BMC Bioinformatics

RESEARCH

Open Access

Mining contextually meaningful subgraphs from a vertex-attributed graph

Riyad Hakim[1] and Saeed Salem[2*]

*Correspondence:
saeed.salem@qu.edu.qa

[1] Computer Science, North Dakota State University, Fargo, North Dakota, USA
[2] Computer Science and Engineering, Qatar University, Doha, Qatar

Abstract

Networks have emerged as a natural data structure to represent relations among entities. Proteins interact to carry out cellular functions and protein-Protein interaction network analysis has been employed for understanding the cellular machinery. Advances in genomics technologies enabled the collection of large data that annotate proteins in interaction networks. Integrative analysis of interaction networks with gene expression and annotations enables the discovery of context-specific complexes and improves the identification of functional modules and pathways. Extracting subnetworks whose vertices are connected and have high attribute similarity have applications in diverse domains. We present an enumeration approach for mining sets of connected and cohesive subgraphs, where vertices in the subgraphs have similar attribute profile. Due to the large number of cohesive connected subgraphs and to overcome the overlap among these subgraphs, we propose an algorithm for enumerating a set of representative subgraphs, the set of all closed subgraphs. We propose pruning strategies for efficiently enumerating the search tree without missing any pattern or reporting duplicate subgraphs. On a real protein-protein interaction network with attributes representing the dysregulation profile of genes in multiple cancers, we mine closed cohesive connected subnetworks and show their biological significance. Moreover, we conduct a runtime comparison with existing algorithms to show the efficiency of our proposed algorithm.

Keywords: Attributed graph, Subgraph enumeration, Cohesive subgraph, Minimum support, Maximal subgraph, Closed subgraph

Introduction

A graph is a mathematical representation of a real-world network. In a graph, we have a set of vertices representing objects and a set of edges representing the relationships between objects in the network. A protein-protein interaction (PPI) network captures the physical interactions between proteins in the cell. Studying PPI networks facilitates understanding complex biological processes. Several approaches have been proposed for mining functional modules from the topological interactions in the network [1].

Advances in high-throughput genomics enabled the collection of gene expression of thousands of genes under various biological and environmental conditions. These gene expression profiles annotate the proteins in the interaction network.

Integrating gene expression profiles with PPI network analysis enhances the ability to understand complex biological systems and enables the discovery of context-specific complexes [1]. For example, in biological network analysis, scientists may be interested in finding sets of proteins (or genes) that are connected as well as being differentially expressed in the same phenotypes [2]. The integration of differential gene expression profile while mining connected PPI subnetworks improves the identification of functional modules [3, 4], therapeutic targets and subnetwork biomarkers [1, 5], and active subnetworks [1, 6].

Even though proteins can have real-valued attributes, in this work, we consider attributes as characteristics that can be either associated or not associated with an object. We represent each attribute with a unique name, which is referred to as an *attribute-term*. In biological networks, a protein (or a gene) can be dysregulated in a few specific diseases, and we can annotate a protein in the PPI network with the names of diseases in which it is dysregulated. We refer to the undirected graph where each vertex has some attribute-terms associated with its vertices as a *vertex-attributed (VA) graph*.

A vertex set (object set) in a VA graph is called *cohesive* if the vertices (objects) in the set share a large number of attribute-terms. The *support* of a vertex set is the number of attribute-terms that are shared by all the vertices in that set. The cohesiveness property of a vertex set is an anti-monotone constraint because if a vertex set is cohesive, all its subsets are also cohesive. Anti-monotone constraints are useful for reducing the search space since pruning the futile branches of the search space becomes easier. If a set violates an anti-monotone constraint, we can assert without any further checking that all its supersets will also violate the same constraint.

However, any anti-monotone constraint based search usually yields a high number of sets with significant redundancy. In case of mining cohesive vertex sets, we can eliminate such redundancy by reporting a cohesive vertex set only if it is not a proper subset of another cohesive vertex set; these cohesive sets are called *maximal* cohesive vertex sets. However, if we do so, we lose some information as we do not report the proper subsets (of maximal sets) that might have larger support than their corresponding maximal sets and could be interesting. We can also adopt a different approach of eliminating redundancy by reporting a cohesive vertex set only if it is not a proper subset of another cohesive vertex set that has the same support; these cohesive sets are called *closed* cohesive vertex sets. Unlike maximal cohesive sets, we can generate all cohesive sets from the set of all closed cohesive sets.

In 2006, Wernicke [7] introduced the ESU algorithm for enumerating the set of all connected vertex sets in an undirected graph. The ESU proposed a graph traversal method that avoids visiting the same subgraph multiple times.

In 2015, Elbassioni [8] and Uno [9] introduced two separate algorithms for enumerating the set of all connected vertex sets of a graph. The algorithm by Elbassioni [8] takes polynomial time between two consecutive outputs (i.e., it has polynomial delay), whereas the algorithm of Uno (called TGE) [9] takes $\mathcal{O}(1)$ amortized time per output. Both algorithms use linear space [8, 9].

In 2014, Maxwell et al. [2] introduced an algorithm (called BDDE) for enumerating the set of all maximal connected vertex sets in an undirected graph with respect to any given anti-monotone constraint. However, the main limitation of BDDE is that it requires exponential space [2]. In 2019, Alokshiya et al. [10] introduced a linear delay algorithm (called RSSP) for enumerating the set of all connected vertex sets in an undirected graph. The RSSP algorithm employed a reverse search strategy for enumerating the set of all connected vertex sets without duplicates. Utilizing RSSP as the core enumeration process, Alokshiya et al. [10] also introduced the RSSP-maximal algorithm for enumerating the set of all maximal cohesive connected vertex sets in a VA graph.

In 2020, we introduced the Miner algorithm for enumerating the set of all connected vertex sets in an undirected graph [11]. The Miner algorithm takes linear time between two consecutive outputs (i.e., it has linear delay) and uses linear space in number of vertices in the graph [11]. Experimental results showed that Miner was faster than both RSSP and TGE algorithms [11]. In the same paper [11], we extended the Miner algorithm and introduced the **C**ohesive **S**ubgraph **Miner** algorithm (CSMiner-maximal) for enumerating the set of all maximal cohesive connected vertex sets in a VA graph.

In this paper, we extend the Miner algorithm and propose the CSMiner-closed algorithm for enumerating the set of all closed cohesive connected vertex sets in a vertex-attributed graph. We provide a proof of correctness and a proof of complexity of the Miner algorithm (which were not present in [11]). Further, we perform comparative runtime analysis to show the efficiency of our proposed approach and the effectiveness of the proposed pruning strategies. We demonstrate the biological significance of the closed cohesive connected subnetworks mined by our algorithm from a protein-protein interaction network whose vertices are associated with disease dysregulation profiles.

