration · Maximal biclique enumeration · Branch-reduce-and-bound · Structural similarity · Large bipartite graph

1 Introduction

Bipartite graphs have been widely used in real-world applications to model relationships between entities of different types, such as customer-product networks [50], author-paper networks [29] and user-event networks [13]. A bipartite graph is denoted by $G = (V_L, V_R, E)$, where the vertex set is partitioned into two disjoint subsets V_L and V_R which are referred to as the L-side and R-side vertices of the bipartite graph, respectively; each edge $e \in E$ can only connect vertices from different sides. Finding dense subgraphs in a bipartite graph is of great significance and encompasses many applications, such as community detection [1, 27], anomaly detection [16, 44], and group recommendation [37, 46].

One classic notion of dense bipartite subgraph is *biclique* [41], which requires every pair of vertices from different sides of the subgraph to be connected by an edge. For example, for the bipartite graph in Fig. 1 which represents researchers publishing in conference venues, the subgraph in the shadowed area is a biclique. In the literature, many algorithms have been proposed to enumerate all maximal bicliques [1, 3, 14, 31, 34, 49, 56] and to identify a biclique of the maximum size [37]. However, the biclique model has a fundamental limitation: vertices in a biclique are not necessarily similar to each other, despite that they share a set of common neighbors (i.e., vertices on the other side of the biclique). Consider the six researchers in the biclique in Fig. 1, all of them publish in database conferences. Besides, researchers r_1, r_2, r_3 also publish in machine learning (ML) conferences, while r_4, r_5, r_6 publish in high-performance computing (HPC) conferences. Thus, the two groups of researchers, $\{r_1, r_2, r_3\}$ and $\{r_4, r_5, r_6\}$, are likely from different backgrounds and communities, i.e., ML versus HPC.

✉ Kai Yao
kyao8420@uni.sydney.edu.au

Lijun Chang
Lijun.Chang@sydney.edu.au

Jeffrey Xu Yu
yu@se.cuhk.edu.hk

[1] The University of Sydney, Sydney, Australia

[2] The Chinese University of Hong Kong, Hong Kong, China

Published online: 25 January 2024

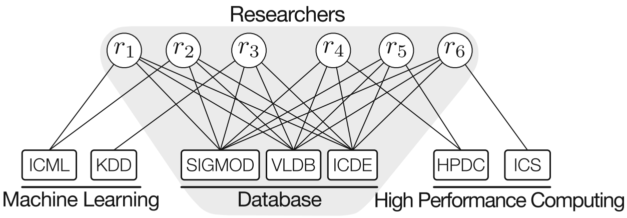


Fig. 1 Example of researcher-venue bipartite graph

Motivated by this, we formulate the notion of *similar-biclique* by requiring all vertices from one side of the biclique to be similar to each other. Our empirical studies show that similar-bicliques can be detected much more efficiently than bicliques. Identifying similar-bicliques is useful for the following applications.

- **Community detection.** Similar-biclique satisfies all the key requirements of community structure for bipartite graphs [27, 51, 57], and thus can be used to detect communities in interaction-type bipartite graphs such as user-rate-movie, customer-buy-product, and author-write-paper. Firstly, being a biclique, interactions between vertices from the two sides are intensive. Secondly, by enforcing the similarity constraint, users in a similar-biclique are similar to each other, e.g., having similar behaviors or interests.
- **Anomaly detection.** Similar-biclique can also be used for anomaly detection, which is a common task in e-commerce [4, 12, 37]. Here, the transactions of customers purchasing products form a customer-product bipartite graph. To improve the ranking of certain products, fraudsters may create fake accounts to purchase the products, i.e., click farming [10]. These fake accounts and the products they promote inevitably form a densely connected group, and meanwhile, these fake accounts will display a high level of synchronized behavior with each other [21]. Thus, suspicious groups (i.e., both the fraudulent accounts and the products they promote) can be captured by similar-bicliques.

Formally speaking, given a similarity threshold $0 < \varepsilon \leq 1$ and a size constraint $\tau \geq 0$, a vertex subset $C \subseteq V_L \cup V_R$ in a bipartite graph $G = (V_L, V_R, E)$ is a *similar-biclique* if (1) C is a biclique (i.e., $C_L \times C_R \subseteq E$), (2) all vertices of C_L are similar to each other (i.e., $\mathsf{sim}(u, v) \geq \varepsilon, \forall u, v \in C_L$), and (3) C satisfies the size constraint (i.e., $|C_L| \geq \tau$ and $|C_R| \geq \tau$). Here, C_L denotes $C \cap V_L$ and C_R denotes $C \cap V_R$; $\mathsf{sim}(u, v)$ measures the structural similarity between u and v, which is computed based on their neighbors $N(u)$ and $N(v)$ and will be formally defined in Sect. 3; the size constraint τ is introduced to avoid generating too small or too skewed results. Note that, we only apply the similarity constraint to one side of the vertices (either V_L or V_R as they are interchangeable), since in applications we are usually only interested in the similarity between "users." Nevertheless, the general technical ideas presented in this work can also be applied to the variant of similar-biclique that imposes the similarity constraint on both sides of vertices.

We in this work aim to enumerate all *maximal* similar-bicliques in a bipartite graph. We prove that this problem is #P-complete. As each (maximal) similar-biclique is contained in a maximal biclique, we could first enumerate all maximal bicliques, then extract maximal similar-bicliques from maximal bicliques, and finally eliminate all similar-bicliques that are either duplicates or not maximal. However, this approach is inefficient, as enumerating all maximal bicliques by the state-of-the-art algorithm ooMBEA [8] is still time consuming for large graphs. Thus, we propose the MSBE algorithm to directly enumerate maximal similar-bicliques, without first enumerating maximal bicliques.

MSBE follows the general backtracking framework of the Bron–Kerbosch algorithm [5] that enumerates all maximal *cliques* in a unipartite graph. Our observation is that once the set of L-side vertices C_L of a similar-biclique C is determined, its R-side vertices can be simply obtained as $C_R = \bigcap_{u \in C_L} N(u)$. Nevertheless, it is worth pointing out that we cannot ignore C_R during the enumeration process, since (1) the size of C_R will be used for pruning and (2) both C_L and C_R are needed for determining the maximality of the similar-biclique. MSBE iteratively builds up a partial solution $\langle C_L, C_R \rangle$, maintains a candidate set P_L that is used to grow C_L, and maintains an exclusive set Q_L that is used for checking the maximality of $\langle C_L, C_R \rangle$. In each recursion, a vertex $u \in P_L$ is added to C_L to grow the solution; after coming back from the recursion, u is moved from P_L to Q_L to avoid duplicates. To prune the search space, we propose the concept of *dominating*: $u \in P_L$ dominates $v \in P_L$ if $\mathsf{sim}(u, v) \geq \varepsilon$ and $N_{C_R}(u) \supseteq N_{C_R}(v)$, where $N_{C_R}(u) = N(u) \cap C_R$. We prove that if u dominates v, then we can prune the recursion of adding v to C_L when u is moved from P_L to Q_L. Furthermore, according to the definition, (1) each vertex u in a similar-biclique C must have at least τ neighbors in C (i.e., $|N_C(u)| \geq \tau$), and (2) each L-side vertex $u \in C_L$ must have at least $\tau - 1$ vertices that are similar to it; we call the vertices that are similar to u the *similar neighbors* of u, denoted $\Gamma(u)$. Thus, we propose to first prune all vertices that violate either of these two conditions, in a preprocess referred to as *vertex reduction*; our empirical studies show that a large portion of the input graph is pruned by vertex reduction.

We observe that a time-critical operation, in both vertex reduction and the recursion of backtracking, is computing $\Gamma(u)$ which would take $O(\sum_{v \in N(u)} |N(v)|)$ time, for a vertex $u \in V_L$. Note that, $\Gamma(u)$ is not stored in the graph representation, and it is also not affordable to store $\Gamma(u)$ (either in main memory or on disk) after it is computed as this would require a prohibitively large space. For example, it would take over 400GB on Bibsonomy, one of the graphs tested in our experiments. In view of this, we propose an offline-constructed index to speed up the computation of $\Gamma(u)$; note that, our index can be used to process maximal similar-biclique enumeration queries with different ε and τ values. This is based on the fact that for any similarity threshold ε, $\Gamma(u)$ is always a subset of $\Phi_u = \bigcup_{v \in N(u)} N(v)$, the 2-hop structural neighbors of u. Thus, we propose to first compute the similarity between u and every vertex of Φ_u in an offline process, and then compress them into a few segments which are stored in the index. Specifically, each segment is represented by $\mathsf{seg} = \langle \mathsf{v}_{\min}, \mathsf{v}_{\max}, \mathsf{s}_{\max}, c \rangle$ where $\mathsf{v}_{\min} \le \mathsf{v}_{\max}$ are two vertices of Φ_u, $\mathsf{s}_{\max}$ is the largest similarity between u and vertices of Φ_u that are in the range $[\mathsf{v}_{\min}, \mathsf{v}_{\max}]$, and c is the number of vertices of Φ_u that are in $[\mathsf{v}_{\min}, \mathsf{v}_{\max}]$; here, the comparison between vertices is based on their ids. To obtain $\Gamma(u)$, we go through each segment seg of Φ_u that satisfies $\mathsf{seg.s}_{\max} \ge \varepsilon$, and compute the similarity between u and each $v \in [\mathsf{seg.v}_{\min}, \mathsf{seg.v}_{\max}]$; note that, segments with $\mathsf{s}_{\max} < \varepsilon$ are skipped. Furthermore, we also use the index to speed up vertex reduction by first pruning vertices based on upper bounds of $\Gamma(u)$, which can be efficiently obtained based on the index without computing similarities.

Our main contributions are summarized as follows:

- We formulate the concept of similar-biclique, which can be used to detect interesting dense subgraphs from a bipartite graph. To the best of our knowledge, this is the first work investigating structural similarity between vertices in dense bipartite subgraph mining.
- We develop a backtracking algorithm MSBE to enumerate all maximal similar-bicliques in a bipartite graph. Vertex reduction and optimization techniques are proposed.
- We design a novel index structure to facilitate the computation of similar neighbors and propose a two-phase algorithm for efficient vertex reduction based on the index.
- We propose effective and efficient index construction algorithms by investigating two different strategies. We also parallelize our index construction algorithms by utilizing multiple CPU cores.
- We propose index maintenance algorithms to handle dynamic graph updates.
- Extensive empirical studies on 17 real bipartite graphs as well as case studies are conducted to demonstrate the efficiency of our algorithms and the effectiveness of our similar-biclique model. Our algorithm is up to 6 orders of magnitude faster than ooMBEA.

A preliminary version of this work has been published in [52]. Compared with [52], we in this work have proposed a more efficient algorithm TPA for index construction, parallelized our index construction algorithms, proposed index maintenance algorithms to handle dynamic graph updates, and conducted new experimental studies to evaluate the efficiency and effectiveness of our techniques.

Organization. Sect. 2 reviews the related works. Section 3 provides preliminaries including the definition of similar-biclique and the problem statement. Section 4 introduces a baseline algorithm and our MSBE algorithm. Section 5 presents our index structure, index-based algorithms, index construction algorithms, as well as parallelization techniques for index construction. Section 6 introduces our index maintenance algorithms. Section 7 reports experimental results. Section 8 concludes the work.

2 Related works

We categorize the related works as follows.

Maximal biclique enumeration. The problem of enumerating all maximal bicliques has been widely studied. The existing studies can be classified into two categories, depending on whether the input graph is bipartite or not. When the input graph is bipartite, the existing algorithms [1, 43, 56] generally enumerate subsets of vertices from one side, and then, the intersection of their neighbors forms the other side of the biclique. Besides, frequent item-set mining techniques have also been utilized to enumerate maximal bicliques [30, 31, 49, 55], as these two problems are highly related to each other. The state-of-the-art algorithm for maximal biclique enumeration over bipartite graphs is ooMBEA proposed in [8], which is compared in our experiments. There are also studies that aim at enumerating all maximal (non-induced) bicliques from a general graph, i.e., the input graph is not bipartite. For example, it is studied from a theoretical viewpoint in [14], consensus algorithms are proposed in [3], and a divide-and-conquer algorithm is proposed in [34]. However, these algorithms are generally slower than the algorithms that specifically handle bipartite graphs. Moreover, none of the existing studies on maximal biclique enumeration take into account the structural similarity between vertices.

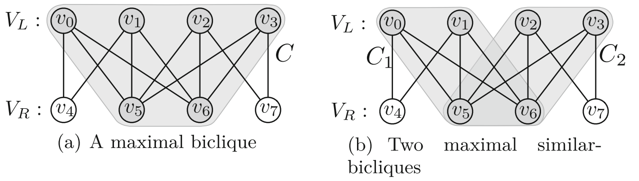
Maximal clique enumeration. The problem of enumerating all maximal cliques in a unipartite graph has also been extensively studied. The existing algorithms generally follow the backtracking framework of Bron and Kerbosch [5] with optimization techniques being proposed in [7, 9, 15, 38, 47]. However, these algorithms cannot handle bipartite graphs.

Dense bipartite subgraph identification. Other models have also been proposed for dense bipartite subgraph identification, such as quasi-biclique [35], k-biplex [54], (α, β)-core [24], k-bitruss [59], and k-wing [44]. Quasi-biclique and k-biplex relax the biclique model by allowing each vertex in one side of the result to miss a certain number of neighbors from the other side. On the other hand, (α, β)-core requires each vertex from one side to be connected to a certain number of vertices from the other side, and k-bitruss and k-wing require each edge to be involved in a certain number of $(2, 2)$-bicliques. None of these models consider similarity between vertices, and our case study in Sect. 7.3 demonstrates that similar-biclique outperforms k-biplex and (α, β)-core in anomaly detection. Note that it is also possible to integrate similarity constraint into these dense bipartite subgraph models, in a similar way to similar-bicliques. However, this would require thorough studies, from problem hardness analysis to algorithm design, for each model. For example, although (α, β)-core can be computed in polynomial time, computing *similar*-(α, β)-cores may require an exponential time as the number of maximal *similar*-(α, β)-cores could be exponential. Thus, we leave these to our future studies.

Structural vertex similarity. Structural vertex similarity refers to similarity measures between pairs of vertices that are computed based solely on the topology of the graph [28]. They are usually categorized as *local-topology-based* and *global-topology-based* similarity measures. For the former, the similarity between two vertices is computed based on their neighbors, i.e., locally. Examples include Jaccard similarity [19], cosine similarity [42], Dice's coefficient [11], hub depressed/promoted index [36], and Adamic–Adar index [2]. For the latter, the global structure information is utilized to derive the similarity between two vertices. Examples include Katz [22], SimRank [20], C-Rank [53], P-Rank [58], and MatchSim [33]. As global-topology-based measures usually need to access the entire graph to compute the similarity, they are computationally more expensive than the local-topology-based measures. Thus, we adopt local-topology-based measures in this work.

3 Preliminary

We consider an unweighted and undirected bipartite graph $G = (V_L, V_R, E)$, where V_L and V_R denote the two disjoint vertex sets (i.e., L-side vertices and R-side vertices) and $E \subseteq V_L \times V_R$ denotes the edge set. Without loss of generality, we assume that vertices of V_L take (integer) ids from $\{1, 2, \dots, |V_L|\}$, and vertices of V_R take ids from $\{1+|V_L|, 2+|V_L|, \dots, |V_R|+|V_L|\}$. For any vertex $v \in V_L$ (resp. $v \in V_R$), we say it is an L-side vertex (resp. R-side vertex). For any vertex subset $C \subseteq V_L \cup V_R$, we use C_L and C_R to denote the subsets of vertices of C that are from V_L and V_R, respectively, i.e., $C_L = C \cap V_L$ and $C_R = C \cap V_R$. We call the set of neighbors of u in G the *structural neighbors* of u, denoted $N_G(u) = \{v \mid (u, v) \in E\}$, and denote the *structural degree* of u by $d_G(u) = |N_G(u)|$. Besides structural neighbor and structural degree, we will also define similar neighbor and similar degree based on structural similarity.



(a) A maximal biclique

(b) Two maximal similar-bicliques

Fig. 2 Maximal biclique versus maximal similar-biclique

Definition 1 (*Structural Similarity*) Given two vertices u and v in G, their structural similarity is defined as $\mathsf{sim}(u, v) = \frac{|N_G(u) \cap N_G(v)|}{|N_G(u) \cup N_G(v)|}$.

The structural similarity $\mathsf{sim}(u, v)$ is between 0 and 1. It measures the Jaccard similarity between the set of structural neighbors of u and that of v. Given a similarity threshold $\varepsilon > 0$, we say u and v are similar if $\mathsf{sim}(u, v) \geq \varepsilon$. The set of vertices that are similar to u is called the *similar neighbors* of u, denoted $\Gamma_{G,\varepsilon}(u)$, i.e., $\Gamma_{G,\varepsilon}(u) = \{v \in V_L \cup V_R \mid \mathsf{sim}(u, v) \geq \varepsilon\}$. Accordingly, denote $\delta_{G,\varepsilon}(u) = |\Gamma_{G,\varepsilon}(u)|$ the *similar degree* of u. Note that as the structural similarity between vertices from different sides is always 0, similar neighbors of vertex u only contain vertices from the same side as u. For presentation simplicity, we call structural similarity simply as *similarity* and omit the subscripts G and/or ε from the notations.

Definition 2 (*Similar-Biclique*) Given a bipartite graph $G = (V_L, V_R, E)$ and a similarity threshold $\varepsilon > 0$, a vertex subset $C \subseteq V_L \cup V_R$ is a similar-biclique if

- C is a *biclique*, i.e., $C_L \times C_R \subseteq E$, and
- all vertices from the L-side are similar to each other, i.e., $\mathsf{sim}(u, v) \geq \varepsilon, \forall u, v \in C_L$.