The rest of this paper is organized as follows. In section 2, we provide a detailed explanation of our approach for enumerating the set of all closed cohesive connected vertex sets. Next, in section 3, we provide a runtime comparison on a human PPI network with disease phenotypes as attribute-terms. We also provide biological enrichment analysis of the gene sets extracted by our algorithm from the same protein-protein interaction network. Finally, in section 4, we conclude with a summary of the proposed work and directions for future work.

Method

An undirected graph G is a structure of a set of vertices and a set of connections between some pairs of vertices. The graph G is represented by a tuple (V, E), where V is the set of vertices and $E \subseteq \{\{u, v\} : u, v \in V\}$ is the set of edges (i.e., unordered pairs of vertices). We assume that there is a one-to-one function $r : V \to \mathbb{N}$ that maps each vertex $v \in V$ to a unique non-negative integer, called *rank* of the vertex. We write '$u \prec v$' to denote that the vertex $u \in V$ has lower rank than the vertex $v \in V$, i.e., $r(u) < r(v)$.

Algorithm 1 Miner

Input : $G = (V, E)$: an undirected graph
Output: $\mathcal{CVS}_G$

```
/* 0-based list indexing assumed */
foreach α ∈ V do
    EXTEND({}, [α], α, 0 + 1)
Function EXTEND(V_cvs, V_ext, v, start)
    V_cvs ← V_cvs ∪ {v}
    V_ext ← V_ext || (N(v)\V_ext)
    output V_cvs
    foreach x_i ∈ V_ext, with i ≥ start do
        if x_i > α then
            EXTEND(V_cvs, V_ext, x_i, i + 1)
```

The *neighbor set* of a vertex $v \in V$, denoted $\mathrm{N}(v)$, is the set of all vertices in V that are adjacent to v.

$$\mathrm{N}(v) = \{u \in V : \{u, v\} \in E\} \tag{1}$$

The *open neighborhood* of a vertex set $S \subseteq V$, denoted $N_{op}(S)$, is the set of all vertices that are not in S and are adjacent to at least one vertex in S [7].

$$N_{op}(S) = \{u \in V \setminus S : \exists v \in S \textit{ such that } u \in \mathrm{N}(v)\} \tag{2}$$

Given a graph $G = (V, E)$, the *induced subgraph* of a vertex set $S \subseteq V$, denoted $G[S]$, is the graph with set of vertices S and set of edges E_S, where E_S consists of all the edges in E that have both endpoints in S.

$$\begin{aligned} G[S] &= (S, E_S) \\ E_S &= \{\{u, v\} \in E : u, v \in S\} \end{aligned} \tag{3}$$

Since there is a one-to-one mapping between a vertex set $S \subseteq V$ and its induced subgraph $G[S]$, we use the two terms interchangeably. A vertex set $S \subseteq V$ is called a *connected vertex set* if and only if its induced subgraph $G[S]$ is connected.

We define the *concatenation* of a list X_1 and a set X_2, denoted $X_1 \,||\, X_2$, as a new list consisting of all elements in X_1 (maintaining the same order) followed by all elements in X_2. We assume that the starting index of any list-like data structure is 0.

Mining connected vertex sets

Given an undirected graph $G = (V, E)$, the set of all connected vertex sets in G:

$$\mathcal{CVS}_G = \{S \subseteq V : G[S] \textit{ is connected}\} \tag{4}$$

Problem Definition 1 Given an undirected graph $G = (V, E)$, enumerate the set of all connected vertex sets $\mathcal{CVS}_G$.

Algorithm 1 shows the pseudo-code of the Miner algorithm [11] that enumerates the set of all connected vertex sets in an undirected graph $G = (V, E)$. In line 2, with a call to the function EXTEND, the algorithm starts an enumeration process from a single vertex $\alpha \in V$, called *anchor vertex*, and the process completes by enumerating every connected vertex set in G that contains α as the vertex with the lowest rank among all the vertices in that set (see Lemma 1). The algorithm selects each vertex in the graph as an anchor vertex and restarts the enumeration process from that vertex (lines 1-2).

Due to the recursive nature of the algorithm, the algorithm is best understood if we associate a tree T_α with the enumeration process started from an anchor vertex α. As each vertex in the graph is selected as an anchor vertex, the algorithm generates a forest $\mathcal{F} = \{T_\alpha : \alpha \in V\}$ consisting of $|V|$ trees. In order to distinguish between the vertices of input graph G and the vertices of enumeration tree T_α, we refer to the vertices of graph G as vertices and the vertices of tree T_α as nodes. Each node (except the root node) in the enumeration tree represents a connected vertex set. Each node of T_α has three components: V_{cvs}, V_{ext}, and *start*. The vertex set V_{cvs} contains the vertices that constitute the connected vertex set represented by the node and the list V_{ext} has the union of neighbor set of each vertex in V_{cvs}. The *start* index is a variable pointing to an element in the list V_{ext}.

Node generation principle

1. *Root and anchor nodes*: The root node of a tree T_α is a node with $V_{cvs} = \{\}$, $V_{ext} = [\alpha]$, and $start = 0$, i.e., $V_{ext}[start] = \alpha$. In line 2, by calling the function EXTEND, the root node generates its only child node (called *anchor node* of T_α) with $V_{cvs} = \{\alpha\}$, $V_{ext} = [\alpha] \parallel (\mathrm{N}(\alpha) \setminus [\alpha])$, and $start = 1$.
2. *Generating child*: Let us consider a node in T_α with $V_{cvs} = S \neq \{\}$, $V_{ext} = X$, $start = k$. For brevity, we refer to a node with $V_{cvs} = S$ as simply '*a node S*'. In line 9, by calling the function EXTEND for each vertex $x_i = X[i]$ with $i >= k$ and $x_i > \alpha$, the node S generates a child node $S \cup \{x_i\}$. Moreover, the child node's V_{ext} is expanded to include the neighbors of the extending vertex x_i that are not already in its parent node's V_{ext}, i.e., the child node's $V_{ext} = X \parallel (\mathrm{N}(x_i) \setminus X)$, and the child node's starting index *start* is updated to point to the vertex immediately after the extending vertex x_i in its V_{ext}, i.e., the child node's $start = i + 1$.

In the path from root node to a descendant node in a tree, every time a child is generated by extending a node with a vertex v, the child node's V_{ext} is constructed by the concatenation of its parent's V_{ext} and neighbor set of v (excluding vertices already in parent's V_{ext}). Since V_{ext} of a node consists of neighbors of each vertex in its V_{cvs}, the open neighborhood of V_{cvs} of any non-root node can be obtained from its V_{ext} as follows:

$$N_{op}(V_{cvs}) = V_{ext} \setminus V_{cvs} \tag{5}$$

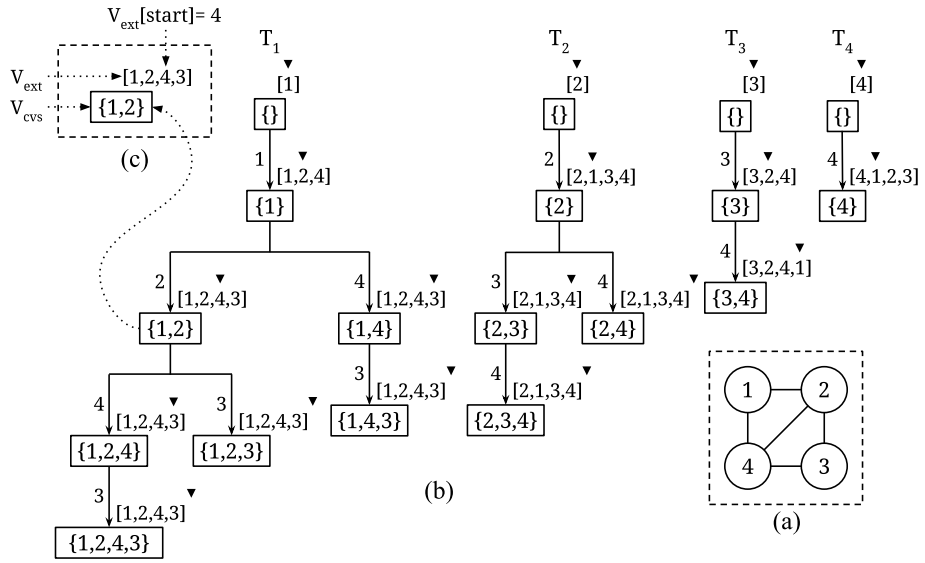

Fig. 1 Enumeration of set of all connected vertex sets in a sample graph. (**a**) A sample undirected graph. Each vertex is labeled by its rank. (**b**) Enumeration forest $\{T_1, T_2, T_3, T_4\}$ generated by the Miner algorithm for the sample graph. (**c**) Components of a node in a tree

In Fig. 1, (b) shows the complete forest for the sample graph in (a).

Proof of correctness

To prove that the algorithm is correct, first we show that the algorithm outputs all connected vertex sets in the given graph (Lemma 1). Then we show that the algorithm does not output the same connected vertex set more than once (Lemma 2 and Lemma 3).

Let us introduce some notations. The number of edges in the shortest path between two vertices in a graph is called the *distance* between the vertices. Let $S_D = \{v_0, \dots\} \subseteq V$ be a connected vertex set, where v_0 has the lowest rank among all the vertices in S_D (i.e., $v \in S_D \implies v \geq v_0$), and D is the distance of the most distant vertices from v_0 in $G[S_D]$. We refer to a vertex in S_D with distance d ($0 \leq d \leq D$) from v_0 in $G[S_D]$ as a distance-d vertex. So, in S_D, there are one distance-0 vertex (which is the vertex v_0), one or more distance-1 vertices, one or more distance-2 vertices, and so on up to distance-D vertices. Let m_d denote the number of distance-d vertices in S_D. As there can be only one distance-0 vertex in S_D, $m_0 = 1$. For $d > 0$, we use the placeholder notation $\Pi_{i=1}^{m_d} v_d^{(i)}$ to represent a combination of all distance-d vertices $v_d^{(1)}, v_d^{(2)}, \dots, v_d^{(m_d)}$ in S_D without associating any order among them. Hence, after sorting the vertices in S_D in ascending order of distance from v_0 in $G[S_D]$, we get $S_D = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \dots, \Pi_{i=1}^{m_D} v_D^{(i)}\}$. Please note that any connected vertex set in the input graph G can be represented in this format.

Lemma 1 *Tree T_{v_0} has a node $S_D = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \dots, \Pi_{i=1}^{m_D} v_D^{(i)}\}$.*

Proof For $D = 0$, $S_0 = \{v_0\}$, which is the anchor node of T_{v_0} (base step).

For $D > 0$, we assume that the nodes $S_0 = \{v_0\}$, $S_1 = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}\}$, $S_2 = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \Pi_{i=1}^{m_2} v_2^{(i)}\}$, $\dots$, $S_k = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \dots, \Pi_{i=1}^{m_k} v_k^{(i)}\}$ with $k < D$, are

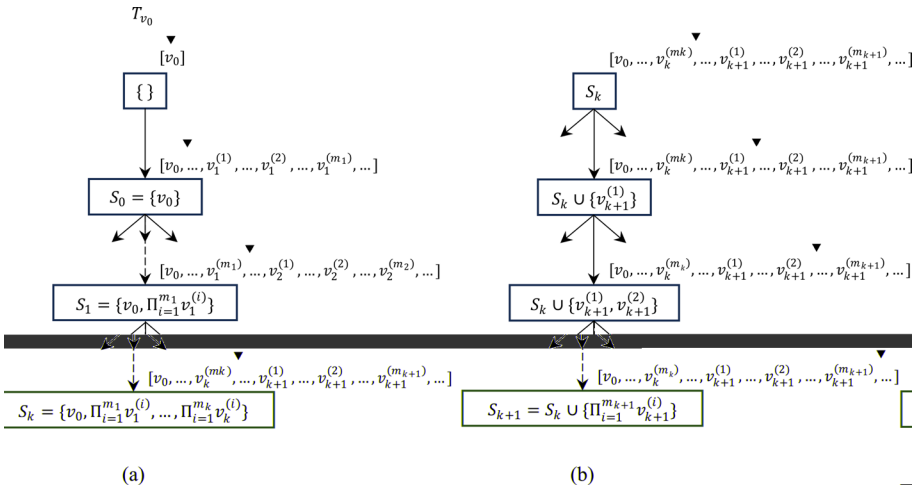

Fig. 2 Proof of Lemma 1 illustration. (**a**) Inductive hypothesis. (**b**) Inductive step. A solid arrow indicates a child, whereas a dotted arrow indicates a descendant

generated in T_{v_0} in such an order that, for each $0 \leq i < k$, node S_{i+1} is a descendant of node S_i (see Fig. 2). Based on this assumption (i.e., our inductive hypothesis), we show that the node $S_k = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \ldots, \Pi_{i=1}^{m_k} v_k^{(i)}\}$ has a descendant $S_{k+1} = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \ldots, \Pi_{i=1}^{m_k} v_k^{(i)}, \Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}\}$ in T_{v_0} (inductive step).