We also denote C as $\langle C_L, C_R \rangle$. A similar-biclique is *maximal* if it is not a subset of any larger similar-biclique.

Note that for presentation simplicity, the similarity constraint is assumed to be considered for the L-side vertices. To apply the similarity constraint for R-side vertices in applications, we can simply swap the roles of V_L and V_R in G.

Example 1 Figure 2a shows a bipartite graph G with $V_L = \{v_0, \dots, v_3\}$ and $V_R = \{v_4, \dots, v_7\}$, in which the subgraph C in the gray area is a maximal biclique. However, v_1 and v_2 in the L-side of C are not similar to each other for $\varepsilon = 0.6$, i.e., $\mathsf{sim}(v_1, v_2) = 0.5$. Subgraphs C_1 and C_2 as shown in Fig. 2b are two maximal similar-bicliques, in each of which all vertices on the L-side are similar to each other.

Problem 1 (*Maximal Similar-Biclique Enumeration*) Given a bipartite graph $G = (V_L, V_R, E)$, a similarity threshold $\varepsilon > 0$, and a size constraint $\tau > 0$, we study the problem of enumerating all maximal similar-bicliques C in G that satisfy the size constraint τ (i.e., $|C_L| \geq \tau$ and $|C_R| \geq \tau$).

The size constraint τ is adopted here to avoid generating too small or too skewed similar-bicliques (i.e., with very few or no vertices in one side). For presentation simplicity, we assume the same size constraint τ for both sides. Note that the techniques that we are going to present in this work can be straightforwardly extended to handle different size constraints on the two sides.

Theorem 1 *The problem of enumerating all maximal similar-bicliques is #P-complete.*

Proof The #P-completeness of our problem directly follows from the facts that (1) the problem of enumerating all maximal bicliques is #P-complete [23, 26], and (2) it is a special case of our problem, i.e., by setting $\varepsilon = \frac{1}{|V_R|}$. Note that, for this small ε, two vertices of V_L are similar if and only if they have at least one common neighbors in V_R. Thus, every maximal similar-biclique is also a maximal biclique, and vice versa. □

Remark about Structural Similarity. In Definition 1, we use the Jaccard similarity to measure the structural similarity since it has been widely used and shown great success in many graph analysis tasks, such as structural graph clustering [6, 48], link prediction [32, 39], and local graph sparsification [45]. Nevertheless, other local-topology-based similarity measures reviewed in Sect. 2, such as cosine similarity $\frac{|N_G(u) \cap N_G(v)|}{\sqrt{d_G(u) \times d_G(v)}}$ and hub promoted index $\frac{|N_G(u) \cap N_G(v)|}{\min\{d_G(u), d_G(v)\}}$, can be easily plugged into our model and algorithms. We will point out the changes that need to be made to adopt these measures, when presenting our algorithms.

4 Our algorithms

In this section, we propose an MSBE algorithm to enumerate all maximal similar-bicliques. Before that, we first in Sect. 4.1 present a baseline algorithm based on the existing maximal biclique enumeration algorithms.

4.1 A baseline algorithm

It is easy to observe that maximal similar-bicliques must be contained in maximal bicliques, since every similar-biclique is also a biclique. Thus, a naive approach is first enumerating all maximal bicliques by invoking one of the existing maximal biclique enumeration algorithms, and then post-processing the detected maximal bicliques to obtain maximal similar-bicliques. Specifically, for each maximal biclique, we extract maximal similar-bicliques by imposing the similarity constraint on L-side vertices, and then eliminate all similar-bicliques that are either duplicates or non-maximal. We omit the details, since our empirical study in Sect. 7 shows that enumerating all maximal bicliques by the state-of-the-art algorithm ooMBEA [8] is already time consuming for large graphs.

Algorithm 1: MSBE$(G = (V_L, V_R, E), \varepsilon, \tau)$

```
1  for each $u \in V_L \cup V_R$ do del$(u) \leftarrow$ false
   VReduce$(G, \varepsilon, \tau, $del$(\cdot))$;
2  for each $u \in V_L$ s.t. del$(u) =$ false do
3      $C_L \leftarrow \{u\}$; $C_R \leftarrow \{v \in N(u) \mid $del$(v) = $false$\}$;
4      $P_L \leftarrow \emptyset$; $Q_L \leftarrow \emptyset$;
5      Obtain $\Gamma(u)$;
6      for each $v \in \Gamma(u)$ do
7          if $v > u$ then $P_L \leftarrow P_L \cup \{v\}$ else $Q_L \leftarrow Q_L \cup \{v\}$
8      Enum$(C_L, C_R, P_L, Q_L)$;

   Procedure Enum$(C_L, C_R, P_L, Q_L)$
9  if $\nexists u \in P_L \cup Q_L$ s.t. $N(u) \supseteq C_R$ then
10     if $|C_L| \geq \tau$ and $|C_R| \geq \tau$ then output $\langle C_L, C_R \rangle$
11 for each $u \in P_L$ do
12     $C'_L \leftarrow C_L \cup \{u\}$; $C'_R \leftarrow C_R \cap N(u)$;
13     Obtain $\Gamma(u)$;
14     $P'_L \leftarrow P_L \cap \Gamma(u)$; $Q'_L \leftarrow Q_L \cap \Gamma(u)$;
15     if $|C'_L| + |P'_L| \geq \tau$ and $|C'_R| \geq \tau$ then
       Enum$(C'_L, C'_R, P'_L, Q'_L)$  $P_L \leftarrow P_L \setminus \{u\}$; $Q_L \leftarrow Q_L \cup \{u\}$;
```

4.2 Our MSBE algorithm

According to the definition of similar-biclique, if we build a similarity graph G_s for V_L where two vertices of V_L are connected by an edge if their similarity is at least ε, then for every (maximal) similar-biclique C, its L-side vertices C_L form a clique in G_s. Moreover, once the L-side vertices C_L of a maximal similar-biclique are determined, the R-side vertices C_R can be easily obtained as the set of common neighbors of C_L, i.e., $C_R = \bigcap_{u \in C_L} N(u)$. In view of this, we propose to adopt the general backtracking framework of the Bron–Kerbosch algorithm [5] to enumerate all maximal similar-bicliques. However, there are two issues. (1) The similarity graph G_s is not available in the input. (2) The set of L-side vertices C_L of a maximal similar-biclique C is not necessarily a *maximal* clique in G_s, though C_L is a clique in G_s. This is because, a too large C_L may result in a too small C_R, violating the size constraint τ on C_R.

We propose techniques to address the above issues, and the pseudocode of our algorithm is shown in Algorithm 1, denoted MSBE. We first prune unpromising vertices by invoking VReduce (Lines 1–2), which will be introduced shortly in Algorithm 2; a vertex u is pruned if del$(u) =$ **true**. For each remaining vertex $u \in V_L$ with del$(u) =$ **false**, we enumerate all maximal similar-bicliques containing u (Lines 3–10). To do so, we iteratively grow a partial solution $\langle C_L, C_R \rangle$ where C_L is initialized as $\{u\}$ (Line 4). Besides C_L and C_R, we also maintain P_L and Q_L in a similar fashion to the Bron–Kerbosch algorithm [5], such that each vertex of $P_L \cup Q_L$ is similar to all vertices of C_L. Specifically, P_L is a set of

Algorithm 2: VReduce$(G, \varepsilon, \tau, \mathsf{del}(\cdot))$

```
1 for each $u \in V_L \cup V_R$ do $d(u) \leftarrow |N(u)|$ for each $u \in V_L$ do
  Obtain $\Gamma(u)$ and set $\delta(u) \leftarrow |\Gamma(u)|$ while $(\exists u \in V_L \cup V_R$ s.t.
  $\mathsf{del}(u) =$ false and $d(u) < \tau)$ or $(\exists u \in V_L$ s.t. $\mathsf{del}(u) =$ false
  and $\delta(u) < \tau - 1)$ do
2     for each $v \in N(u)$ do $d(v) \leftarrow d(v) - 1$ if $u \in V_L$ then
3         Obtain $\Gamma(u)$;
4         for each $v \in \Gamma(u)$ do $\delta(v) \leftarrow \delta(v) - 1$
5     $\mathsf{del}(u) \leftarrow$ true;
```

Algorithm 3: SimNei$(G, u, \varepsilon, \mathsf{del}(\cdot))$

```
  Output: $\Gamma(u)$
1 $\Gamma(u) \leftarrow \emptyset$;
2 for each $v \in V_L$ do $c(v) \leftarrow 0$ for each $v \in N(u)$ do
3     for each $w \in N(v)$ and $w \neq u$ do $c(w) \leftarrow c(w) + 1$
4 for each $v \in V_L$ s.t. $c(v) \neq 0$ and $\mathsf{del}(v) =$ false do
5     if $\frac{c(v)}{d(u)+d(v)-c(v)} \geq \varepsilon$ then $\Gamma(u) \leftarrow \Gamma(u) \cup \{v\}$
6 return $\Gamma(u)$
```

candidate vertices that are used to grow C_L, and Q_L is a set of previously considered candidate vertices that are used for checking the maximality of $\langle C_L, C_R \rangle$. P_L and Q_L are initialized at Lines 5–9, after which we invoke the procedure Enum for enumeration (Line 10); to avoid duplicates, the similar neighbors of u with larger vertex id than u are put in P_L, and those with smaller vertex id are put in Q_L.

In Enum, if the current similar-biclique $\langle C_L, C_R \rangle$ is maximal, we report it (Lines 11–12); note that $\langle C_L, C_R \rangle$ is maximal if and only if there is no vertex $u \in P_L \cup Q_L$ such that $N(u) \supseteq C_R$. Next, Enum iteratively adds a vertex of P_L to C_L, updates the corresponding C_R, P_L and Q_L, and recursively invokes itself to enumerate more similar-bicliques (Lines 14–17). After processing $u \in P_L$, we remove u from P_L and add it to Q_L (Line 18).

Vertex Reduction. As a similar-biclique needs to have at least τ vertices on each side, each vertex u in a similar-biclique C must have at least τ structural neighbors in C (i.e., $|N_C(u)| \geq \tau$). Furthermore, as all L-side vertices in a similar-biclique C are similar to each other, each L-side vertex u must have at least $\tau - 1$ similar neighbors in C (i.e., $|\Gamma_C(u)| \geq \tau - 1$). As a result, we can exclude all vertices that violate either of these two conditions from being considered in the enumeration procedure Enum, i.e., mark them as deleted; we call this process as *vertex reduction.*

We propose an algorithm VReduce to conduct vertex reduction, whose pseudocode is shown in Algorithm 2. Firstly, we obtain the structural degree $d(u)$ for each vertex $u \in V_L \cup V_R$ (Line 1), and obtain the similar degree $\delta(u)$ for each L-side vertex $u \in V_L$ (Line 2). Then, as long as there is a non-deleted vertex u with $d(u) < \tau$ or a non-deleted L-side vertex u with $\delta(u) < \tau - 1$, we mark u as deleted (Line 8), decrease the structure degree of u's structural neighbors by 1 (Line 4), and also decrease the similar degree of u's similar neighbors by 1 if u is an L-side vertex (Lines 5–7).

Compute Similar Neighbors $\Gamma(u)$. One fundamental operation in both Algorithm 1 and Algorithm 2 is computing $\Gamma(u)$ for an L-side vertex u; note that $\Gamma(u)$ is not stored with the graph G. A straightforward method to collect $\Gamma(u)$ is computing $\mathsf{sim}(u, v)$ for each vertex $v \in V_L$. The time complexity would be $O(|E|)$, as it needs to visit every edge of G. This is inefficient, by noting that Algorithm 1 and Algorithm 2 need to compute $\Gamma(u)$ for many vertices u and multiple times.

We propose a more efficient algorithm in Algorithm 3, denoted SimNei. Instead of blindly computing $\mathsf{sim}(u, v)$ for each $v \in V_L$, we only compute $\mathsf{sim}(u, v)$ for those v with $\mathsf{sim}(u, v) > 0$. Our main idea is to first compute the number of common neighbors $c(v)$ between u and v for each 2-hop structural neighbor v of u (Lines 3–4). Then, $\mathsf{sim}(u, v)$ can be calculated as $\frac{c(v)}{d(u)+d(v)-c(v)}$ (Line 6).[1] Note that, in our implementation, to make the time complexity of SimNei to be independent of $|V_L|$ which may be large, we only initialize $c(\cdot)$ once at the beginning of the entire algorithm execution (e.g., in MSBE), and after using $c(\cdot)$ at Line 4–6 of Algorithm 3 we reset those updated $c(\cdot)$ to be 0. In addition, we also collect at Line 4 the set of vertices with non-zero $c(\cdot)$ into a set S, such that Line 5 as well as the resetting of $c(\cdot)$ can be conducted in $O(|S|)$ time. As a result, the time complexity of SimNei is $O(\sum_{v \in N(u)} d(v))$, which is lower than $O(|E|)$.

Optimizations for Enum. We further propose two optimization techniques to speed up the Enum procedure. Recall that, an instance of Enum is represented by (C_L, C_R, P_L, Q_L) where $C_R = \bigcap_{u \in C_L} N(u)$ and $|C_R| \geq \tau$,[2] and aims to enumerate all maximal similar-bicliques C^* satisfying $C_L \subset C_L^* \subseteq C_L \cup P_L$. Firstly, an enumeration instance can be terminated once we know that it will not generate any maximal similar-biclique. That is, by including any subset of vertices of P_L into C_L, the resulting similar-biclique would not be maximal. This is formulated in the lemma below.

Lemma 1 (Early Termination) *If there exists a vertex $u \in Q_L$ such that u is similar to all vertices of P_L and $N(u) \supseteq C_R$, then there is no maximal similar-biclique C^* with $C_L \subset C_L^* \subseteq C_L \cup P_L$ and thus we can terminate this enumeration instance.*

Proof Suppose there is such a maximal similar-biclique C^* with $C_L \subset C_L^* \subseteq C_L \cup P_L$. Then, we must have $C_R^* \subseteq C_R$ and thus $N(u) \supseteq C_R \supseteq C_R^*$. Since $u \in Q_L$ is similar to all

[1] The formula should be changed to $\frac{c(v)}{\sqrt{d(u) \times d(v)}}$ for cosine similarity, and to $\frac{c(v)}{\min\{d(u), d(v)\}}$ for hub promoted index.

[2] To be more precise, we should exclude from C_R all vertices that are marked as deleted.

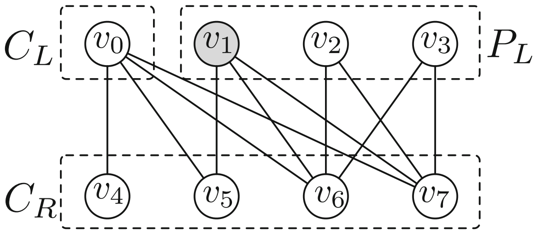


Fig. 3 Example of domination

vertices of P_L (and also note that all vertices of Q_L, including u, are similar to all vertices of C_L according to the construction of Q_L), u is similar to all vertices of C_L^*. Consequently, $C^* \cup \{u\}$ is also a similar-biclique, contradicting that C^* is a maximal similar-biclique. □

Secondly, we can reduce the number of instances generated at Line 17 of Algorithm 1, based on the concept of dominating set.

Definition 3 (*Dominating Set*) Given an instance (C_L, C_R, P_L, Q_L) of Enum, for two distinct vertices $u, v \in P_L \cup Q_L$, we say that u dominates v if $\mathsf{sim}(u, v) \geq \varepsilon$ and $N_{C_R}(u) \supseteq N_{C_R}(v)$, where $N_{C_R}(u) = N(u) \cap C_R$. The dominating set of u, denoted $\mathsf{DomSet}(u)$, is the subset of vertices of P_L that are dominated by u, i.e., $\mathsf{DomSet}(u) = \{v \in P_L \mid \mathsf{sim}(u, v) \geq \varepsilon \wedge N_{C_R}(u) \supseteq N_{C_R}(v)\}$.

Note that, a vertex does not dominate itself.

Lemma 2 *Given an instance (C_L, C_R, P_L, Q_L) of* Enum *and a vertex $u^* \in P_L \cup Q_L$, any maximal similar-biclique C^* with $C_L \subset C_L^* \subseteq C_L \cup P_L$ must contain a vertex of $P_L \backslash \mathsf{DomSet}(u^*)$.*

Proof Suppose there is such a maximal similar-biclique C^* with $C_L \subset C_L^* \subseteq C_L \cup P_L$ that contains no vertex of $P_L \backslash \mathsf{DomSet}(u^*)$. That is, $C_L^* \backslash C_L \subseteq \mathsf{DomSet}(u^*)$. Then, it is easy to verify that $C^* \cup \{u^*\}$ is also a similar-biclique, contradicting the maximality of C^*. □

Example 2 Consider the instance in Fig. 3 where $Q_L = \emptyset$. For $\varepsilon = 0.6$, $v_1 \in P_L$ is similar to both v_2 and v_3. Moreover, we can see that $N_{C_R}(v_1) \supseteq N_{C_R}(v_2)$ and $N_{C_R}(v_1) \supseteq N_{C_R}(v_3)$. Thus, $\mathsf{DomSet}(v_1) = \{v_2, v_3\}$, and we know that every maximal similar-biclique C^* with $C_L \subset C_L^* \subseteq C_L \cup P_L$ must contain v_1 since $P_L \backslash \mathsf{DomSet}(v_1) = \{v_1\}$.