In the anchor node $S_0 = \{v_0\}$, V_{ext} is the concatenation of the list $[v_0]$ and neighbor set of v_0, and $start = 1$, i.e., *start* points to the vertex immediately after the extending vertex v_0 in V_{ext}. So, each of the vertices $\Pi_{i=1}^{m_1} v_1^{(i)}$ must be located at or after the *start* index in V_{ext} of node S_0 (see node S_0 in Fig. 2). Similarly, in the path from node $S_0 = \{v_0\}$ to its descendant $S_1 = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}\}$, every time a child is generated by extending with a vertex $v_1^{(i)}$ $(1 \leq i \leq m_1)$, its V_{ext} becomes the concatenation of its parent's V_{ext} and neighbor set of $v_1^{(i)}$ (excluding vertices already in parent's V_{ext}), and *start* points to the vertex immediately after the extending vertex $v_1^{(i)}$ in V_{ext}. So, each of the vertices $\Pi_{i=1}^{m_2} v_2^{(i)}$ must be located at or after the *start* index in V_{ext} of node S_1 (see node S_1 in Fig. 2). By applying the same argument for node $S_k = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \ldots, \Pi_{i=1}^{m_k} v_k^{(i)}\}$, we can say that each of the vertices $\Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}$ must be located at or after the *start* index in V_{ext} of node S_k (see node S_k in Fig. 2).

Let us assume that the vertices $\Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}$ appear in the order $v_{k+1}^{(1)}, v_{k+1}^{(2)}, \ldots, v_{k+1}^{(m_{k+1})}$ at or after the *start* index in V_{ext} of node S_k. So, $v_{k+1}^{(1)}$ must be selected at some point in line 7. Since $v_{k+1}^{(1)} > v_0$ (as v_0 has the lowest rank in S_D), the function EXTEND is called in line 9, and the node S_k generates the child $S_k \cup \{v_{k+1}^{(1)}\}$. Similarly, the node $S_k \cup \{v_{k+1}^{(1)}\}$ generates the child $S_k \cup \{v_{k+1}^{(1)}, v_{k+1}^{(2)}\}$, and finally, the node $S_k \cup \{\Pi_{i=1}^{m_{k+1}-1} v_{k+1}^{(i)}\}$ generates the child $S_{k+1} = S_k \cup \{\Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}\}$. Note that, if the order of the vertices $\Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}$ in V_{ext} of node S_k were different, the node S_k would still generate the descendant S_{k+1}. However,

the vertices $\Pi_{i=1}^{m_{k+1}} v_{k+1}^{(i)}$ would have been added in a different order, which does not matter. □

Lemma 2 *No two nodes in a tree have the same V_{cvs}.*

Proof (Two nodes on the same branch) As there is never any duplicate vertex in V_{ext} of a node and the *start* index always points to the vertex immediately after the last extending vertex in V_{ext}, the V_{cvs} of a node must contain one additional vertex than that of its parent. So, the V_{cvs} of a node is always larger (and hence different) than that of each of its ancestors by at least one vertex.

(Two nodes on different branches) In a tree, let $S \cup S_1$ and $S \cup S_2$ be two nodes with the node S being their earliest common ancestor. Let x and y be two vertices with x located before y in V_{ext} of node S such that the node $S \cup \{x\}$ is a child of node S on the path from node S to node $S \cup S_1$ and the node $S \cup \{y\}$ is a child of node S on the path from node S to node $S \cup S_2$. Since the node $S \cup \{x\}$ is a child of node S on the path from node S to node $S \cup S_1$, we can infer that $x \in S_1$. Now, if we can show that $x \notin S_2$, we can say that the nodes $S \cup S_1$ and $S \cup S_2$ differ by at least one vertex.

As there is never any duplicate vertex in V_{ext} of a node, $x \neq y$. So, if $S_2 = \{y\}$ (i.e., $S \cup S_2$ is a child of S), $x \notin S_2$. Now, we consider the case where $S_2 \neq \{y\}$, i.e., node $S \cup S_2$ is a descendant of node $S \cup \{y\}$. As x is located before y in V_{ext} of node S and consequently in V_{ext} of each of its descendants, x cannot appear after y in V_{ext} of a descendant of node $S \cup \{y\}$. Moreover, a descendant of node $S \cup \{y\}$ is generated only by extending with a vertex located after y in V_{ext}. So, $x \notin S_2$. □

Lemma 3 *No tree $T_{\alpha \neq v_0}$ has a node $S_D = \{v_0, \Pi_{i=1}^{m_1} v_1^{(i)}, \ldots, \Pi_{i=1}^{m_D} v_D^{(i)}\}$.*

Proof Once a vertex is included in V_{cvs} of a node, the vertex is never removed from the V_{cvs} of any of its descendants. As a result, V_{cvs} of every node (except root node) in a tree T_α contains the anchor vertex α in it. Since v_0 has the lowest rank in S_D, no tree T_α with $\alpha < v_0$ will generate the node S_D. Moreover, the checking done in line 8 ensures that a tree T_α does not generate any node by extending with a vertex lower than its anchor vertex α. So, no tree T_α with $\alpha > v_0$ will generate the node S_D either. □

Theorem 1 *Given an undirected graph G, the Miner algorithm enumerates the set of all connected vertex sets in G without any redundancy.*

Proof (Completeness) Lemma 1 shows that, for any connected vertex set $S \subseteq V$, where $v_0 \in S$ has the lowest rank among all the vertices in S, the tree T_{v_0} generates a node with $V_{cvs} = S$. Since every vertex in the graph is selected as an anchor vertex and a tree is generated from that vertex, each connected vertex set of the given graph must be generated as a node in at least one of the $|V|$ trees in the forest $\mathcal{F}$.