To apply Lemma 2, we revise Line 13 of Algorithm 1 as follows: we first choose a vertex u^* from $P_L \cup Q_L$ before the "for loop," and then replace "$u \in P_L$" with "$u \in P_L \backslash \mathsf{DomSet}(u^*)$" in the "for loop" statement. This means that we do not generate enumeration instances, at Line 17 of Algorithm 1, for vertices $u \in \mathsf{DomSet}(u^*)$. To maximize the benefit of Lemma 2, u^* is chosen as the one that minimizes $|P_L \backslash \mathsf{DomSet}(u^*)|$ among all vertices of $P_L \cup Q_L$. If u^* dominates a substantial number of vertices in P_L (i.e., when the cardinality of $\mathsf{DomSet}(u^*)$ is large), applying Lemma 2 can significantly reduce the overall running time.

Theorem 2 *The time complexity of Algorithm 1 is $\mathcal{O}(|V_L| \cdot |E| \cdot 2^{|V_L|})$.*

Proof We first prove that the time complexity of VReduce (i.e., Algorithm 2) is $\mathcal{O}(\sum_{v \in V_R}(d(v))^2)$. In Algorithm 2, Line 1 runs in $\mathcal{O}(|E|)$ time. Line 2 runs in $\mathcal{O}(\sum_{v \in V_R}(d(v))^2)$ time, since computing $\Gamma(u)$ by Algorithm 3 takes time $\mathcal{O}(\sum_{v \in N(u)} d(v))$ as discussed above, by noting that

$$\sum_{u \in V_L} \sum_{v \in N(u)} d(v) = \sum_{(u,v) \in E} d(v)$$
$$= \sum_{v \in V_R} \sum_{u \in N(v)} d(v) = \sum_{v \in V_R} (d(v))^2$$

Since each vertex $u \in V_L \cup V_R$ is removed at most once at Lines 4–8, the total cost of running Line 4 for all deleted vertices is $\mathcal{O}(|E|)$, and the total cost of running Lines 6–7 for all deleted vertices is $\mathcal{O}(\sum_{v \in V_R}(d(v))^2)$, the same as that of Line 2. In addition, we use a queue to store the vertices that should be deleted (i.e., satisfying the conditions at Line 3) such that finding a vertex at Line 3 takes constant time. Thus, the time complexity of VReduce is $\mathcal{O}(\sum_{v \in V_R}(d(v))^2)$.

Now, we prove that the time complexity of MSBE (Algorithm 1) is $\mathcal{O}(|V_L| \cdot |E| \cdot 2^{|V_L|})$. Firstly, it is easy to see that the time complexity of Algorithm 1 excluding Line 10 (i.e., the recursion) is $\mathcal{O}(\sum_{v \in V_R}(d(v))^2)$, by following the analysis of the time complexity of VReduce. Secondly, invoking Enum with input (C_L, C_R, P_L, Q_L) takes time $\mathcal{O}(|E| \cdot 2^{|P_L|})$, since the recursion builds a complete binary search tree with each instance (C_L, C_R, P_L, Q_L) has two children: one including u into C_L, one excluding u from C_L. The time for generating the first child (Lines 14–17) is $\mathcal{O}(\sum_{v \in N(u)} d(v)) \subseteq \mathcal{O}(|E|)$, and the time for generating the second child is $\mathcal{O}(1)$ (Line 18). In addition, the total number of leaf instances is 2^{P_L}. Thus, each invocation to Enum at Line 10 of Algorithm 1 takes time $\mathcal{O}(|E| \cdot 2^{|P_L|}) \subseteq \mathcal{O}(|E| \cdot 2^{|V_L|})$, and the total time complexity follows. □

Discussions. MSBE is different from both maximal clique enumeration algorithms for unipartite graphs and maximal biclique enumeration algorithms for bipartite graphs, as follows. Firstly, MSBE needs to compute the similar neighbors for L-side vertices, which are not required by any of the existing algorithms. Secondly, compared with maximal clique enumeration algorithms, MSBE needs to consider common structural neighbors C_R of C_L, in addition to common similar neighbors. Thirdly, our optimization techniques for Enum are also different from the existing ones.

In MSBE, we need to obtain the similar neighbors $\Gamma(\cdot)$ of an L-side vertex multiple times, e.g., at Lines 6 and 15 of Algorithm 1 and Lines 2 and 6 of Algorithm 2. We can either invoke SimNei to compute $\Gamma(u)$ every time when it is needed or store $\Gamma(u)$ in main memory after it is computed for the first time and then directly retrieve it for all subsequent requests. We use MSBE to denote the algorithm that uses the first strategy, and mat-MSBE the algorithm that uses the second strategy. (Here, mat stands for materialization.)

5 Speeding up similar neighbor computation and vertex reduction

MSBE has the disadvantage of repeatedly computing the similar neighbors from scratch which is time consuming, while mat-MSBE may demand an extremely large main memory to store the similar neighbors. For example, it would take more than 400GB main memory for the graph bibsonomy used in our experiments even for a moderate $\varepsilon = 0.5$. In this section, we propose an offline-constructed index to speed up the computation of $\Gamma(u)$ as well as vertex reduction. We give an overview of the index structure in Sect. 5.1, present our index-based algorithms in Sect. 5.2, discuss index construction in Sect. 5.3, and finally parallelize our index construction algorithm in Sect. 5.4.

5.1 Overview of index structure

Let Φ_u be the set of 2-hop structural neighbors of u, i.e., $\Phi_u = \bigcup_{v \in N(u)} N(v)$. Firstly, we have the following lemma.

Lemma 3 *For any similarity threshold $\varepsilon > 0$, the set of similar neighbors of u is a subset of Φ_u, i.e., $\Gamma_\varepsilon(u) \subseteq \Phi_u$.*

Proof The correctness of the lemma directly follows from the fact that any vertex $v \notin \Phi_u \cup \{u\}$ has no common neighbor with u and thus $\mathsf{sim}(u, v) = 0$. □

Based on Lemma 3, one possible indexing strategy is precomputing and storing $\mathsf{sim}(u, v)$ for each $u \in V_L \cup V_R$ and each $v \in \Phi_u$.[3] However, the space complexity of this strategy would be $O(|V_L|^2 + |V_R|^2)$, which is prohibitively high even for moderate-sized graphs since the space requirement is essentially the same as the case of mat-MSBE when ε is very small. For example, even for a moderate-sized graph with 10^6 vertices, the storage space would be over 2TB.

Instead of storing Φ_u and the structural similarities in their raw format, we summarize them into a few segments.

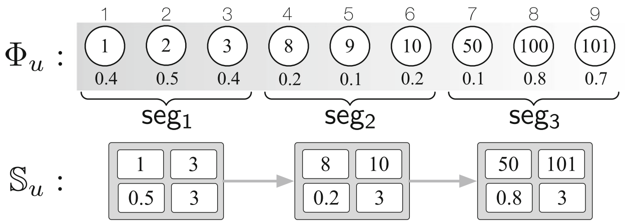


Fig. 4 Overview of index structure

Definition 4 (*Segment*) A segment, denoted seg, of Φ_u is a four-tuple $\langle \mathsf{v}_{\min}, \mathsf{v}_{\max}, \mathsf{s}_{\max}, c \rangle$, where $\mathsf{v}_{\min} \le \mathsf{v}_{\max}$ are two vertices of Φ_u,

$$\mathsf{s}_{\max} = \max_{v \in \Phi_u : \mathsf{v}_{\min} \le v \le \mathsf{v}_{\max}} \mathsf{sim}(u, v)$$

and $c = |\{v \in \Phi_u \mid \mathsf{v}_{\min} \le v \le \mathsf{v}_{\max}\}|$. Here, vertex comparison is based on vertex id.

Given a segment $\mathsf{seg} = \langle \mathsf{v}_{\min}, \mathsf{v}_{\max}, \mathsf{s}_{\max}, c \rangle$ of Φ_u, we use $V(\mathsf{seg})$ to denote $\{v \in \Phi_u \mid \mathsf{v}_{\min} \le v \le \mathsf{v}_{\max}\}$. It is immediate from the definition that $c = |V(\mathsf{seg})|$ and

- $\mathsf{v}_{\min}$ (resp. $\mathsf{v}_{\max}$) is the smallest (resp. largest) vertex id in $V(\mathsf{seg})$;
- $\mathsf{s}_{\max}$ is the largest similarity between u and a vertex of $V(\mathsf{seg})$, and thus $\mathsf{s}_{\max}$ provides an upper bound of $\mathsf{sim}(u, v)$ for all $v \in V(\mathsf{seg})$.

Thus, we say that seg *covers* vertices $V(\mathsf{seg})$. A set of segments $\mathbb{S}_u = \{\mathsf{seg}_1, \ldots, \mathsf{seg}_k\}$ covers Φ_u if

$$\bigcup_{\mathsf{seg} \in \mathbb{S}_u} V(\mathsf{seg}) = \Phi_u$$

In this work, we only consider disjoint segments, i.e., $V(\mathsf{seg}_i) \cap V(\mathsf{seg}_j) = \emptyset$ for $i \ne j$. Our index structure, denoted $\mathcal{I}$, covers Φ_u by a set of segments, for all $u \in V_L \cup V_R$. That is, $\mathcal{I}$ consists of $\mathbb{S}_u$ such that $\mathbb{S}_u$ covers Φ_u, for all $u \in V_L \cup V_R$.

Example 3 Figure 4 shows the 2-hop structural neighbors Φ_u of u, which are sorted in increasing order regarding vertex id. The decimal below each vertex is the similarity w.r.t. u. Φ_u is covered by three segments $\mathsf{seg}_1, \mathsf{seg}_2, \mathsf{seg}_3$. Take seg_1 as an example, the two numbers in the first row (i.e., 1 and 3) represent $\mathsf{v}_{\min}$ and $\mathsf{v}_{\max}$, and the two numbers in the second row (i.e., 0.5 and 3) represent $\mathsf{s}_{\max}$ and c.

5.2 Index-based algorithms

In this subsection, we present index-based algorithms for similar neighbor computation and for vertex reduction.

[3] Note that, we also need to index Φ_u for $u \in V_R$, since in practice the similarity constraint can be put on either L-side or R-side vertices.

Algorithm 4: indexedSN$(u, \varepsilon, G, \mathcal{I}, \mathsf{del}(\cdot))$

```
1 $\Gamma(u) \leftarrow \emptyset$;
2 for each seg $\in \mathbb{S}_u$ s.t. seg.$\mathsf{s}_{\max} \geq \varepsilon$ do
3     for each $v \in [$seg.$\mathsf{v}_{\min}$, seg.$\mathsf{v}_{\max}]$ do
4         if del$(v)$ = false and $v \neq u$ then
5             if ub$(u, v) \geq \varepsilon$ and sim$(u, v) \geq \varepsilon$ then
6                 $\Gamma(u) \leftarrow \Gamma(u) \cup \{v\}$;
7 return $\Gamma(u)$
```

Index-based Similar Neighbor Computation. The pseudocode of using the index $\mathcal{I}$ to efficiently obtain the similar neighbors $\Gamma(u)$ for a vertex u is shown in Algorithm 4, denoted indexedSN. We go through each segment seg $\in \mathbb{S}_u$ with seg.$\mathsf{s}_{\max} \geq \varepsilon$ (Line 2), and compute sim(u, v) for each $v \in [$seg.$\mathsf{v}_{\min}$, seg.$\mathsf{v}_{\max}]$ (Line 3); recall that (1) seg.$\mathsf{s}_{\max}$ upper bounds sim(u, v) for each $v \in V($seg$)$, and (2) $V($seg$)$ is not stored in the index structure $\mathcal{I}$. As computing sim(u, v) needs to intersect two sets $N(u)$ and $N(v)$ which is costly, we propose to first apply a filtering for the pair u and v based on an upper bound ub(u, v) of sim(u, v) (Line 5); if ub$(u, v) < \varepsilon$, then we have sim$(u, v) \leq$ ub$(u, v) < \varepsilon$ and thus $v \notin \Gamma(u)$. For the similarity in Definition 1, it is easy to verify that sim$(u, v) \leq \frac{\min\{d(u),d(v)\}}{\max\{d(u),d(v)\}}$; we set this as ub$(u, v)$, which can be calculated in constant time.[4] indexedSN is expected to run faster than SimNei (Algorithm 3) as the former can skip an entire segment if its $\mathsf{s}_{\max}$ is smaller than ε.

Index-based Two-Phase Vertex Reduction. Based on indexedSN, we can speed up VReduce (Algorithm 2) by invoking indexedSN to compute $\Gamma(u)$. However, this is still inefficient, as VReduce needs to compute $\Gamma(u)$ for all $u \in V_L$ (see Line 2 of Algorithm 2). We propose to utilize the index $\mathcal{I}$ to first obtain an upper bound of the similar degree for vertex reduction, as proved in the lemma below.

Lemma 4 (Upper Bound of Similar Degree) *Let $\mathbb{S}_u$ be the set of segments that cover Φ_u. Then, the similar degree $\delta_\varepsilon(u)$ of u is upper bounded by*

$$\sum_{\mathsf{seg} \in \mathbb{S}_u :\, \mathsf{seg.s}_{\max} \geq \varepsilon} \mathsf{seg}.c$$

Proof This lemma directly follows from the fact that sim$(u, v) < \varepsilon$ for all $v \in \bigcup_{\mathsf{seg} \in \mathbb{S}_u :\, \mathsf{seg.s}_{\max} < \varepsilon} V(\mathsf{seg})$. □

Consider the part of the index in Fig. 4 and suppose $\varepsilon = 0.4$. By scanning $\mathbb{S}_u$, we obtain an upper bound of u's similar degree as 6, i.e., seg$_1.c$ + seg$_3.c = 6$; seg$_2$ is omitted since its $\mathsf{s}_{\max}$ is only 0.2.

Algorithm 5: indexedVR$(G, \mathcal{I}, \varepsilon, \tau, \mathsf{del}(\cdot))$

```
/* Phase-I: vertex reduction based on structural
   degree and upper bound of similar degree                */
1 for each $u \in V_L \cup V_R$ do $d(u) \leftarrow |N(u)|$ for each $u \in V_L$ do
  $\bar{\delta}(u) \leftarrow \sum_{\mathsf{seg} \in \mathbb{S}_u : \mathsf{seg.s}_{\max} \geq \varepsilon}$ seg.$c$ while $(\exists u \in V_L \cup V_R$ s.t.
  del$(u)$ = false and $d(u) < \tau)$ or $(\exists u \in V_L$ s.t. del$(u)$ = false
  and $\bar{\delta}(u) < \tau - 1)$ do
2     for each $v \in N(u)$ do $d(v) \leftarrow d(v) - 1$ del$(u) \leftarrow$ true;
/* Phase-II: vertex reduction based on structural
   degree and similar degree                               */
3 for each $u \in V_L \cup V_R$ do del2$(u) \leftarrow$ del$(u)$ for each $u \in V_L$ s.t.
  del$(u)$ = false do
4     $(\delta_p(u)$, idx$(u)) \leftarrow$
      progressiveSN$(u, \varepsilon, G, \mathcal{I}$, del2$(\cdot), \tau - 1, 1)$;
5 while $(\exists u \in V_L \cup V_R$ s.t. del$(u)$ = false and $d(u) < \tau)$ or
  $(\exists u \in V_L$ s.t. del$(u)$ = false and $\delta_p(u) < \tau - 1)$ do
6     for each $v \in N(u)$ do $d(v) \leftarrow d(v) - 1$ if $u \in V_L$ then
7         $\Gamma(u) \leftarrow$ indexedSN$(u, \varepsilon, G, \mathcal{I}$, del$(\cdot))$;
8         for each $v \in \Gamma(u)$ do
9             $\delta_p(v) \leftarrow \delta_p(v) - 1$;
10            if $\delta_p(v) = \tau - 2$ and $d(v) \geq \tau$ then
11                $(r$, idx$(v)) \leftarrow$
                  progressiveSN$(v, \varepsilon, G, \mathcal{I}$, del2$(\cdot)$, 1, idx$(v))$;
12                $\delta_p(v) \leftarrow \delta_p(v) + r$;
13    del$(u) \leftarrow$ true;

  Procedure progressiveSN$(u, \varepsilon, G, \mathcal{I}$, del2$(\cdot), t, b)$
  /* Let $\mathbb{S}_u$ be $\{$seg$_1$, seg$_2, \ldots,$ seg$_{|\mathbb{S}_u|}\}$          */
14 $r \leftarrow 0$;
15 for each $i \in \{b, b+1, \ldots, |\mathbb{S}_u|\}$ s.t. seg$_i$.$\mathsf{s}_{\max} \geq \varepsilon$ do
16     for each $v \in [$seg$_i$.$\mathsf{v}_{\min}$, seg$_i$.$\mathsf{v}_{\max}]$ do
17         if del2$(v)$ = false and $v \neq u$ then
18             if ub$(u, v) \geq \varepsilon$ and sim$(u, v) \geq \varepsilon$ then
19                 $r \leftarrow r + 1$;
20     if $r \geq t$ then return $(r, i + 1)$
21 return $(r, |\mathbb{S}_u| + 1)$
```

Furthermore, we also observe that the structural degree can be obtained efficiently. Thus, we propose a two-phase approach for vertex reduction, which first conducts vertex reduction by using structural degree and upper bound of similar degree in Phase-I, and then using structural degree and similar degree in Phase-II. The pseudocode of our two-phase vertex reduction is shown in Algorithm 5, denoted indexedVR. In Phase-I, we first obtain the structural degree $d(u)$ for each $u \in V_L \cup V_R$ (Line 1), and an upper bound $\bar{\delta}(u)$ of the similar degree for each vertex $u \in V_L$ (Line 2). Then, as long as there is a non-deleted vertex $u \in V_L \cup V_R$ satisfying $d(u) < \tau$ or a non-deleted vertex $u \in V_L$ satisfying $\bar{\delta}(u) < \tau - 1$ (Line 3), we mark u as deleted and update the structural degree of its structural neighbors (Lines 4–5); note that, we do not update $\bar{\delta}(\cdot)$ in Phase-I. In Phase-II, we first compute a *progressive* similar degree, denoted $\delta_p(\cdot)$, for each non-deleted L-side vertex, by invoking progressiveSN (Lines 7–8). Here, $\delta_p(u)$ is a lower bound of u's similar

[4] The upper bound for cosine similarity is $\frac{\min\{d(u),d(v)\}}{\sqrt{d(u) \times d(v)}}$, while the upper bound for hub promoted index is 1 and thus not useful.

degree $\delta(u)$, and it records the number of similar neighbors that have been computed for u; our computation of $\delta_p(u)$ ensures that $\delta_p(u) \geq \tau - 1$ if and only if $\delta(u) \geq \tau - 1$. Then, as long as there is a non-deleted vertex $u \in V_L \cup V_R$ satisfying $d(u) < \tau$ or a non-deleted vertex $u \in V_L$ satisfying $\delta_p(u) < \tau - 1$, we mark u as deleted (Line 18) and update the structural degree of its structural neighbors (Line 10). Furthermore, if u is an L-side vertex, we also obtain the set $\Gamma(u)$ of similar neighbors of u (Line 12) and update the progressive similar degree $\delta_p(v)$ to satisfy the invariant that $\delta_p(v) \geq \tau - 1$ if and only if $\delta(v) \geq \tau - 1$ for each $v \in \Gamma(u)$ (Lines 13–17). Note that, in our implementation, we use a queue to maintain the vertices that satisfy the condition at Line 3 or Line 9; as a result, we do not need to loop through all non-deleted vertices to find the unpromising vertices.

In Algorithm 5, for an L-side vertex u, we compute $\delta_p(u)$ instead of $\delta(u)$. Our main motivation is that for an L-side vertex u satisfying $d(u) \geq \tau$, we only need to compute $\tau - 1$ of its similar neighbors to certify that it is a promising vertex. That is, we stop the computation of $\Gamma(u)$ once $\delta_p(u) \geq \tau - 1$; however, if some similar neighbors of u are later removed (i.e., marked as deleted), then we need to update $\delta_p(u)$ by computing more similar neighbors of u (Lines 15–17 of Algorithm 5). As a result, for vertices with high similar degrees in the remaining graph (i.e., obtained by removing all unpromising vertices), we only need to compute a small portion of their similar neighbors to prevent them from being removed and thus save unnecessary similar neighbor computations. The pseudocode of computing $\delta_p(u)$ is shown in Lines 19–26 of Algorithm 5, denoted progressiveSN. It is invoked only when $\delta_p(u) < \tau - 1$ and there are still unchecked segments of Φ_u. In progressiveSN, we check the segments of $\mathbb{S}_u$ one by one (Line 20–24) and stop once we have found enough similar neighbors for u (Line 25). We record the index of the first unchecked segment in $\mathsf{idx}(u)$ (Line 8). Note that in Algorithm 5, a copy of $\mathsf{del}(\cdot)$ is stored in $\mathsf{del2}(\cdot)$ at Line 6. The usage of $\mathsf{del2}(\cdot)$ is specific to the progressiveSN procedure. This distinction of $\mathsf{del}(\cdot)$ and $\mathsf{del2}(\cdot)$ is necessary because for each L-side vertex v, we compute its similar neighbors and thus similar degree $\delta_p(v)$ progressively (Lines 8 and 16). When we remove an unqualified vertex u at Lines 10–18, we also decrement $\delta_p(v)$ by one for each similar neighbor v of u (Line 14). Consequently, it is possible that u has not yet been considered in $\delta_p(v)$ when u is removed, but $\delta_p(v)$ is decremented by one due to the removal of u. As a remedy, progressiveSN considers all vertices that are alive when entering the while loop of Line 9, i.e., all vertices with $\mathsf{del2}(\cdot)$ being false.

indexedVR is better than VReduce (Algorithm 2), since (1) Phase-I of indexedVR is lightweight but very effective at pruning vertices as demonstrated by our empirical studies, and (2) indexedVR uses indexedSN and progressiveSN to compute the similar neighbors.

Overall Algorithm. Our index-based MSBE improves on Algorithm 1 by replacing the invocation to VReduce at Line 2 with invoking indexedVR for vertex reduction, and invoking indexedSN to compute $\Gamma(u)$ at Lines 6 and 15. Nevertheless, the time complexity of index-based MSBE remains $\mathcal{O}(|V_L| \cdot |E| \cdot 2^{|V_L|})$ as proved in Theorem 2, by noting that the time complexity of indexedSN remains $\mathcal{O}(|E|)$. Despite of having the same time complexity, our empirical studies in Sect. 7 show that the index-based approach can improve the efficiency of MSBE by several orders of magnitude.

5.3 Index construction

In this subsection, we present two algorithms to construct the index based on the ideas of *largest gap* and *steady segment*, respectively. Note that, the indexes are constructed offline, and once constructed, they can be used to process maximal similar-biclique enumeration queries with different ε and τ values.

Largest Gap (LG) Index. Recall that, our index structure summarizes a subset of vertices of Φ_u and their similarities to a vertex u by four numbers $\mathsf{seg} = \langle \mathsf{v}_{\min}, \mathsf{v}_{\max}, \mathsf{s}_{\max}, c \rangle$, where $\mathsf{s}_{\max}$ is an upper bound of the similarity between u and each $v \in \Phi_u$ such that $\mathsf{v}_{\min} \leq v \leq \mathsf{v}_{\max}$. To obtain the similar neighbors of u that are in the range $[\mathsf{v}_{\min}, \mathsf{v}_{\max}]$, we need to go through each vertex $v \in [\mathsf{v}_{\min}, \mathsf{v}_{\max}]$ and test its similarity with u (e.g., see Line 3 of Algorithm 4) even if $v \notin V(\mathsf{seg})$. We call a vertex v that is in the range $[\mathsf{v}_{\min}, \mathsf{v}_{\max}]$ but not in $V(\mathsf{seg})$ a *fake vertex*.

Intuitively, we should minimize the number of fake vertices when constructing the index. We call the index built by this strategy the largest gap (LG) index. It is constructed as follows. Suppose we are going to cover Φ_u by k segments. This is equivalent to find $k - 1$ cut points in the sequence of vertices of Φ_u that are ordered in increasing vertex id order; denote the sequence of vertices as $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$. Note that, the vertex ids are not consecutive, i.e., it is possible that $v_2 - v_1 > 1$. We represent the $k - 1$ cut points by $k - 1$ index values $\{\ell_1, \cdots, \ell_{k-1}\}$ such that $1 < \ell_1 < \ell_2 < \cdots < \ell_{k-1} < |\Phi_u|$, i.e., the i-th cut point is between vertex $v_{\ell_i - 1}$ and vertex v_{ℓ_i}. Define $\ell_0 = 1$ and $\ell_k = |\Phi_u| + 1$. Then, segment seg_i covers vertices $\{v_{\ell_{i-1}}, \ldots, v_{\ell_i - 1}\}$, for $1 \leq i \leq k$. Let f_1 be the number of fake vertices if we cover Φ_u by only one segment, i.e., $f_1 = v_{|\Phi_u|} - v_1 + 1 - |\Phi_u|$ where $v_{|\Phi_u|} - v_1 + 1$ is the total number of vertices in the range $[v_1, v_{|\Phi_u|}]$. It is easy to verify that the number of fake vertices of covering Φ_u by k segments with the $k - 1$ cut points $\{\ell_1, \ldots, \ell_{k-1}\}$ is $f_1 - \sum_{i=1}^{k-1}(v_{\ell_i} - v_{\ell_i - 1} + 1)$. As f_1 is a fixed number for Φ_u, minimizing the number of fake vertices is equivalent to maximizing $\sum_{i=1}^{k-1}(v_{\ell_i} - v_{\ell_i - 1} + 1)$.

Definition 5 (*Gap*) Given the sequence of vertices $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$ of Φ_u that are ordered in increasing vertex id order,

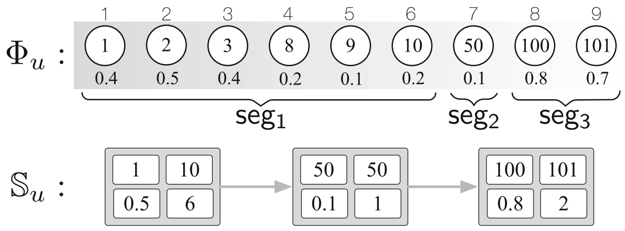


Fig. 5 Example of consLG

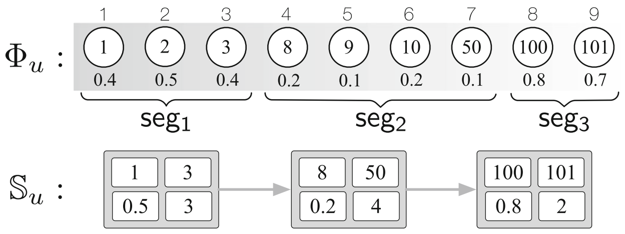


Fig. 6 Example of consSS

the *gap* of vertex v_i for $i > 1$ is defined as $v_i - v_{i-1} + 1$; the gap of v_1 is defined as 0.

Thus, the LG index constructs k segments to cover Φ_u, where the $k-1$ cut points are the $k-1$ vertices with the largest gaps.

Example 4 Figure 5 shows that the three segments constructed by the largest gap strategy for the same Φ_u as Example 3. The two vertices with the largest gap are v_{100} and v_{50}.

Choosing the number of segments to cover Φ_u. It is easy to see that the more the number of segments, the fewer the number of fake vertices introduced by the segments. In the extreme case of covering Φ_u by $|\Phi_u|$ segments, there will be no fake vertices introduced. However, the space complexity would be too high to be practical as discussed in Sect. 5.1. Thus, we set the number of segments for covering Φ_u as $\alpha \cdot \log |\Phi_u|$ where α is a user defined parameter, in viewing that a fixed number for different Φ_u will not work as $|\Phi_u|$ varies a lot across different vertices u.

Steady Segment (SS) Index. The LG index ignores the similarities (between u and different vertices) in a segment and thus may result in a very wide range of similarity values for a segment. This is not good for indexedSN and progressiveSN, as they need to check all vertices covered by a segment seg even if there is only one vertex in seg whose similarity to u is no lower than ε. Motivated by this, we aim to construct *steady segments* such that all similarities in a segment are close to each other.

Definition 6 (*Steady Segment*) Given a steady threshold $0 < \gamma < 1$, a segment $\mathsf{seg} = \langle \mathsf{v}_{\min}, \mathsf{v}_{\max}, \mathsf{s}_{\max}, c\rangle$ of Φ_u is steady if

$$\max_{v \in V(\mathsf{seg})} \mathsf{sim}(u, v) - \min_{v \in V(\mathsf{seg})} \mathsf{sim}(u, v) \leq \gamma$$

The first term, $\max_{v \in V(\mathsf{seg})} \mathsf{sim}(u, v)$, is exactly $\mathsf{seg.s}_{\max}$. For ease of presentation, we denote the second term, $\min_{v \in V(\mathsf{seg})} \mathsf{sim}(u, v)$, by $\mathsf{seg.s}_{\min}$, the smallest similarity value. A segment seg is steady if $\mathsf{seg.s}_{\max} - \mathsf{seg.s}_{\min} \leq \gamma$. The main advantage of a steady segment is that if seg is steady and satisfies $\mathsf{seg.s}_{\max} \geq \varepsilon$, then it is likely that many vertices of $V(\mathsf{seg})$ have similarity values to u no lower than ε, and thus, most of the computation will not be wasted.

Ideally, we would like to find the minimum number of steady segments to cover Φ_u. However, the number of steady segments required could be very large. For example, if the steady threshold γ is very close to 0 and all vertices of Φ_u have different similarity values to u, then the number of required steady segments to cover Φ_u is $|\Phi_u|$. Thus, we instead construct a fixed number of steady segments to cover as many vertices of Φ_u as possible and then cover the remaining uncovered vertices of Φ_u by as few segments as possible by ignoring the difference between the similarity values.

Given γ and k, our problem is to find k steady segments to cover as many vertices of Φ_u as possible. We first construct, for each vertex $v \in \Phi_u$, a maximal steady segment seg_v that starts at v (i.e., $\mathsf{seg}_v.\mathsf{v}_{\min} = v$), and then select k segments $\mathbb{S}^*$ from $\{\mathsf{seg}_v \mid v \in \Phi_u\}$ such that $\left|\bigcup_{v \in \mathbb{S}^*} V(\mathsf{seg}_v)\right|$ is maximized. This is an instance of the maximum k-coverage problem which is NP-hard [40]. We select the k segments in a greedy manner. That is, the k segments are selected one-by-one. Let S be the starting vertices of the currently selected segments. Then, the next segment to be added to S is $\arg\max_{v \in \Phi_u} \left|\bigcup_{v' \in S \cup \{v\}} V(\mathsf{seg}_{v'})\right|$. As this function is submodular, the greedy approach achieves an approximation ratio of $1 - \frac{1}{e}$ [17].

The pseudocode is shown in Algorithm 6, denoted consSS. For each vertex u, we first compute its 2-hop structural neighbors Φ_u and their similarities to u (Line 2) and then invoke MaximalSteadySegments to compute maximal steady segments that start at each vertex $v_i \in \Phi_u$ (Line 3). In MaximalSteadySegments, the maximal steady segment seg_{v_i} that starts at v_i is computed by iteratively trying to add the next vertex to the segment (Lines 19–27). Next, we iteratively add to $\mathbb{S}_u$ the segment of $\mathbb{C}$ that covers the largest number of uncovered vertices of Φ_u (Lines 4–12); as a result, after adding a segment into $\mathbb{S}_u$, we also need to update the remaining segments of $\mathbb{C}$ to be disjoint from the segments of $\mathbb{S}_u$ (Lines 8–12). During this process, for time efficiency consideration, we do not maintain $\mathsf{seg.s}_{\max}$; instead, we compute $\mathsf{seg.s}_{\max}$ for each segment $\mathsf{seg} \in \mathbb{S}_u$ later (Line 13). Finally, we create the minimum number of segments to cover all vertices of Φ_u that are not covered by $\mathbb{S}_u$ (Lines 14–15).

Algorithm 6: consSS$(G = (V_L, V_R, E), \alpha, \gamma)$

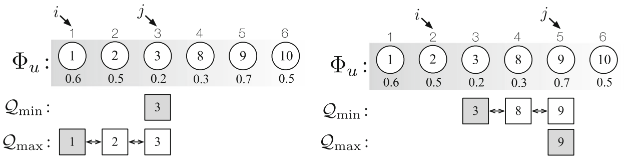
```
1  for each $u \in V_L \cup V_R$ do
2      $\Phi_u \leftarrow$ SimNei$(G, u, \frac{1}{2|V_L|+2|V_R|})$;
3      $\mathbb{C} \leftarrow$ MaximalSteadySegments$(G, u, \Phi_u, \gamma)$;
4      $\mathbb{S}_u \leftarrow \emptyset$;  $k \leftarrow \min\{|\Phi_u|, \alpha \cdot \log |\Phi_u|\}$;
5      while $|\mathbb{S}_u| < k$ and $\mathbb{C} \neq \emptyset$ do
6          seg$^* \leftarrow \arg\max_{\mathsf{seg} \in \mathbb{C}}$ seg.$c$;
7          $\mathbb{S}_u \leftarrow \mathbb{S}_u \cup \{$seg$^*\}$;
8          for each seg $\in \mathbb{C}$ do
9              if seg$^*$.v$_{\min}$ < seg.v$_{\min}$ $\leq$ seg$^*$.v$_{\max}$ then
10                 Remove seg from $\mathbb{C}$
11             else if seg.v$_{\min}$ < seg$^*$.v$_{\min}$ $\leq$ seg.v$_{\max}$ then
12                 Let $v$ be the vertex that immediately precedes
                   seg$^*$.v$_{\min}$ in $\Phi_u$, change seg.v$_{\max}$ to $v$, and update
                   seg.$c$ accordingly in $\mathbb{C}$;
13     for each seg $\in \mathbb{S}_u$ do Compute seg.s$_{\max}$ for each maximal
       consecutive sequence of vertices $v_i, v_{i+1}, \ldots, v_j$ of $\Phi_u$ that
       are not covered by $\mathbb{S}_u$ do
14         Add to $\mathbb{S}_u$ the segment that covers $\{v_i, \ldots, v_j\}$;
15 return $\mathcal{I} \leftarrow \{\mathbb{S}_u \mid u \in V_L \cup V_R\}$
   Procedure MaximalSteadySegments$(G, u, \Phi_u, \gamma)$
16 $\mathbb{C} \leftarrow \emptyset$;
17 Let $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$ be vertices of $\Phi_u$ in increasing vertex id
   order;
18 for $i \leftarrow 1$ to $|\Phi_u|$ do
19     s$_{\min} \leftarrow 1$;  s$_{\max} \leftarrow 0$;
20     for $j \leftarrow i$ to $|\Phi_u|$ do
21         if sim$(u, v_j)$ < s$_{\min}$ then s$_{\min} \leftarrow$ sim$(u, v_j)$ if
           sim$(u, v_j)$ > s$_{\max}$ then s$_{\max} \leftarrow$ sim$(u, v_j)$ if
           s$_{\max}$ − s$_{\min}$ > $\gamma$ then
22             seg$_{v_i} \leftarrow \langle v_i, v_{j-1}$, null, $j - i\rangle$;
23             break;
24     $\mathbb{C} \leftarrow \mathbb{C} \cup \{$seg$_{v_i}\}$;
25 return $\mathbb{C}$
```

Example 5 Figure 6 shows the three steady segments constructed for the same Φ_u in Examples 3 and 4, where $\gamma = 0.1$.