(Non-redundancy) Lemma 2 shows that a connected vertex set is not generated more than once by a tree, and Lemma 3 shows that a connected vertex set is not generated by more than one tree. Therefore, a connected vertex set is not generated more than once in the forest $\mathcal{F}$. □

We note that the Miner algorithm can be considered a variation of the ESU algorithm [7] with the key difference being the state of the list V_{ext} between recursive function calls. While updating the list V_{ext}, the ESU algorithm does not add vertices lower than or equal to the anchor vertex to the list V_{ext}, whereas the Miner algorithm does. This modification is required to efficiently identify the maximal or closed sets in the CSMiner algorithms (discussed later), which are based on the Miner algorithm.

Implementation details

We utilize the ranks of the vertices as unique labels of the vertices. We implement each of the node components V_{cvs} and V_{ext} by an array data structure of size $|V|$ that behaves like a stack. We maintain two indices to indicate the current elements in V_{cvs} and V_{ext}. Moreover, instead of having V_{cvs} and V_{ext} as local variables in the function EXTEND, we make both V_{cvs} and V_{ext} globally accessible so that only one instance of each variable is created during the entire execution of the algorithm. V_{cvs} and V_{ext}. After calling the function EXTEND to extend a node with a vertex v, we push v onto V_{cvs} and push all vertices in $\mathrm{N}(v)$, which are not present in V_{ext}, onto V_{ext}. Let m be the number of vertices pushed onto V_{ext}. Before returning from the function EXTEND, we pop one vertex from V_{cvs} and pop m vertices from V_{ext} to restore both variables to its previous state.

Let the vertices in the input graph be ranked from 1 to $|V|$. For the purpose of checking the presence of a vertex in V_{ext}, we associate with V_{ext} a boolean array $B : B[0], B[1], \ldots, B[|V|]$ with all entries initialized to *false*, indicating the absence of vertices in V_{ext}. When a vertex v is added to V_{ext} while generating a node, we update the value of $B[r(v)]$ to *true*, and when a vertex v is removed from V_{ext} while backtracking, we set $B[r(v)]$ to *false*. Now, to determine if a vertex v is present in V_{ext}, we simply check if $B[r(v)]$ is *true*, which is a constant time operation. So, adding a single neighbor to V_{ext} (with checking for duplicate entry) is a constant time operation.

Complexity analysis

Time complexity

An enumeration algorithm is said to have *linear delay* if the time to compute the next output (if a solution remains) or to detect that there is no more output (if no solution remains) is bounded by a linear function of the input size in the worst case.

Lemma 4 *The Miner algorithm has linear delay in terms of number of vertices in the input graph.*

Proof Let n denote the number of vertices in the input graph. The delay in the Miner algorithm is dominated by the following operations:

1. The number of times the comparison $x_i > \alpha$ fails at a stretch (line 8) before generating the succeeding node, where α is the anchor vertex, and x_i is a vertex selected from V_{ext} for extension.
2. (If a solution remains) Adding neighbors of the extending vertex v to V_{ext} of the succeeding node (line 5). Recall from the implementation details that adding a single neighbor to V_{ext} is a constant time operation. Since $|\mathrm{N}(v)| \leq n$, this operation takes $\mathcal{O}(n)$ time in the worst case.

Now we show that the first operation also takes $\mathcal{O}(n)$ time in the worst case. We use the placeholder notation $\sim_i$ to represent a sequence of vertices in V_{ext} of a node such that $v \in \sim_i \implies v < \alpha$. The integer subscript i is used to uniquely identify a sequence $\sim_i$ in a tree. In Fig. 3, we illustrate the three possible cases (a), (b), and (c) when there is at least one more solution to output, and the only possible case (d) when there is no more solution to output.

First we consider the case in Fig. 3a, where the current node is a non-leaf node S, and we analyze the time to generate its first child $S \cup \{v_1\}$. The algorithm executes $| \sim_0 |$ comparisons before generating the child $S \cup \{v_1\}$. Since V_{ext} of a node never contains a duplicate vertex, it follows that $| \sim_0 | < n$. So, the delay is $\mathcal{O}(n)$ in this case.

Now we consider the case in Fig. 3b, where the current node is a leaf node, and it is not located on the right-most branch of a tree. In this case, the leaf node must have an earliest ancestor $S \neq \{\}$ that generates the succeeding node. Let the current node be the leaf node $S \cup \{v_1, \ldots, v_k\}$, which is a descendent at depth k starting from the ancestor S. Now the algorithm executes $| \sim_{k-1} | + | \sim_k |$ comparisons, and then backtracks all the way up to the ancestor S. While backtracking, the algorithm executes

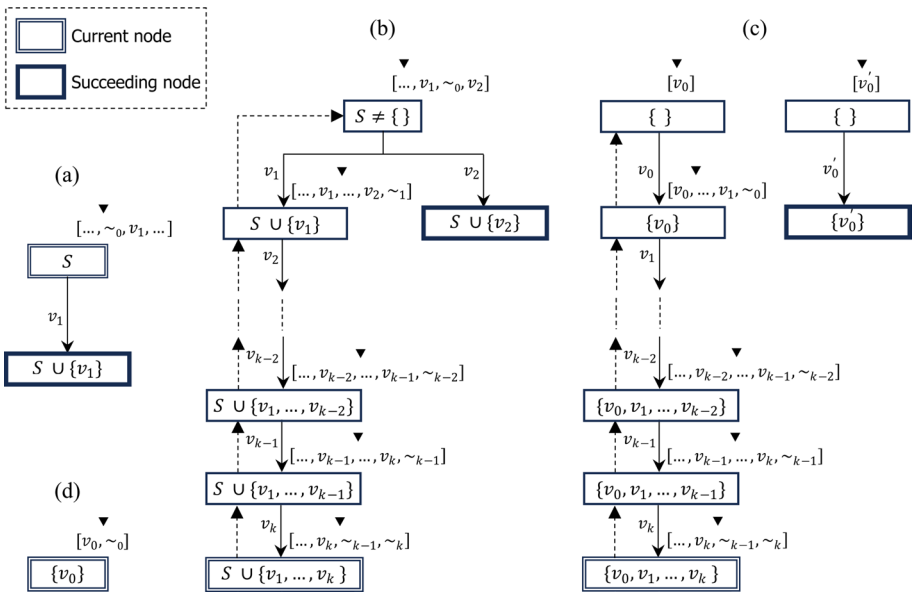

Fig. 3 Time Complexity Analysis. (**a**) Delay from a non-leaf node to its succeeding node. (**b**) Delay from a leaf node to its succeeding node in the same tree. (**c**) Delay from a leaf node to its succeeding node in the next tree. (**d**) Delay for detecting that no more output remains

$|\sim_{k-1}| + |\sim_{k-2}| + \ldots + |\sim_0|$ comparisons, and then extends node S with vertex v_2 to generate the succeeding node $S \cup \{v_2\}$. So, the total number of comparisons before generating the succeeding node is $|\sim_{k-1}| + (|\sim_k| + \ldots + |\sim_0|)$. As a vertex cannot appear more than once in V_{ext} of a node, all the vertices in the sequences $\sim_0, \ldots, \sim_k$ are unique. Hence, $|\sim_k| + \ldots + |\sim_0| < n$, which also implies that $|\sim_{k-1}| < n$. So, the delay is $\mathcal{O}(n)$ in this case as well.