Time complexity of consSS. For each vertex $u \in V_L \cup V_R$, Line 2 of Algorithm 6 takes $O(\sum_{v \in N(u)} d(v))$ time, and Line 3 takes $O(|\Phi_u|^2)$ time. The while loop at Line 5 runs for at most $\alpha \cdot \log |\Phi_u|$ iterations, and each iteration takes $O(|\Phi_u|)$ time. Lines 13–15 take $O(|\Phi_u|)$ time. Thus, the total time complexity of consSS is

$$O\Big(\sum_{u \in V_L \cup V_R} \big(|\Phi_u|^2 + \sum_{v \in N(u)} d(v)\big)\Big)$$

Computing all maximal steady segments in linear time. From the above time complexity analysis of consSS, we can see that the time complexity of computing the initial maximal steady segments for all vertices dominates the total time complexity of consSS. In view of this, we propose an efficient algorithm to compute the maximal steady segments for all vertices in linear time, i.e., $O(|\Phi_u|)$. The general idea is based on the following lemma.



(a) Processing vertex v_1 (b) Processing vertex v_2

Fig. 7 Computing maximal steady segments ($\gamma = 0.3$)

Lemma 5 *Let $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$ be the vertices of Φ_u sorted in increasing vertex id order. For any $1 \leq i < |\Phi_u|$, the largest vertex id in the maximal steady segment of v_{i+1} is no less than that of v_i, i.e.,* seg$_{v_{i+1}}$.v$_{\max} \geq$ seg$_{v_i}$.v$_{\max}$.

Proof This is easy to see since the segment covering vertices from v_{i+1} to seg$_{v_i}$.v$_{\max}$, which is a subset of $V($seg$_{v_i})$, must be steady. □

Following Lemma 5, we use two pointers, i and j, to compute the maximal steady segments for all vertices of Φ_u. Here, i is the index of the start vertex and j is 1 plus the index of seg$_{v_i}$.v$_{\max}$ in Φ_u. We will increase i from 1 to $|\Phi_u|$; according to Lemma 5, j will not decrease. For initialization, both i and j are set as 1. To compute seg$_{v_i}$, we iteratively increase j by 1 until either $j = |\Phi_u| + 1$ or the segment covering vertices $\{v_i, \ldots, v_j\}$ is not steady; when this process finishes, we set seg$_{v_i}$.v$_{\max} = v_{j-1}$. After computing seg$_{v_i}$, we continue to compute the maximal steady segment for the next vertex by increasing i by 1. Note that we do not reset j after increasing i; this is correct according to Lemma 5. For example, for Φ_u in Fig. 7, when $i = 1$, we have $j = 3$ and thus seg$_{v_1}$.v$_{\max} = v_{j-1} = v_2$; when $i = 2$, we have $j = 5$ and thus seg$_{v_2}$.v$_{\max} = v_4$. Since both i and j are non-decreasing, the time complexity would be $c \cdot |\Phi_u|$ where c is the time complexity of checking whether the segment covering a given set of consecutive vertices is steady or not.

To check whether the segment covering a given set S of consecutive vertices of Φ_u is steady or not, all we need to obtain are $\min_{v \in S}$ sim(u, v) and $\max_{v \in S}$ sim(u, v). Since S consists of *consecutive* vertices of Φ_u, we could use some advanced data structures (e.g., segment tree or the one presented in our conference version [52]) to obtain $\min_{v \in S}$ sim(u, v) and $\max_{v \in S}$ sim(u, v) in $\log |\Phi_u|$ time. In this work, we propose to use a simple data structure (specifically, a queue) to achieve this in amortized constant time. We focus our discussion on how to obtain $\min_{v \in S}$ sim(u, v); note that $\max_{v \in S}$ sim(u, v) can be obtained in a similar way. Our data structure stores vertices of S that are not min-dominated by other vertices of S.

Definition 7 (*Min-Domination*) Given a vertex u and a vertex subset $S \subseteq \Phi_u$, we say that $v \in S$ *min-dominates* $v' \in$

S if (1) v has a larger vertex id than v' (i.e., $v > v'$) and $\mathsf{sim}(u, v) \leq \mathsf{sim}(u, v')$.

It is easy to see that $\min_{v' \in S} \mathsf{sim}(u, v')$ is equal to $\mathsf{sim}(u, v)$ for v being the smallest vertex of S that is not min-dominated by other vertices of S. We store in a queue $\mathcal{Q}_{\min}$ the list of vertices of S that are not min-dominated by other vertices of S; note that storing all the vertices (instead of just the smallest such vertex) that are not min-dominated will make our maintaining of $\mathcal{Q}_{\min}$ easy. As vertices in $\mathcal{Q}_{\min}$ are stored in increasing id order, their similarities to u are also strictly increasing. Suppose $S = \{v_i, \ldots, v_j\}$ and $\mathcal{Q}_{\min}$ currently stores the non-min-dominated vertices of S. To maintain $\mathcal{Q}_{\min}$ for $S' = \{v_{i+1}, \ldots, v_{j'}\}$ where $j' \geq j$, we only need to

- remove v_i from $\mathcal{Q}_{\min}$ if $v_i \in \mathcal{Q}_{\min}$,
- remove the longest suffix of $\mathcal{Q}_{\min}$ that is min-dominated by a vertex of $\{v_{j+1}, \ldots, v_{j'}\}$, and
- add all vertices of $\{v_{j+1}, \ldots, v_{j'}\}$ that are not min-dominated by any vertex of S'.

We note that all these operations can be conducted in amortized constant time, which will be discussed shortly.

Algorithm 7: $\mathsf{TPA}(G, u, \Phi_u, \gamma)$

```
1  $\mathbb{C} \leftarrow \emptyset$;
2  Let $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$ be vertices of $\Phi_u$ in increasing vertex id
   order;
3  $i \leftarrow 1, j \leftarrow 1$;
4  $\mathcal{Q}_{\min} \leftarrow \{v_1\}, \mathcal{Q}_{\max} \leftarrow \{v_1\}$;
5  while $i \leq |\Phi_u|$ do
6      if $\mathcal{Q}_{\min}.front() < v_i$ then $\mathcal{Q}_{\min}.pop\_front()$ if
       $\mathcal{Q}_{\max}.front() < v_i$ then $\mathcal{Q}_{\max}.pop\_front()$ while
       $\mathcal{Q}_{\min}.empty()$ or $\mathcal{Q}_{\max}.empty()$ or
       $\mathsf{sim}(u, \mathcal{Q}_{\max}.front()) - \mathsf{sim}(u, \mathcal{Q}_{\min}.front()) \leq \gamma$ do
7          if $j \leq |\Phi_u|$ then $j \leftarrow j + 1$ if $j > |\Phi_u|$ then break
           while $!\mathcal{Q}_{\min}.empty()$ and
           $\mathsf{sim}(u, \mathcal{Q}_{\min}.back()) \geq \mathsf{sim}(u, v_j)$ do
8              $\mathcal{Q}_{\min}.pop\_back()$
9          $\mathcal{Q}_{\min}.push\_back(v_j)$;
10         while $!\mathcal{Q}_{\max}.empty()$ and
           $\mathsf{sim}(u, \mathcal{Q}_{\max}.back()) \leq \mathsf{sim}(u, v_j)$ do
11             $\mathcal{Q}_{\max}.pop\_back()$
12         $\mathcal{Q}_{\max}.push\_back(v_j)$;
13     $\mathsf{seg}_{v_i} \leftarrow \langle v_i, v_{j-1}, \mathsf{null}, j - i \rangle$;
14     $\mathbb{C} \leftarrow \mathbb{C} \cup \{\mathsf{seg}_{v_i}\}$;
15     $i \leftarrow i + 1$;
16 return $\mathbb{C}$
```

The pseudocode of our linear-time algorithm for computing all maximal steady segments is shown in Algorithm 7. Same as $\mathsf{MaximalSteadySegments}$ in Algorithm 6, we first sort vertices of Φ_u into increasing vertex id order (Line 2); let $\{v_1, v_2, \ldots, v_{|\Phi_u|}\}$ be the vertices in sorted order. We initialize the two pointers i and j to both be 1 (Line 3) and initialize two queues $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\max}$ (used for obtaining the minimum and the maximum similarity in a segment, respectively) by v_1 (Line 4). Then, we iteratively compute the maximal steady segment for v_i, $1 \leq i \leq |\Phi_u|$ (Lines 5–19). To compute the maximal steady segment for v_i, we first pop the front element from $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\min}$ if it is smaller than v_i (Lines 6–7), and then keep increasing j if the segment covering vertices $\{v_i, \ldots, v_j\}$ is still steady (Lines 8–16). The testing of whether the segment covering vertices $\{v_i, \ldots, v_j\}$ is still steady or not is achieved by comparing $\mathsf{sim}(u, \mathcal{Q}_{\max}.front()) - \mathsf{sim}(u, \mathcal{Q}_{\min}.front())$ with γ (Line 8); as discussed above, $\mathcal{Q}_{\min}.front()$ stores the vertex that has the lowest similarity to u among vertices of $\{v_i, \ldots, v_j\}$ while $\mathcal{Q}_{\max}.front()$ stores the one with the highest similarity. Note that we consistently have $j \geq i$ throughout the entire computation since the segment covering one vertex is trivially steady. After increasing j (Line 9) and j still being no larger than $|\Phi_u|$, we update $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\max}$ to store all the non-min-dominated and non-max-dominated vertices of $\{v_i, \ldots, v_j\}$, respectively (Lines 11–16). Specifically, to update $\mathcal{Q}_{\min}$, we first remove from it all vertices that are min-dominated by v_j (Lines 11-12), and then push v_j to the back of $\mathcal{Q}_{\min}$ (Line 13); $\mathcal{Q}_{\max}$ is updated similarly. Once we reach a j such that either $j > |\Phi_u|$ (Line 10) or the segment covering vertices $\{v_i, \ldots, v_j\}$ is no longer steady, we set the maximal steady segment seg_{v_i} of v_i to be $\langle v_i, v_{j-1}, \mathsf{null}, j - i \rangle$ (Line 17) and add it to $\mathbb{C}$ (Line 18).

Lines 5–19 of Algorithm 7 run in $O(|\Phi_u|)$ time, since (1) j will increase for $|\Phi_u|$ times and (2) each vertex of Φ_u will be inserted into and popped from $\mathcal{Q}_{\min}$ (resp. $\mathcal{Q}_{\max}$) at most once. Note that, in our implementation, we also store the similarities into $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\max}$ such that the similarity between u and any vertex at the front or back of $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\max}$ can be retrieved in constant time. Thus, the total time complexity of Algorithm 7 is dominated by Line 2 which takes $O(|\Phi_u| \log |\Phi_u|)$ time. By invoking Algorithm 7 at Line 3 of Algorithm 6, the time complexity of Algorithm 6 becomes $O\Big(\sum_{u \in V_L \cup V_R} \big(\alpha |\Phi_u| \log |\Phi_u| + \sum_{v \in N(u)} d(v)\big)\Big)$.

Example 6 Let's consider Φ_u in Fig. 7 and compute the maximal steady segments for its vertices with $\gamma = 0.3$; note that $(v_1, \ldots, v_6) = (1, 2, 3, 8, 9, 10)$. We initialize both i and j as 1, and initialize both $\mathcal{Q}_{\min}$ and $\mathcal{Q}_{\max}$ as $\{v_1\}$. To compute the maximal steady segments for v_1, we keep increasing j until the segment covering vertices $\{v_i, \ldots, v_j\}$ is no longer steady. When j is increased to 2, we update $\mathcal{Q}_{\min}$ as $\{v_2\}$ (note that v_1 is min-dominated by v_2 and thus removed from $\mathcal{Q}_{\min}$) and update $\mathcal{Q}_{\max}$ as $\{v_1, v_2\}$; the segment covering $\{v_1, v_2\}$ is steady as $\mathsf{sim}(u, \mathcal{Q}_{\max}.front()) - \mathsf{sim}(u, \mathcal{Q}_{\min}.front()) = 0.1 < \gamma$. When $j = 3$, we update

$\mathcal{Q}_{\min}$ as $\{v_3\}$ and update $\mathcal{Q}_{\max}$ as $\{v_1, v_2, v_3\}$, as shown in Fig. 7a. The segment covering $\{v_1, v_2, v_3\}$ is no longer steady since $\mathsf{sim}(u, \mathcal{Q}_{\max}.front()) - \mathsf{sim}(u, \mathcal{Q}_{\min}.front()) = 0.4 > \gamma$, and thus, $\mathsf{seg}_{v_1} = \langle v_1, v_2, \mathsf{null}, 2\rangle$.

Next, we increase i to 2 for computing the maximal steady segment for v_2; we also remove v_1 from $\mathcal{Q}_{\max}$ since $v_1 < v_i$. Same as illustrated above, we keep increasing j until the segment covering $\{v_2, \ldots, v_j\}$ is no longer steady. When j is still 3, $\mathcal{Q}_{\min} = \{v_3\}$ and $\mathcal{Q}_{\max} = \{v_2, v_3\}$ and thus the segment is steady. When $j = 4$, $\mathcal{Q}_{\min} = \{v_3, v_4\}$ and $\mathcal{Q}_{\max} = \{v_2, v_4\}$ and the segment is still steady. When $j = 5$, $\mathcal{Q}_{\min} = \{v_3, v_4, v_5\}$ and $\mathcal{Q}_{\max} = \{v_5\}$ as shown in Fig. 7b, and the segment is no longer steady. Thus, $\mathsf{seg}_{v_2} = \langle v_2, v_4, \mathsf{null}, 3\rangle$.

5.4 Parallelization

Our index construction algorithms can be easily parallelized, since processing Φ_u is totally independent from processing $\Phi_{u'}$ for $u \neq u'$. Thus, we propose to further speed up the index construction with shared-memory parallelization (i.e., handling different vertices concurrently with multiple threads). Specifically, we use OpenMP to parallelize the "for loop" at Line 1 of Algorithm 6 in our parallel implementation.

6 Index maintenance

In this section, we propose algorithms to dynamically maintain our index structures that are proposed in Sect. 5.3, as real-world graphs are consistently changing with old edges being removed and/or new edges being added. Note that the algorithms introduced in this section can be applied to both LG index and SS index.

Here, we mainly focus on edge deletion and insertion, by first discussing the case of edge deletion in Sect. 6.1 and then the case of edge insertion in Sect. 6.2. Note that vertex deletion (resp. insertion) can be handled as a sequence of edge deletion (resp. insertion) followed by (resp. following) isolated vertex deletion (resp. insertion). When removing a vertex from a graph, we preserve its id and do not assign the id to other vertices. When inserting a new vertex into the graph, its id is set as 1 plus the current largest id.

6.1 Edge deletion

Let (u, v) be the edge that is removed from G. Let Φ_u and Φ_v be the sets of 2-hop neighbors of u and v, respectively, in the graph G before removing the edge (u, v). After removing edge (u, v), among $\{N(w) \mid w \in V_L \cup V_R\}$, only $N(u)$ and $N(v)$ change. Thus, only similarities in $\{\mathsf{sim}(u, w) \mid w \in \Phi_u\} \cup \{\mathsf{sim}(v, w') \mid w' \in \Phi_v\}$ will change their value. For presentation simplicity, we assume that Φ_w does not change for any $w \in V_L \cup V_R$; that is, even if the similarity $\mathsf{sim}(w, w')$ decreases from a positive value to 0, we still consider w' to be in Φ_w. Thus, we have the following lemma.

Lemma 6 *When an edge (u, v) is removed from G, only the part $\{\mathbb{S}_w \mid w \in \Phi_u \cup \Phi_v \cup \{u, v\}\}$ of the index structure will change.*

In the following, we only discuss the index maintenance for $\mathbb{S}_u$ and $\mathbb{S}_w$ where $w \in \Phi_u$; note that $\mathbb{S}_v$ and $\mathbb{S}_{w'}$ for $w' \in \Phi_v$ can be maintained similarly. For any vertex $w \in \Phi_u$, we also have $u \in \Phi_w$ and thus there must exist a segment $\mathsf{seg} \in \mathbb{S}_w$ that covers u before the update, i.e., $\mathsf{seg}.\mathsf{v}_{\min} \leq u \leq \mathsf{seg}.\mathsf{v}_{\max}$. Consequently, we only need to update $\mathsf{seg}.\mathsf{s}_{\max}$ as $\max\{\mathsf{sim}(u, w), \mathsf{seg}.\mathsf{s}_{\max}\}$, where $\mathsf{sim}(u, w)$ is the updated similarity between u and w. Note that if $\mathsf{sim}(u, w) < \mathsf{seg}.\mathsf{s}_{\max}$, it is also possible to decrease $\mathsf{seg}.\mathsf{s}_{\max}$; but for updating efficiency consideration, we do not decrease $\mathsf{seg}.\mathsf{s}_{\max}$. For the same reason, we also do not update (i.e., decrease) $\mathsf{seg}.c$. For vertex u, $\mathbb{S}_u$ can be updated in a similar way; that is, for each vertex $w \in \Phi_u$, we find the segment $\mathsf{seg}' \in \mathbb{S}_u$ that covers w and update $\mathsf{seg}'.\mathsf{s}_{\max}$ as $\max\{\mathsf{sim}(u, w), \mathsf{seg}'.\mathsf{s}_{\max}\}$. Note that, we only need to update one segment for $\mathbb{S}_w$, but we may need to update multiple segments for $\mathbb{S}_u$.

Optimization. In the above process, if $\mathsf{sim}(u, w)$ becomes smaller after the update, then we actually do not update the segment that covers u or w. Thus, we can skip all such $w \in \Phi_u$ that $\mathsf{sim}(u, w)$ is smaller after removing the edge (u, v). Fortunately, this set of vertices can be easily characterized by the following lemma.

Lemma 7 *After removing edge (u, v) from the graph, the similarity $\mathsf{sim}(u, w)$ for $w \in \Phi_u$ decreases if and only if $(w, v) \in E$, and it increases otherwise.*

Proof Let $d(u)$ and $d(w)$ be the degrees of u and w before the update, and C be the number of common neighbors of u and w before the update. Then, $\mathsf{sim}(u, w) = \frac{C}{d(u)+d(w)-C}$ before the update. After removing (u, v), $d(w)$ remains unchanged while $d(u)$ decreases by 1. We consider two cases:

- If $(w, v) \in E$, the number of common neighbors between u and w will decrease by 1. Thus, the similarity between u and w after the update becomes $\mathsf{sim}(u, w) = \frac{C-1}{d(u)+d(w)-C}$, which is smaller than their similarity before the update.
- If $(w, v) \notin E$, the number of common neighbors between u and w will remain unchanged and $\mathsf{sim}(u, w) = \frac{C}{d(u)+d(w)-C-1}$, which is larger than their similarity before the update.

This completes the proof. □

Algorithm 8: EdgeDelete*$(G, u, v, \mathcal{I})$

```
1  UpdateDeletion($G, u, v, \mathcal{I}$);
2  UpdateDeletion($G, v, u, \mathcal{I}$);
3  Remove the edge $(u, v)$ from $G$;
   Procedure UpdateDeletion($G, u, v, \mathcal{I}$)
4  $\Phi_u \leftarrow$ SimNei($G, u, v, \frac{1}{2|V_L|+2|V_R|}$);
5  for each $w \in \Phi_u$ do
6      if $(w, v) \in E$ then
7          continue;   /* sim$(u, w)$ is decreasing */;
8      Obtain the seg $\in \mathbb{S}_w$ s.t. seg.v$_{\min} \le u \le$ seg.v$_{\max}$;
9      seg.s$_{\max} \leftarrow \max\{$sim$(u, w)$, seg.s$_{\max}\}$;
10     Obtain seg$' \in \mathbb{S}_u$ s.t. seg$'$.v$_{\min} \le w \le$ seg$'$.v$_{\max}$;
11     seg$'$.s$_{\max} \leftarrow \max\{$sim$(u, w)$, seg$'$.s$_{\max}\}$;
```

Following Lemma 7, when we process a 2-hop neighbor $w \in \Phi_u$ of u, we first check if $(w, v) \in E$. If $(w, v) \in E$, we know that the similarity between u and w must be decreasing and thus do nothing. The pseudocode of our algorithm for handling edge deletion is shown in Algorithm 8. We first invoke UpdateDeletion to update $\{\mathbb{S}_w \mid w \in \Phi_u \cup \{u\}\}$ (Line 1), then invoke UpdateDeletion again (but with different arguments) to update $\{\mathbb{S}_w \mid w \in \Phi_v \cup \{v\}\}$, and finally remove the edge (u, v) from the graph. The pseudocode of the procedure UpdateDeletion is shown in Lines 4–11 of Algorithm 8. We first invoke a variant of SimNei to compute u's old 2-hop neighbors (i.e., before removing (u, v)) and to compute the new (i.e., updated) similarity between u and each $w \in \Phi_u$ (Line 4); this can be done in a similar way to Algorithm 3, and we omit the details. Then, for each vertex $w \in \Phi_u$, if $(w, v) \in E$, we know from Lemma 7 that sim(w, u) must be decreasing and we skip w (Lines 6–7); otherwise, we update the index $\mathbb{S}_w$ and index $\mathbb{S}_u$ (Lines 8–11). Specifically, we search for the segment in $\mathbb{S}_w$ that covers u and update its s$_{\max}$ (Lines 8–9). Then, we search for the segment in $\mathbb{S}_u$ that covers w and update its s$_{\max}$ (Lines 10–11).

6.2 Edge insertion

Let (u, v) be the edge that is inserted into G. Let Φ_u and Φ_v be the sets of 2-hop neighbors of u and v, respectively, in the graph G after adding the edge (u, v). After adding edge (u, v), among $\{N(w) \mid w \in V_L \cup V_R\}$, only $N(u)$ and $N(v)$ change. Thus, only similarities in $\{$sim$(u, w) \mid w \in \Phi_u\} \cup \{sim(v, w') \mid w' \in \Phi_v\}$ will change their value. Thus, we have the following lemma.

Lemma 8 *When an edge (u, v) is inserted into G, only the part $\{\mathbb{S}_w \mid w \in \Phi_u \cup \Phi_v \cup \{u, v\}\}$ of the index structure will change.*

In the following, we only discuss the index maintenance for $\mathbb{S}_u$ and $\mathbb{S}_w$ where $w \in \Phi_u$; $\mathbb{S}_v$ and $\mathbb{S}_{w'}$ for $w' \in \Phi_v$ can be maintained similarly. To update $\mathbb{S}_w$ for $w \in \Phi_u$, we first try to find the segment in $\mathbb{S}_w$ that covers u (i.e., seg $\in \mathbb{S}_w$ s.t. seg.v$_{\min} \le u \le$ seg.v$_{\max}$). If there is such a segment seg, then we update seg.s$_{\max}$ as $\max\{$sim(u, w), seg.s$_{\max}\}$ and increase seg.c by 1 if the only common neighbor of u and w is v. Otherwise, there is no segment of $\mathbb{S}_w$ covering u (this means that u is a new 2-hop neighbor of w), then we create a new segment for u and add it into $\mathbb{S}_w$. The updating of $\mathbb{S}_u$ is conducted in a similar way.

Optimization. Similar to the optimization in Sect. 6.1, we can skip all such $w \in \Phi_u$ that sim(u, w) becomes smaller after adding the edge (u, v). This set of vertices is characterized by the following lemma.

Lemma 9 *After adding edge (u, v) into the graph, the similarity sim(u, w) for $w \in \Phi_u$ increases if and only if $(w, v) \in E$, and it decreases otherwise.*

Proof This can be proved in a similar way to the proof of Lemma 7. □

Algorithm 9: EdgeInsert-Opt$(G, u, v, \mathcal{I})$

```
1  Insert the new edge $(u, v)$ into $G$;
2  UpdateInsertion($G, u, v, \mathcal{I}$);
3  UpdateInsertion($G, v, u, \mathcal{I}$);
   Procedure UpdateInsertion($G, u, v, \mathcal{I}$)
4  for each $w \in N(v)$ do
5      Compute sim$(u, w)$;
6      if $\exists$seg$_i \in \mathbb{S}_w$ s.t. seg$_i$.v$_{\min} \le u \le$ seg$_i$.v$_{\max}$ then
7          seg$_i$.s$_{\max} \leftarrow \max\{$sim$(u, w)$, seg$_i$.s$_{\max}\}$;
8          if ComNei$(u, w) = 1$ then seg$_i.c \leftarrow$ seg$_i.c + 1$
9      else
10         seg $\leftarrow \langle u, u,$ sim$(u, w), 1\rangle$;
11         $\mathbb{S}_w \leftarrow \mathbb{S}_w \cup \{$seg$\}$;
12     if $\exists$seg$_i \in \mathbb{S}_u$ s.t. seg$_i$.v$_{\min} \le w \le$ seg$_i$.v$_{\max}$ then
13         seg$_i$.s$_{\max} \leftarrow \max\{$sim$(u, w)$, seg$_i$.s$_{\max}\}$;
14         if ComNei$(u, w) = 1$ then seg$_i.c \leftarrow$ seg$_i.c + 1$
15     else
16         seg $\leftarrow \langle w, w,$ sim$(u, w), 1\rangle$;
17         $\mathbb{S}_u \leftarrow \mathbb{S}_u \cup \{$seg$\}$;
```

Following Lemma 9, we only need to update the index for $w \in \Phi_u$ such that $(w, v) \in E$. Thus, we can design an algorithm EdgeInsert* in a similar way to EdgeDelete* (Algorithm 8); that is, similar to Lines 6–7 of Algorithm 8, we skip the processing of w if $(w, v) \notin E$. We omit the pseudocode of EdgeInsert*. This can significantly improve the updating time. Nevertheless, we can do even better, by noting that $\{w \in \Phi_u \mid (w, v) \in E\}$ is the same as $N(v)$. Thus, instead of first generating Φ_u and then skipping non-neighbors of v from Φ_u, we only process neighbors of v. The pseudocode of our further optimized algorithm is shown in Algorithm 9, denoted EdgeInsert-Opt. We first insert the edge (u, v) into the graph, then invoke UpdateInsertion to update $\{\mathbb{S}_w \mid w \in \Phi_u \cup \{u\}\}$, and finally invoke UpdateInsertion again (but with different arguments) to update $\{\mathbb{S}_w \mid w \in \Phi_v \cup \{v\}\}$. The pseudocode of the procedure UpdateInsertion is shown in Lines 4–17 of Algorithm 9. In UpdateInsertion, for each vertex $w \in N(v)$ (Line 4), we first compute $\mathsf{sim}(u, w)$ (Line 5) by computing the number of common neighbors of u and w, denoted $\mathsf{ComNei}(u, w)$, and then update the index $\mathbb{S}_w$ (Lines 6–11) and the index $\mathbb{S}_u$ (Lines 12–17). Specifically, to update $\mathbb{S}_w$ by u and $\mathsf{sim}(u, w)$, we search for the segment seg_i in $\mathbb{S}_w$ that covers u and update $\mathsf{seg}_i.\mathsf{s}_{\max}$ if necessary (Lines 6–8). If the number of common neighbors between u and w is 1, we know that u is w's new 2-hop neighbor and we thus increase $\mathsf{seg}_i.c$ by 1 (Line 8); note that, even if $\mathbb{S}_w$ has a segment covering u, it is still possible that u is w's new 2-hop neighbor because segments contain fake vertices. If $\mathbb{S}_w$ has no segment covering u (this means that u is w's new 2-hop neighbor), we create a new segment for u and add it to $\mathbb{S}_w$ (Lines 9–11). Update $\mathbb{S}_u$ by w and $\mathsf{sim}(u, w)$ is conducted similarly.

EdgeInsert* vs EdgeInsert-Opt. Both EdgeInsert* and EdgeInsert-Opt have their advantages and disadvantages. EdgeInsert* needs to construct the full set Φ_u of u's 2-hop neighbors and compute the similarity between u and each vertex $w \in \Phi_u$. The set Φ_u could be large compared with $N(v)$ that is processed by EdgeInsert-Opt. Nevertheless, the similarity between u and each $w \in N(v)$ is potentially computed more efficiently in EdgeInsert*, since it invokes SimNei to compute all the similarities at the same time which enables computation sharing among the similarity computation of different vertices. In contrast, EdgeInsert-Opt needs to conduct set intersection to compute $N(u) \cap N(w)$ which is used in calculating $\mathsf{sim}(u, w)$. Our empirical study shows that EdgeInsert-Opt performs better in practice.

6.3 Index reconstruction

To achieve high updating efficiency, our index maintenance strategy discussed above only maintains a relaxed version of our index. For example, $\mathsf{seg}.\mathsf{s}_{\max}$ is only guaranteed to be an upper bound (i.e., can be larger than) $\max_{v \in V(\mathsf{seg})} \mathsf{sim}(u, v)$; this is because we didn't update $\mathsf{seg}.\mathsf{s}_{\max}$ when the similarity between u and a vertex covered by seg becomes smaller. Also, after inserting an edge (u, v), when we cannot find a segment in $\mathbb{S}_u$ that covers u's new 2-hop neighbor w, we create a new segment for w and insert this segment into $\mathbb{S}_u$. As a result, the effectiveness of using our index to speed up query processing may deteriorate when a large number of updates has occurred. In view of this, we propose to reconstruct the index from scratch when the number of updates exceeds a threshold. To achieve this, we record the number of updates that have occurred, including vertex insertion/deletion and edge insertion/deletion. We reconstruct the entire index structure from scratch once this number is larger than $\zeta \times |E|$, where ζ is the parameter to control the frequency of index rebuilding and $|E|$ is the number of edges in the bipartite graph for which the index was last time reconstructed. Note that, when we reconstruct the index from scratch, we also reassign the vertex id to make sure that vertices of V_L take (integer) ids from $\{1, 2, \ldots, |V_L|\}$, and vertices of V_R take ids from $\{1 + |V_L|, 2 + |V_L|, \ldots, |V_R| + |V_L|\}$.

7 Experiments

In this section, we evaluate the efficiency of our algorithms as well as the effectiveness of our similar-biclique model.

Algorithms. We compare the following algorithms.

– ooMBEA: the state-of-the-art algorithm proposed in [8] for enumerating all maximal bicliques.
– MSBE: our Algorithm 1 equipped with all the optimizations in Sect. 4.2.
– mat-MSBE: the materialized version of MSBE, as discussed at the end of Sect. 4.2.
– LG-MSBE and SS-MSBE: our index-based algorithms that use the largest gap and steady segment index, respectively.

The source code of our algorithms can be found at https://github.com/kyaocs/Similar-Biclique-Idx. The source code of ooMBEA is obtained from the authors of [8].

All our algorithms are implemented in C++ and run in main memory. All experiments, except the one on evaluating our parallel index construction algorithm, are conducted on a machine with an Intel(R) 3.2GHz CPU and 64GB main memory running Ubuntu 18.04.5. We set a timeout of 10 hours for running an algorithm on a graph.

Datasets. We evaluate the algorithms on 17 real bipartite graphs, all of which are publicly available on KONECT .[5] Statistics of the graphs are shown in Table 1, where the graphs are listed in increasing order regarding the number of edges.

Query Parameters. A maximal similar-biclique enumeration query consists of two parameters, ε and τ. ε is chosen from $\{0.4, 0.5, 0.6, 0.7, 0.8\}$, and is set as 0.5 by default. τ is chosen from $\{3, 4, 5, 6, 7\}$, and is set as 3 by default. In addition, we also have parameters α and γ in index construction; we set $\alpha = 1$ and $\gamma = 0.3$ by default.

7.1 Evaluating query efficiency

In this subsection, we evaluate the query efficiency of our algorithms. Note that, we also implemented a version of MSBE without the optimizations of Enum proposed in

[5] http://konect.cc/networks/.

Table 1 Statistics of graphs

Abbreviation	Graph	$\lvert V_L \rvert$	$\lvert V_R \rvert$	$\lvert E \rvert$	Type
YT	YouTube	94,238	30,087	293,360	Membership
GH	GitHub	56,519	120,867	440,237	Membership
LX	Linux	42,045	337,509	599,858	Post
BS	Bibsonomy	767,447	5794	801,784	Assignment
BC	BookCross	105,278	340,523	1,149,739	Rating
AM	ActorMovie	127,823	383,640	1,470,404	Appearance
WU	WebUni	6202	200,148	1,948,004	Appearance
CU	CiteULike	731,769	153,277	2,338,554	Assignment
TV	TVTropes	64,415	87,678	3,232,134	HasFeature
IM	IMDB	303,617	896,302	3,782,463	Appearance
AZ	Amazon	1,879,572	1,162,941	4,955,492	Rating
DI	Discogs	1,754,823	270,771	5,302,276	Affiliation
FL	Flickr	395,979	103,631	8,545,307	Membership
DB	DBLP	1,953,085	5,624,219	12,282,059	Authorship
NY	NYTimes	299,752	101,636	69,679,427	Appearance
DE	Delicious	833,081	33,778,221	101,798,957	Interaction
OR	Orkut	2,783,196	8,730,857	327,037,487	Affiliation

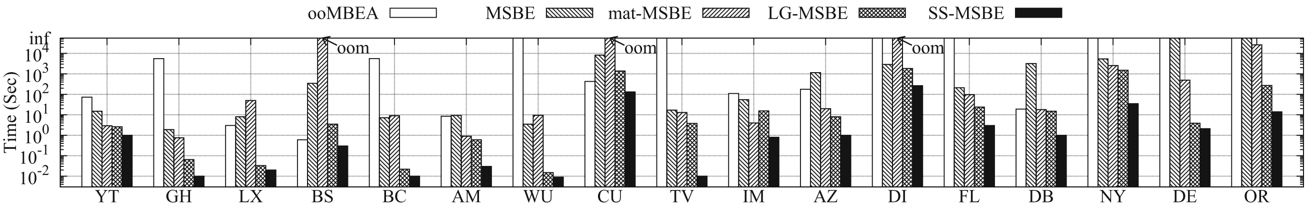
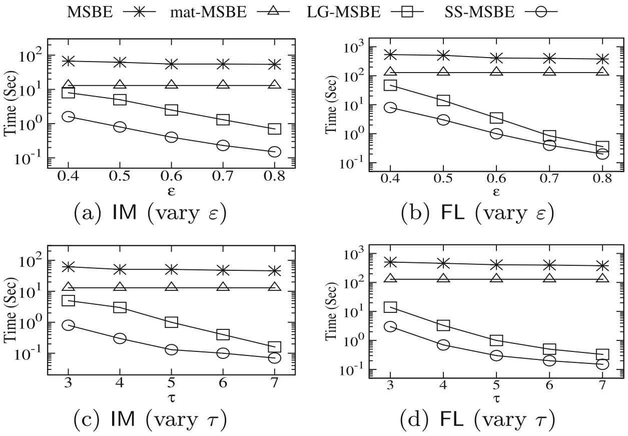


Fig. 8 Running time on all graphs ($\varepsilon = 0.5$, $\tau = 3$)

Sect. 4.2; it is omitted from the experiments since it times out in almost all the testings.