The case in Fig. 3c occurs when the current node is a leaf node, and it is located on the right-most branch of a tree. Using arguments similar to that used in the last case, we can show that the delay in this case is also $\mathcal{O}(n)$.

The case in Fig. 3d (i.e, there is no more remaining output) occurs if and only if the current node is the anchor node of a tree T_{v_0}, where the anchor vertex v_0 has the highest rank among all the vertices in the input graph. This case is similar to the case in Fig. 3a. The algorithm executes $|\sim_0|$ comparisons. Since $|\sim_0| < n$, the delay is $\mathcal{O}(n)$. □

Space complexity

The algorithm uses two globally accessible lists V_{cvs} and V_{ext} of maximum size $n = |V|$. A boolean array B of size $n + 1$ keeps track of vertices present in V_{ext} (see implementation details). Moreover, the space required for the local variables in the function EXTEND is $\mathcal{O}(n)$ as the depth of the enumeration tree is bounded by n. So, the total space required by the algorithm, excluding the space required for the input graph, is $\mathcal{O}(n)$.

Mining cohesive connected vertex sets

A *vertex-attributed (VA) graph* $\mathcal{G} = (V, E, \mathcal{I}, f)$ is an undirected graph where each vertex has a set of attribute-terms associated with it. An attribute-term is just a name that represents a certain characteristic of a vertex. In the VA graph $\mathcal{G}$, V is the set of vertices, E is the set of edges (undirected), $\mathcal{I}$ is the set of all attribute-terms under consideration, and $f : V \to 2^{\mathcal{I}}$ is the function that associates each vertex $v \in V$ with an *attribute-term set* $I \subseteq \mathcal{I}$. We say that the attribute-term set of a vertex $v \in V$ is $f(v)$. Similarly, the attribute-term set of a vertex set $S \subseteq V$, denoted $A(S)$, is the set of all common attribute-terms associated with each vertex in S, i.e., $A(S) = \cap_{v \in S} f(v)$. The *support* of a vertex set $S \subseteq V$, denoted $|A(S)|$, is the number of elements in its attribute-term set $A(S)$.

Given a user-defined minimum support δ, a vertex set $S \subseteq V$ is a *cohesive* vertex set if and only if its support is at least δ, i.e., $|A(S)| \geq \delta$. Additionally, the vertex set S is a *cohesive connected* vertex set if and only if it is both cohesive and connected. The set of all cohesive connected vertex sets in $\mathcal{G}$ with minimum support δ:

$$\mathcal{P}_{\mathcal{G},\delta} = \{S \subseteq V : \mathcal{G}[S] \textit{ is connected} \wedge |A(S)| \geq \delta\} \tag{6}$$

Depending on its connectivity, a cohesive connected vertex set can have a large number of subsets that are also cohesive connected. For example, all the subsets of a

fully-connected cohesive vertex set S are cohesive connected. The number of these subsets is $2^{|S|} - 2$. It would generate an exponential number of patterns if we report all cohesive vertex sets. Thus, we propose to mine cohesive connected vertex sets that have less redundancy.

Maximal and closed sets

Given a minimum support δ, a cohesive connected vertex set $S \subseteq V$ is *maximal* if and only if there is no proper superset of S that is also a cohesive connected vertex set. A maximal cohesive connected vertex set cannot be extended while maintaining both connectivity and minimum support. The set of all maximal cohesive connected vertex sets in $\mathcal{G}$ with minimum support δ:

$$\mathcal{M}_{\mathcal{G},\delta} = \{S \in \mathcal{P}_{\mathcal{G},\delta} : \nexists S' \supset S \text{ such that } S' \in \mathcal{P}_{\mathcal{G},\delta}\} \tag{7}$$

Given a minimum support δ, a cohesive connected vertex set $S \subseteq V$ is *closed* if and only if there is no proper superset of S that is also a cohesive connected vertex set and has the same support as S. A closed cohesive connected vertex set cannot be extended while maintaining both connectivity and the same support. The set of all closed cohesive connected vertex sets in $\mathcal{G}$ with minimum support δ:

$$\mathcal{C}_{\mathcal{G},\delta} = \{S \in \mathcal{P}_{\mathcal{G},\delta} : \nexists S' \supset S \text{ such that } S' \in \mathcal{P}_{\mathcal{G},\delta} \wedge |A(S')| = |A(S)|\} \tag{8}$$

Let $S \cup \{v\}$ be a superset of S, where $v \in V \setminus S$. In both equations 7 and 8, for a superset $S \cup \{v\}$ to be connected, v must be adjacent to at least one vertex in S, i.e., v must be in the open neighborhood of S. So, we rewrite the sets of maximal and closed sets as:

$$\mathcal{M}_{\mathcal{G},\delta} = \{S \in \mathcal{P}_{\mathcal{G},\delta} : \nexists v \in N_{op}(S) \text{ such that } |A(S \cup \{v\})| \geq \delta\} \tag{9}$$

$$\mathcal{C}_{\mathcal{G},\delta} = \{S \in \mathcal{P}_{\mathcal{G},\delta} : \nexists v \in N_{op}(S) \text{ such that } |A(S \cup \{v\})| = |A(S)|\} \tag{10}$$

By definition, every maximal and closed set is a cohesive connected vertex set. Moreover, every maximal set is a closed set as a maximal set cannot be extended. So, we have $\mathcal{M}_{\mathcal{G},\delta} \subseteq \mathcal{C}_{\mathcal{G},\delta} \subseteq \mathcal{P}_{\mathcal{G},\delta}$. Both $\mathcal{M}_{\mathcal{G},\delta}$ and $\mathcal{C}_{\mathcal{G},\delta}$ are concise representations of $\mathcal{P}_{\mathcal{G},\delta}$.

Problem Definition 2 Given a VA graph $\mathcal{G}$ and minimum support δ, enumerate the set of all maximal cohesive connected vertex sets $\mathcal{M}_{\mathcal{G},\delta}$.