Running time on all graphs. The running time of the five algorithms on all graphs with default ε and τ is illustrated in Fig. 8. We can see that mat-MSBE slightly improves upon MSBE when it is feasible to store the similar neighbors of all vertices in main memory. However, mat-MSBE remains notably slow, primarily attributed to the substantial computational cost involved in computing similar neighbors for L-side vertices and constructing the similarity graph G_s. Additionally, mat-MSBE encounters out-of-memory issues on BS, CU, and DI, as marked by "oom" in Fig. 8; for example, the memory consumption on BS would be over 400GB. Note that the memory consumption of mat-MSBE mainly depends on the structure, rather than the size, of the input graph, and thus mat-MSBE does not run out-of-memory on other larger graphs. Our two index-based algorithms, LG-MSBE and SS-MSBE, are the fastest and they outperform the other two index-free algorithms by up to 5 orders of magnitude. SS-MSBE is generally faster than LG-MSBE. Compared with the state-of-the-art maximal biclique enumeration algorithm ooMBEA, SS-MSBE is up to 6 orders of magnitude faster. Thus, we exclude ooMBEA from our remaining evaluations.



(a) IM (vary ε)

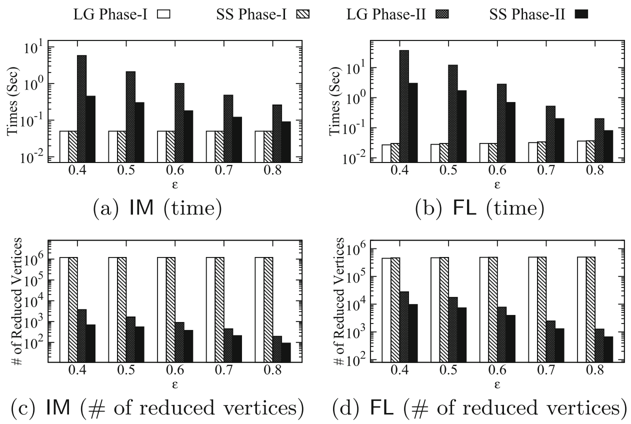
(b) FL (vary ε)

(c) IM (vary τ)

(d) FL (vary τ)

Fig. 9 Running time by varying ε and τ

Running time by varying ε and τ. The running time of our four algorithms on IM and FL by varying ε and τ are shown in Fig. 9. We can see that the running time of LG-MSBE and SS-MSBE decreases when either ε or τ increases. This is



(a) IM (time)

(b) FL (time)

(c) IM (# of reduced vertices)

(d) FL (# of reduced vertices)

Fig. 10 Efficiency of indexedVR ($\tau = 3$)

because, more vertices will be pruned by indexedVR when either ε or τ increases, and thus the enumeration process of LG-MSBE and SS-MSBE run faster. Also, indexedVR runs faster when ε or τ increases, as can be seen from Fig. 10. In contrast, the running time of MSBE and mat-MSBE is not so sensitive to ε or τ, as the dominating part of these two algorithms is computing similar neighbors for vertices.

Efficiency of indexedVR. In this experiment, we evaluate the efficiency of indexedVR for our two index structures. Recall that indexedVR (Algorithm 5) has two phases. Thus, we separately report the results of each phase. The running time on IM and FL are shown in Fig. 10a and b. We can see that the two index structures take almost the same time for the first phase, while the second phase of SS index-based indexedVR is much faster than LG index-based. This can partially be explained by the number of vertices that need to be pruned in the second phase, as reported in Fig. 10c and d. We remark that, for a fixed ε and τ, the total number of pruned vertices by different indexes are the same, and also the same as that pruned by the index-free approach VReduce. Thus, from the number of vertices that are pruned in Phase-II as shown in Fig. 10, we can conclude that SS index prunes much more vertices than LG index in Phase-I. For example, for dataset FL and $\varepsilon = 0.4$, SS index prunes 467, 329 vertices in Phase-I and 9, 691 vertices in Phase-II, while LG index prunes 449, 195 vertices in Phase-I and 27, 825 vertices in Phase-II. As the second phase dominates the running time, SS index is superior.

Evaluate the optimizations for Enum. In this experiment, we evaluate the performance of our two optimization techniques (i.e., Lemma 1 and Lemma 2) on the Enum procedure. The results are reported in Fig. 11. Specifically, we use our fastest algorithm SS-MSBE to evaluate these two optimization techniques. Note that SS-MSBE applies both optimizations. For this testing, we also implement SS-MSBE-v1 that adopts neither of the two optimizations, SS-MSBE-v2 that adopts only Lemma 1, and SS-MSBE-v3 that adopts only Lemma 2.

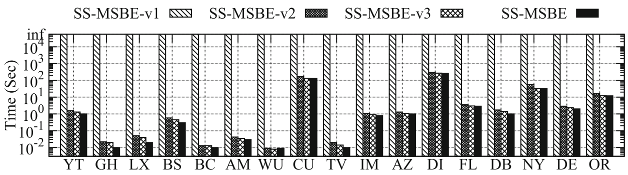


Fig. 11 Evaluation of the optimizations for Enum

From Fig. 11, we can see that SS-MSBE-v1 cannot finish in a reasonable time due to lacking of these optimization techniques, and both SS-MSBE-v2 and SS-MSBE-v3 run significantly faster than SS-MSBE-v1 demonstrating that both Lemma 1 and Lemma 2 significantly improved the performance. We can also observe that the improvements of SS-MSBE over SS-MSBE-v2 and over SS-MSBE-v3 are not that significant. This is because the additional improvements brought by Lemma 2 over Lemma 1 and brought by Lemma 1 over Lemma 2 are not that significant. Nevertheless, SS-MSBE consistently runs faster than SS-MSBE-v2 and SS-MSBE-v3; thus we apply both Lemmas in our algorithm.

7.2 Evaluating indexing techniques

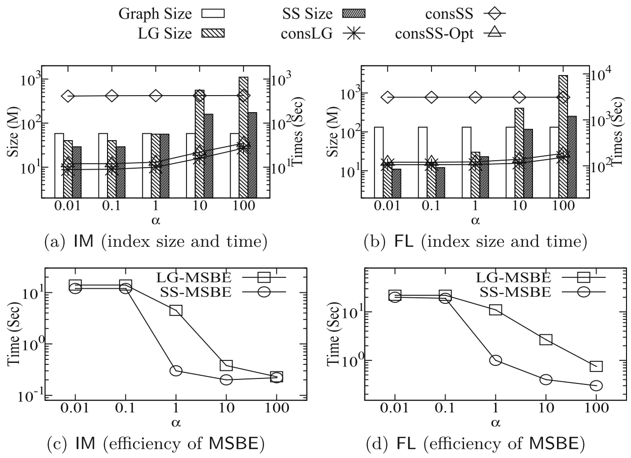
Index size and construction time on all graphs. The size of the two indexes on all graphs are shown in the fourth column and last column of Table 2. As a comparison, we also report the graph size in the second column of Table 2. We can see that in most cases, the sizes of the two indexes are similar to each other and are at the same level as the graph size, and thus they are affordable to be stored in main memory.

The running time of our index construction algorithms consLG, consSS, consSS* and consSS-Opt are reported in the third, fifth, sixth and seventh columns of Table 2, respectively. Here, consSS* uses a similarity tree to compute all the maximal steady segments at Line 3 of Algorithm 6; the details can be found in our preliminary version [52]. consSS-Opt invokes TPA (Algorithm 7) to compute all the maximal steady segments. We can see that consLG runs the fastest due to its simplicity. By applying our novel optimization technique, consSS-Opt is much faster than consSS and consSS* and is only slightly slower than consLG. We omit consSS* from all the remaining testings.

Index performance by varying α. In this experiment, we evaluate the effect of α on the index size, index construction time and efficiency of MSBE. The results are shown in Fig. 12. Recall that α controls the number of segments constructed for Φ_u. As expected, the index size and index construction time increase along with the increasing of α, as shown in Fig. 12a and b. When α is no larger than 1, the index size is at most at the same level as the graph size, but when α reaches 100, the index size can be much larger than the graph size. As shown in Fig. 12c and d, the running time of both LG-MSBE

Table 2 Index size and construction time

Graph	Size	Largest gap index		Steady segment index			
		consLG (s)	Size	consSS (s)	consSS* (s)	consSS-Opt (s)	Size
YT	4.6M	3	6.7M	21	8	6	8.3M
GH	7M	1.1	6.6M	90	4.9	1.5	6.2M
LX	9.2M	48	31M	9485	588	99	50M
BS	12.4M	506	75M	49,685	6975	805	59M
BC	18M	16	27M	1200	130	28	30M
AM	24M	1.5	24M	81	7	2.1	20M
WU	30M	11	15M	656	40	15	16M
CU	36M	1130	73M	296,094	12,547	1510	103M
TV	50M	11	4.4M	110	14	12	2.3M
IM	58M	10	56M	420	28	13	56M
AZ	76M	22	122M	2220	152	37	124M
DI	82M	549	132M	45,967	5187	863	146M
FL	132M	106	30M	3102	235	122	23M
DB	188M	29	299M	1905	177	45	324M
NY	1.1G	2623	35M	20,934	4397	2708	14M
DE	1.5G	3071	2.3G	129,304	13, 435	3910	3.1G
OR	5G	21,874	690M	246,045	23,872	23,164	459M



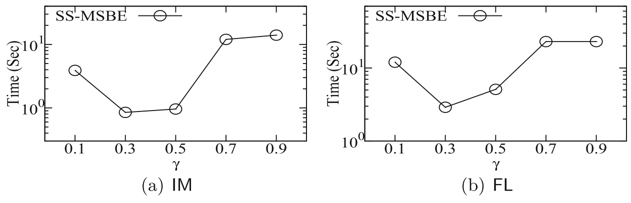
(a) IM (index size and time)

(b) FL (index size and time)

(c) IM (efficiency of MSBE)

(d) FL (efficiency of MSBE)

Fig. 12 Index performance by varying α



(a) IM

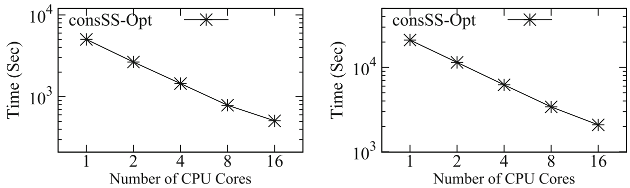
(b) FL

Fig. 13 Efficiency of SS-MSBE by varying γ

and SS-MSBE decreases when α increases. This is because the more the number of segments, the fewer the number of fake vertices. To strike a balance between index size and efficiency of MSBE, we recommend to set $\alpha \in [0.1, 10]$.

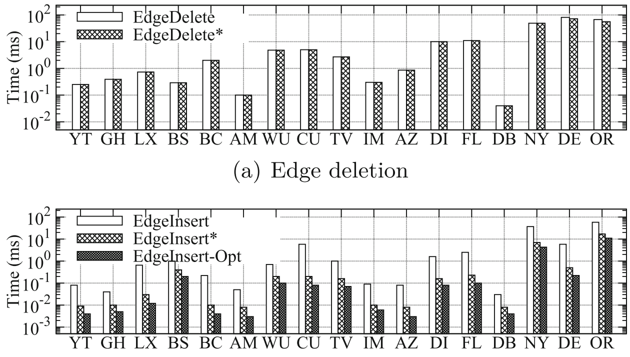
Efficiency of SS-MSBE**by varying** γ. In this experiment, we evaluate the performance of SS-MSBE for different γ values. Note that the index size and index construction time of consSS-Opt are almost not affected by γ; thus, we omit these results. This is because consSS-Opt selects a fixed number of steady segments (i.e., $\alpha \log |\Phi_u|$) to cover as many vertices of Φ_u as possible, and then, it covers all remaining uncovered vertices of Φ_u by using the fewest number of disjoint segments. Thus, the total number of segments generated for Φ_u is at most $2\alpha \log |\Phi_u| + 1$, which is independent of γ. Figure 13 shows the running time of SS-MSBE by varying γ from 0.1 to 0.9. We can see that when γ is small (e.g., $\gamma < 0.3$), the performance of SS-MSBE is not good. The main reason is that when γ is small, a steady segment will cover fewer vertices due to the tighter constraint. As a result, more vertices need to be covered by the ordinary segments, which then results in introducing more fake vertices. Also, when γ is large, the performance of SS-MSBE becomes worse. This is because for large γ (e.g., $\gamma = 1$), a steady segment is no longer steady and degenerates to the ordinary segment. This motivates us to introduce steady segment. We recommend the value of γ to be in [0.3, 0.5].

Parallel index construction. In this experiment, we evaluate the parallelized version of our index construction algorithm consSS-Opt by varying the number of CPU cores from 1 to 16 on our largest datasets DE and OR. This experiment was conducted on a machine with an Intel(R) 2.6GHz CPU and 16 cores. As shown in Fig. 14, consSS-Opt has a near-linear speedup by using multiple CPU cores. For example, on OR, the running time of consSS-Opt decreases from 21, 101 sec-



(a) DE (b) OR

Fig. 14 Speeding up consSS-Opt by using multiple CPU cores



(a) Edge deletion

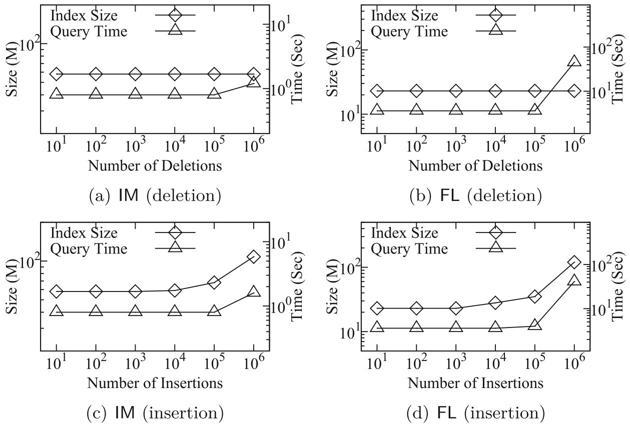
(b) Edge insertion

Fig. 15 Running time of index maintenance

onds (around 6 hours) to 2, 084 seconds (around 0.5 hour) when we increase the core number from 1 to 16. This demonstrates that the index construction can be easily parallelized, and our index can be constructed more efficiently with the help of multiple CPU cores which makes our index more attractive.

Efficiency of index maintenance. In this experiment, we evaluate the performance of our index maintenance algorithms. For edge deletion, we randomly delete 1, 000 edges from the graph one by one and report the average processing time. For comparison, we also implement a version of EdgeDelete* that does not performance the optimization (i.e., Lemma 7), denoted EdgeDelete. The average processing time of EdgeDelete* and EdgeDelete for handling edge deletions is reported in Fig. 15a. We can see that the running time of EdgeDelete* is at millisecond-level. For example, when deleting an edge from DB, the maintenance of the index can be completed in less than 0.1 ms. However, the effectiveness of optimization technique is not obvious, i.e., EdgeDelete* performs similarly to EdgeDelete. This mainly because the number of u's 2-hop neighbors that are adjacent to v is much smaller than the total number of u's 2-hop neighbors; thus, the pruning at Lines 6–7 of Algorithm 8 is not very effective.

For edge insertion, we randomly insert 1, 000 new edges into the graph one by one and report the average processing time. We compare EdgeInsert-Opt with EdgeInsert* and its non-optimized version EdgeInsert. The results are reported in Figure 16. We can see that EdgeInsert* runs significantly faster than EdgeInsert due to the optimization technique described in Lemma 9. Nevertheless, EdgeInsert-Opt runs the fastest, since it only processes the neighbors of u and v for the newly inserted edge (u, v).



(a) IM (deletion) (b) FL (deletion)

(c) IM (insertion) (d) FL (insertion)

Fig. 16 Index size and query efficiency w.r.t the number of updates

Index size and query efficiency w.r.t. the number of updates. In this experiment, we evaluate the index size and query efficiency w.r.t. the number of updates. In our index maintenance algorithms, for time efficiency consideration, we do not update a segment if one or more similarities in the segment decrease, and we also do not enforce the steady requirement of a segment. As a result, it is expected that the query efficiency will deteriorate when the index has been updated by a lot of edge insertions and deletions. The results for edge deletion on IM and FL are shown in Fig. 16a and b, respectively; here, the query time is the running time of SS-MSBE under the default setting (i.e., $\varepsilon = 0.5$ and $\tau = 3$). We can see that the index size remains steady; this is because deleting edges will not bring new segments to our index. However, the query time increases when the number of updates is over 10^5. The results for edge insertion on IM and FL are shown in Fig. 16c and d, respectively. We can see that both the index size and query time increase significantly when the number of inserted edges is over 10^5. This is because new segments are also created for handling edge insertions. From this experiment, we recommend to reconstruct the index from scratch when the number of updates has reached $0.1 \times |E|$.

7.3 Effectiveness evaluations

Average Jaccard similarity. We compare the average Jaccard similarity between L-side vertices in a maximal (similar-)biclique. Specifically, for each maximal (similar-)biclique C, we compute the average of the Jaccard similarity between all pairs of vertices from C_L, and then the average result of

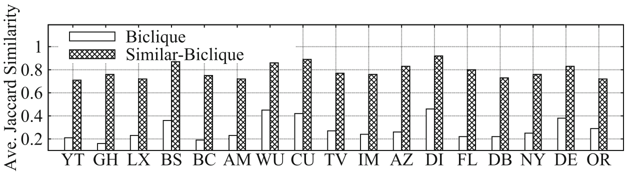
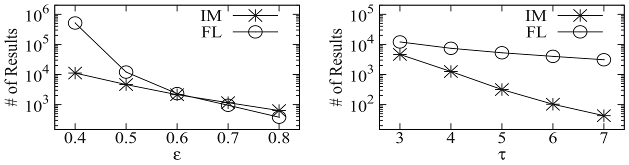


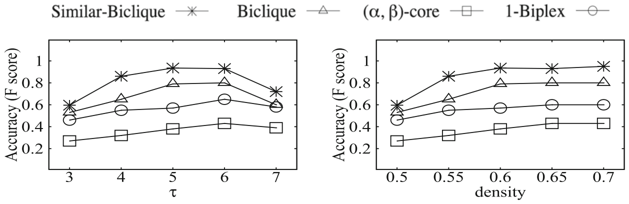
Fig. 17 Average Jaccard similarity



(a) Varying ε

(b) Varying τ

Fig. 18 Number of maximal similar-bicliques



(a) Varying τ

(b) Varying injected subgraph density

Fig. 19 Case study 1: anomaly detection

all maximal (similar-)bicliques is reported in Fig. 17. We can see that vertices in a similar-biclique are much more similar to each other than in a biclique.

Number of maximal similar-bicliques. The number of maximal similar-bicliques for different ε and τ values are shown in Fig. 18. We can see that the number of maximal similar-bicliques decreases with the increase of either ε or τ, which is as expected. This is because the number of similar neighbors for a vertex decreases with the increase of ε. Thus, more and more similar-bicliques disappear under a larger ε. It is worth mentioning that even under a high ε value of 0.7, there is still quite a few similar-bicliques, which exhibit a high level of consistence among members from the same side.

Case study 1: anomaly detection. In this case study, we compare similar-biclique with other dense bipartite subgraph models, biclique, (α, β)-core [24] and k-biplex [54], on anomaly detection in e-commerce applications. As mentioned in the Introduction, to improve the ranking of certain products, e-business owners may employ a set of fraudulent users to purchase a set of designated products. The fraudsters will also purchase other honest products trying to look "normal"; this is called "camouflage" in the literature. We consider a camouflage attack in the same way as [18] on "Amazon Review Data" (Magazine Subscriptions) ,[6] which contains 65, 546 reviews on 2, 316 magazines by 53, 617 users, by injecting 100 fraudulent users and 100 fraudulent products with various edge densities. The amount of camouflage (i.e., edges linking to honest products) added per fraudulent user is equal to the amount of fraudulent edges for that user. We adopt F-score, $\frac{2\times\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}$, to evaluate the accuracy of detecting suspicious users and products. We apply the size constraint τ to all the models, where $\alpha = \beta = \tau$ for the (α, β)-core model; for our similar-biclique model, ε is set as 0.2. The results by varying τ and varying the density of the injected subgraph are shown in Fig. 19. We can see that similar-biclique always achieves the highest accuracy. This is due to the similarity constraint imposed on users by similar-biclique, which naturally captures the reality that fraudulent users usually display a high level of synchronized behavior with each other. In contrast, biclique, 1-biplex, and (α, β)-core all have a low precision and thus low F-score.

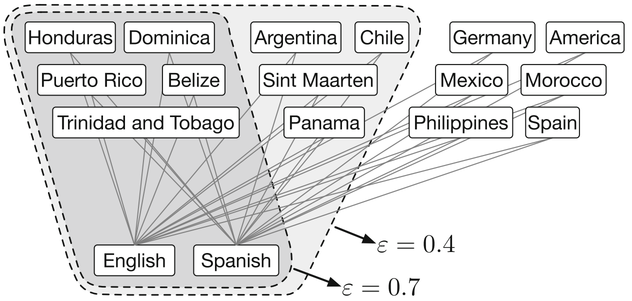


Fig. 20 Case study 2: similar-bicliques in Unicode($\tau = 2$)

Case study 2: interesting pattern detection on Unicode. We also conduct a case study on the Unicode dataset [25] to illustrate the hierarchical structure of similar-bicliques by varying the similarity threshold ε. Unicode captures the languages that are spoken in a country. The three similar-bicliques detected for $\varepsilon = 0.7$, 0.4, 0.01 are reported in Fig. 20, where the entire result corresponds to $\varepsilon = 0.01$; the similarity constraint is imposed on the countries and $\tau = 2$. We have the following observations. Firstly, the five countries in the similar-biclique for $\varepsilon = 0.7$ are all located in the Caribbean Sea Area with English and Spanish being their main language (around 90% population speak English and Spanish). Secondly, more countries from Latin America, e.g., Argentina and Chile, are included in the similar-biclique for $\varepsilon = 0.4$, and the newly added four countries speak more diverse languages. For example, in Sint Maarten, besides English and Spanish, around 8% population speak Virgin Islands Creole English and 4% population speak

[6] https://nijianmo.github.io/amazon/index.html.

Table 3 Case study 3: similar-bicliques in DBLP

Researchers	Research Groups
Albert Reuther, Andrew Prout, Antonio Rosa, Bill Bergeron, Chansup Byun, David Bestor, Julie Mullen, Matthew Hubbell, Peter Michaleas, William Arcand	High performance computing@MIT Lincoln Laboratory
Christian Menolfi, Lukas Kull, Marcel A. Kossel, Matthias Braendli, Pier Andrea Francese, Thomas Morf	CMOS integrated circuits@IBM Research-Zurich
Calvin Yu-Chian Chen, Chang-Hai Tsai, Chien-Yu Chen 0002, Da-Tian Bau, Fuu-Jen Tsai, Hung-Jin Huang, Ming-Hsui Tsai, Tin-Yun Ho, Yea-Huey Chang, Yuan-Man Hsu	Molecular biophysics @Taiwan
......	....

Dutch .[7] Lastly, when ε is 0.01, similar-biclique degenerates to biclique, and more countries are included, e.g., America and Germany. This demonstrates that similar-biclique can detect interesting patterns.

Case study 3: research group identification in DBLP. Our similar-biclique model also supports the case that not all vertices in a side share a common neighbor. In this case study, we show the similar-bicliques in a researcher-write-paper bipartite graph DBLP [8] by using different size constraints τ_L and τ_R on the two sides. The results for $\varepsilon = 0.6$, $\tau_L = 6$, and $\tau_R = 0$ are illustrated in Table 3; thus, researchers in a similar-biclique are not necessarily co-authors of the same paper. We find that the detected similar-bicliques corresponds to research groups in different institutes.

8 Conclusion

In this work, we formulated the notion of similar-biclique, and proposed algorithms as well as optimization techniques to enumerate all similar-bicliques in a bipartite graph. Besides, index structures are also designed to speed up the computation. We also proposed effective and efficient index construction algorithms by investigating two different strategies. In addition, we proposed index maintenance algorithms to handle dynamic graph updates. Extensive empirical studies on real bipartite graphs demonstrated the effectiveness of our similar-biclique model and the efficiency of our algorithms. Case studies show that the similar-biclique model can be used to detect anomalies as well as interesting dense subgraph patterns. Our work initiates the study of integrating similarity constraint into dense bipartite subgraph mining, by taking the biclique model. For future studies, it will be interesting to integrate similarity constraint into other dense bipartite subgraph models, such as quasi-biclique, k-biplex, (α, β)-core, k-bitruss, and k-wing. We believe that our proposed index structures will also be useful for these extensions.

Acknowledgements This work was partially supported by the Australian Research Council Fundings of FT180100256 and DP220103731 and the Research Grants Council of Hong Kong, China, under No. 14202919 and No. 14205520.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

References

1. Abidi, A., Zhou, R., Chen, L., Liu, C.: Pivot-based maximal biclique enumeration. In: IJCAI, pp. 3558–3564 (2020)
2. Adamic, L.A., Adar, E.: Friends and neighbors on the web. Soc. Netw. **25**(3), 211–230 (2003)
3. Alexe, G., Alexe, S., Crama, Y., Foldes, S., Hammer, P.L., Simeone, B.: Consensus algorithms for the generation of all maximal bicliques. Discret. Appl. Math. **145**(1), 11–21 (2004)
4. Allahbakhsh, M., Ignjatovic, A., Benatallah, B., Beheshti, S.-M.-R., Bertino, E., Foo, N.: Collusion detection in online rating systems. In: Asia-Pacific Web Conference, pp. 196–207. Springer (2013)
5. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Commun. ACM **16**(9), 575–577 (1973)
6. Chang, L., Wei Li, L., Qin, W.Z., Yang, S.: pscan: fast and exact structural graph clustering. IEEE Trans. Knowl. Data Eng. **29**(2), 387–401 (2017)
7. Chang, L., Yu, J.X., Qin, L.: Fast maximal cliques enumeration in sparse graphs. Algorithmica **66**(1), 173–186 (2013)
8. Chen, L., Liu, C., Zhou, R., Jiajie, X., Li, J.: Efficient maximal biclique enumeration for large sparse bipartite graphs. Proc. VLDB Endow. **15**(8), 1559–1571 (2022)
9. Cheng, J., Zhu, L., Ke, Y., Chu, S.: Fast algorithms for maximal clique enumeration with limited memory. In: Proceedings of KDD'12, pp. 1240–1248 (2012)

[7] https://www.unicode.org/cldr/cldr-aux/charts/25/summary/root.html.

[8] https://dblp.uni-trier.de/xml/.

10. Dave, V., Guha, S., Zhang, Y.: Viceroi: catching click-spam in search ad networks. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 765–776 (2013)
11. Dice, L.R.: Measures of the amount of ecologic association between species. Ecology **26**(3), 297–302 (1945)
12. Ding, D., Li, H., Huang, Z., Mamoulis, N.: Efficient fault-tolerant group recommendation using alpha-beta-core. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pp. 2047–2050 (2017)
13. El Bacha, R.L, Zin, T.T.: Ranking of influential users based on user-tweet bipartite graph. In: 2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI), pp. 97–101. IEEE (2018)
14. Eppstein, D.: Arboricity and bipartite subgraph listing algorithms. Inf. Process. Lett. **51**(4), 207–211 (1994)
15. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in sparse graphs in near-optimal time. In: International Symposium on Algorithms and Computation, pp. 403–414. Springer (2010)
16. Gangireddy, S.C.R., Long, C., Chakraborty, T.: Unsupervised fake news detection: a graph-based approach. In: Proceedings of the 31st ACM Conference on Hypertext and Social Media, pp. 75–83 (2020)
17. Hochbaum, D.S.: Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In: Approximation Algorithms for NP-Hard Problems, pp. 94–143 (1996)
18. Hooi, B., Song, H.A., Beutel, A., Shah, N., Shin, K., Faloutsos, C.: Fraudar: bounding graph fraud in the face of camouflage. In: Proceedings of KDD'16 (2016)
19. Jaccard, P.: Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. Bull. Soc. Vaudoise Sci. Nat. **37**, 241–272 (1901)
20. Jeh, G., Widom, J.: Simrank: a measure of structural-context similarity. In: Proceedings of KDD'02, pp. 538–543 (2002)
21. Jiang, M., Cui, P., Beutel, A., Faloutsos, C., Yang, S.: Catchsync: catching synchronized behavior in large directed graphs. In: Proceedings of KDD'14, pp. 941–950 (2014)
22. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953)
23. Kloster, K., Sullivan, B.D., van der Poel, A.: Mining maximal induced bicliques using odd cycle transversals. In: Proceedings of the 2019 SIAM International Conference on Data Mining, SDM 2019, Calgary, Alberta, Canada, May 2–4, 2019, pp. 324–332 (2019)
24. Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: Trawling the web for emerging cyber-communities. Comput. Netw. **31**(11–16), 1481–1493 (1999)
25. Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1343–1350 (2013)
26. Kuznetsov, S.O.: On computing the size of a lattice and related decision problems. Order **18**(4), 313–321 (2001)
27. Lehmann, S., Schwartz, M.: Biclique communities. Phys. Rev. E **78**(1), 016108 (2008)
28. Leicht, E.A., Holme, P., Newman, M.E.J.: Vertex similarity in networks. Phys. Rev. E **73**(2), 026120 (2006)
29. Ley, M.: The dblp computer science bibliography: Evolution, research issues, perspectives. In: International Symposium on String Processing and Information Retrieval, pp. 1–10. Springer (2002)
30. Li, J., Li, H., Soh, D., Wong, L.: A correspondence between maximal complete bipartite subgraphs and closed patterns. In: European Conference on Principles of Data Mining and Knowledge Discovery, pp. 146–156. Springer (2005)
31. Li, J., Liu, G., Li, H., Wong, L.: Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: a one-to-one correspondence and mining algorithms. IEEE Trans. Knowl. Data Eng. **19**(12), 1625–1637 (2007)
32. Liben-Nowell, D., Kleinberg, J.: The link-prediction problem for social networks. J. Am. Soc. Inform. Sci. Technol. **58**(7), 1019–1031 (2007)
33. Lin, Z., Lyu, M.R., King, I.: Matchsim: a novel similarity measure based on maximum neighborhood matching. Knowl. Inf. Syst. **32**(1), 141–166 (2012)
34. Liu, G., Sim, K., Li, J.: Efficient mining of large maximal bicliques. In: International Conference on Data Warehousing and Knowledge Discovery, pp. 437–448. Springer (2006)
35. Liu, X., Li, J., Wang, L.: Quasi-bicliques: Complexity and binding pairs. In: International Computing and Combinatorics Conference, pp. 255–264. Springer (2008)
36. Lü, L., Zhou, T.: Link prediction in complex networks: a survey. Phys. A **390**(6), 1150–1170 (2011)
37. Lyu, B., Qin, L., Lin, X., Zhang, Y., Qian, Z., Zhou, J.: Maximum biclique search at billion scale. In: Proceedings of the VLDB Endowment (2020)
38. Makino, K., Uno, T.: New algorithms for enumerating all maximal cliques. In: Scandinavian Workshop on Algorithm Theory, pp. 260–272. Springer (2004)
39. Martínez, V., Berzal, F., Cubero, J.-C.: A survey of link prediction in complex networks. ACM Comput. Surv. (CSUR) **49**(4), 1–33 (2016)
40. Megiddo, N., Zemel, E., Hakimi, S.L.: The maximum coverage location problem. SIAM J. Algebr. Discrete Methods **4**(2), 253–261 (1983)
41. Peeters, R.: The maximum edge biclique problem is np-complete. Discret. Appl. Math. **131**(3), 651–654 (2003)
42. Salton, G.: Automatic Text Processing: the Transformation, Analysis, and Retrieval of, p. 169. Addison-Wesley, Reading (1989)
43. Sanderson, M.J., Driskell, A.C., Ree, R.H., Eulenstein, O., Langley, S.: Obtaining maximal concatenated phylogenetic data sets from large sequence databases. Mol. Biol. Evolut **20**(7), 1036–1042 (2003)
44. Sarıyüce, A.E., Pinar, A.: Peeling bipartite networks for dense subgraph discovery. In: Proceedings of WSDM'18, pp. 504–512 (2018)
45. Satuluri, V., Parthasarathy, S., Ruan, Y.: Local graph sparsification for scalable clustering. In: Proceedings of SIGMOD'11, pp. 721–732 (2011)
46. Su, X., Khoshgoftaar, T.M: A survey of collaborative filtering techniques. In: Advances in Artificial Intelligence, 2009 (2009)
47. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theoret. Comput. Sci. **363**(1), 28–42 (2006)
48. Tseng, T., Dhulipala, L., Shun, J.: Parallel index-based structural graph clustering and its approximation. In: Proceedings of SIGMOD'21, pp. 1851–1864 (2021)
49. Uno, T., Kiyomi, M., Arimura, H., et al.: Lcm ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets. In: Fimi, vol. 126 (2004)
50. Wang, J., De Vries, A.P., Reinders, M.J.T.: Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 501–508 (2006)
51. Wang, X., Liu, J.: A comparative study of the measures for evaluating community structure in bipartite networks. Inf. Sci. **448**, 249–262 (2018)
52. Yao, K., Chang, L., Yu, J.X.: Identifying similar-bicliques in bipartite graphs. Proc. VLDB Endow. **15**(11), 3085–3097 (2022)

53. Yoon, S.-H., Kim, S.-W., Park, S.: C-rank: a link-based similarity measure for scientific literature databases. Inf. Sci. **326**, 25–40 (2016)
54. Yu, K., Long, C., Deepak, P., Chakraborty, T.: On efficient large maximal biplex discovery. IEEE Trans. Knowl. Data Eng. (2021)
55. Zaki, M.J., Hsiao, C.-J.: Charm: an efficient algorithm for closed itemset mining. In: Proceedings of the 2002 SIAM International Conference on Data Mining, pp. 457–473. SIAM (2002)
56. Zhang, Y., Phillips, C.A., Rogers, G.L., Baker, E.J., Chesler, E.J., Langston, M.A.: On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. BMC Bioinf. **15**(1), 1–18 (2014)
57. Zhang, Z.-Y., Ahn, Y.-Y.: Community detection in bipartite networks using weighted symmetric binary matrix factorization. Int. J. Mod. Phys. C **26**(09), 1550096 (2015)
58. Zhao, P., Han, J., Sun, Y.: P-rank: a comprehensive structural similarity measure over information networks. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, pp. 553–562 (2009)
59. Zou, Z.: Bitruss decomposition of bipartite graphs. In: International Conference on Database Systems for Advanced Applications, pp. 218–233. Springer (2016)