Problem Definition 3 Given a VA graph $\mathcal{G}$ and minimum support δ, enumerate the set of all closed cohesive connected vertex sets $\mathcal{C}_{\mathcal{G},\delta}$.

Algorithm 2, first presented in our previous work [11], shows pseudo-code of the CSMiner-maximal algorithm that enumerates the set $\mathcal{M}_{\mathcal{G},\delta}$ in a VA graph $\mathcal{G}$. Algorithm 2 is provided here for comparison with our newly proposed algorithm 3.

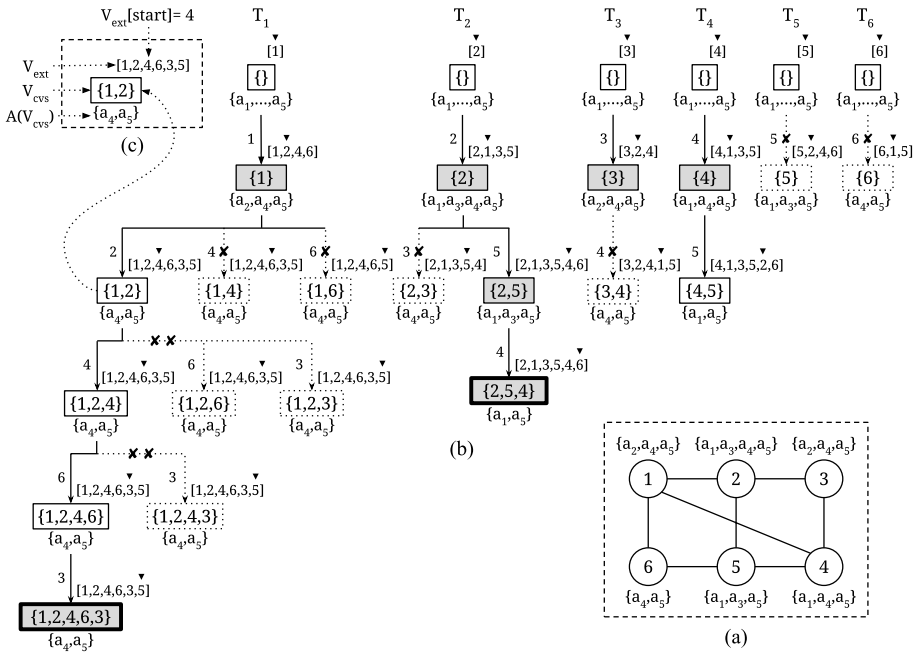

Fig. 4 Enumeration of cohesive connected vertex sets in a sample VA graph. (**a**) A sample VA graph. Each vertex is labeled by its rank. The attribute-term set of each vertex is shown beside the vertex, e.g., $f(1) = \{a_2, a_4, a_5\}$, etc. (**b**) Enumeration forest $\{T_1, \ldots, T_6\}$ generated by each of the Algorithms 2 and 3 with minimum support $\delta = 2$ for the sample VA graph in (**a**). Nodes with thick border are maximal and output by Algorithm 2, whereas nodes with gray color are closed and output by Algorithm 3. (**c**) Components of a node in a tree

Algorithm 3 shows the pseudo-code of the proposed CSMiner-closed algorithm that enumerates the set $\mathcal{C}_{\mathcal{G},\delta}$ in a VA graph $\mathcal{G}$. If our goal was to list all cohesive connected vertex sets, rather than just the closed ones, we could just adapt the Miner algorithm and incorporate an additional checking during child generation (i.e., before calling the function EXTEND) to ensure that each child generated is cohesive. Please note that, if a child is not cohesive, there is no need to generate and extend it any further because the cohesiveness property is an anti-monotone constraint, which ensures that, if a node is not cohesive, none of its superset is cohesive. After this modification (lines 2 and 13 in Algorithm 3), each node in a tree will represent a cohesive connected vertex set, and the forest $\mathcal{F}$ will generate all cohesive connected vertex sets in the given VA graph. However, since we are interested in only the closed cohesive connected vertex sets, we also need to check which of these nodes generated in a tree are closed.

In Fig. 4, (b) shows the complete forest for the sample VA graph in (a) with minimum support $\delta = 2$.

Algorithm 2 CSMiner-maximal

Input : $\mathcal{G} = (V, E, \mathcal{I}, f)$: a VA graph,
δ: minimum support
Output: $\mathcal{M}_{\mathcal{G},\delta}$

```
/* 0-based list indexing assumed */
1  foreach α ∈ V do
2      if |A({α})| ≥ δ then
3          foreach u ∈ N(α), with u < α do
4              if A({α, u})=A({α}) then
5                  goto line 1 // prune
6          EXTEND({}, [α], α, 0 + 1)
7  Function EXTEND(V_cvs, V_ext, v, start)
8      V_cvs ← V_cvs ∪ {v}
9      V_ext ← V_ext || (N(v)\V_ext)
10     maximal ← true
11     foreach x_i ∈ V_ext, with i ≥ start do
12         Let V'_cvs = V_cvs ∪ {x_i}
13         if |A(V'_cvs)| ≥ δ then
14             maximal ← false
15             if x_i > α then
16                 foreach x_j ∈ V_ext, with j < i do
17                     if x_j ∉ V_cvs ∧ A(V'_cvs ∪ {x_j})=A(V'_cvs) then
18                         goto line 11 // prune
19                 EXTEND(V_cvs, V_ext, x_i, i + 1)
20             if A(V'_cvs)=A(V_cvs) then
21                 break // prune rest
22     if maximal then
23         foreach x_i ∈ V_ext, with i < start − 1 do
24             if x_i ∉ V_cvs ∧ |A(V_cvs ∪ {x_i})| ≥ δ then
25                 maximal ← false
26                 break
27         if maximal then
28             output V_cvs
```

Algorithm 3 CSMiner-closed

Input : $\mathcal{G} = (V, E, \mathcal{I}, f)$: a VA graph,
δ: minimum support
Output: $\mathcal{C}_{\mathcal{G},\delta}$

```
/* 0-based list indexing assumed */
1  foreach α ∈ V do
2      if |A({α})| ≥ δ then
3          foreach u ∈ N(α), with u < α do
4              if A({α, u})=A({α}) then
5                  goto line 1 // prune
6          EXTEND({}, [α], α, 0 + 1)
7  Function EXTEND(V_cvs, V_ext, v, start)
8      V_cvs ← V_cvs ∪ {v}
9      V_ext ← V_ext || (N(v)\V_ext)
10     closed ← true
11     foreach x_i ∈ V_ext, with i ≥ start do
12         Let V'_cvs = V_cvs ∪ {x_i}
13         if |A(V'_cvs)| ≥ δ then
14             if x_i > α then
15                 foreach x_j ∈ V_ext, with j < i do
16                     if x_j ∉ V_cvs ∧ A(V'_cvs ∪ {x_j})=A(V'_cvs) then
17                         goto line 11 // prune
18                 EXTEND(V_cvs, V_ext, x_i, i + 1)
19             if A(V'_cvs)=A(V_cvs) then
20                 closed ← false
21                 break // prune rest
22     if closed then
23         output V_cvs
```

Pruning

Generating all nodes (each representing a cohesive connected vertex set) in the forest is still computationally expensive. However, we do not need to generate all nodes in order to enumerate the set of all closed sets because many branches in the enumeration trees do not lead to any closed set. Early detection and pruning of these futile branches improves the performance of the algorithm. We propose the following pruning strategies for early pruning of subtrees that do not lead to any closed set. We propose the following pruning strategies for early pruning of subtrees that do not lead to any closed set. Figure 5 illustrates the three pruning strategies.

Pruning Anchor Node

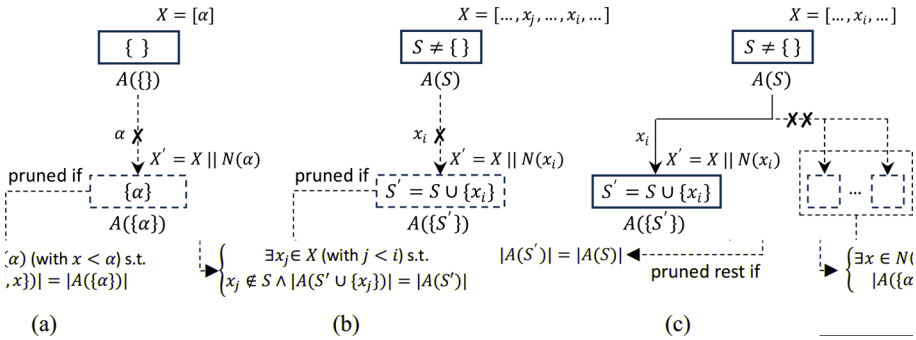

Fig. 5 Conditions for (**a**) pruning an anchor node, (**b**) pruning a non-anchor node, and (**c**) pruning rest of the child nodes

Lemma 5 *Let* $\{\alpha\}$ *be the anchor node of an enumeration tree* T_α*. If there is a vertex* $x \in \mathrm{N}(\alpha)$ *such that* $x < \alpha$ *and* $A(\{\alpha, x\}) = A(\{\alpha\})$*, the tree* T_α *does not generate any closed cohesive connected vertex set.*

Proof First we show that the anchor node $\{\alpha\}$ is not closed. Then we show that any descendant of the anchor node $\{\alpha\}$ is also not closed.

Since x is a neighbor of α and $x \neq \alpha$ (as $x < \alpha$), x belongs to the open neighborhood of α, i.e, $x \in N_{op}(\{\alpha\})$. Moreover, the vertex set $\{\alpha, x\}$ has the same attribute-term set as that of $\{\alpha\}$, i.e., $A(\{\alpha, x\}) = A(\{\alpha\})$. So, node $\{\alpha\}$ can be extended using a vertex in $N_{op}(\{\alpha\})$ while maintaining the same support, i.e., the anchor node $\{\alpha\}$ is not closed.

Let the node $\{\alpha\} \cup S$ be a descendant of the anchor node $\{\alpha\}$. Since $x < \alpha$ and tree T_α does not generate any node by extending with a vertex with lower rank than that of the anchor vertex α, we infer that $x \notin \{\alpha\} \cup S$. Now, $x \in \mathrm{N}(\alpha) \wedge x \notin (\{\alpha\} \cup S) \implies x \in N_{op}(\{\alpha\} \cup S)$. Moreover, $A(\{\alpha, x\}) = A(\{\alpha\}) \implies A(\{\alpha\} \cup S \cup \{x\}) = A(\{\alpha\} \cup S)$. So, node $\{\alpha\} \cup S$ can also be extended using a vertex in $N_{op}(\{\alpha\} \cup S)$ while maintaining the same support, i.e., a descendant of the anchor node $\{\alpha\}$ is also not closed. □

Due to Lemma 5, we can prune the entire subtree rooted at anchor node $\{\alpha\}$ in T_α (lines 3-5 in Algorithm 3). In Fig. 4, the anchor node $\{5\}$ in tree T_5 is pruned because vertex 5 has an adjacent vertex 2 such that $2 < 5$ and $A(\{5, 2\}) = A(\{5\}) = \{a_1, a_3, a_5\}$. Notice that node $\{2, 5\}$ is generated in the tree T_2. Similarly, the anchor node $\{6\}$ in tree T_6 is pruned.

Pruning non-anchor node

Lemma 6 *Let* $S \neq \{\}$ *be a node with* $V_{ext} = X$*, and* $S' = S \cup \{x_i\}$ *be a child of the node* S *in a tree, where* $x_i = X[i]$ *is the extending vertex. If there is a vertex* $x_j = X[j]$ *with* $j < i$ *such that* $x_j \notin S$ *and* $A(S' \cup \{x_j\}) = A(S')$*, the subtree rooted at node* S' *does not generate any closed cohesive connected vertex set, and we say that the child* S' *is covered by vertex* x_j*.*

Proof First we show that the node S' is not closed. Then we show that a descendant of the node S' is also not closed.

Since the vertex x_j is in V_{ext} of node S and $x_j \notin S$, it implies that x_j is in the open neighborhood of S, i.e., $x_j \in N_{op}(S)$ and consequently $x_j \in N_{op}(S')$ (as $x_j \neq x_i$). Moreover, $A(S' \cup \{x_j\}) = A(S')$. So, node S' can be extended using a vertex in $N_{op}(S')$ while maintaining the same support, i.e., node S' is not closed.

Let the node $S' \cup S_1$ be a descendant of node S'. As the vertex x_j is located before the vertex x_i in V_{ext} of node S and consequently in V_{ext} of each of its descendants, x_j cannot appear after x_i in V_{ext} of any descendant of node S'. Moreover, a descendant of node S' is generated only by extending with a vertex located after x_i in its V_{ext}. So, $x_j \notin S_1$. Now, $x_j \in N_{op}(S') \wedge x_j \notin S_1 \implies x_j \in N_{op}(S' \cup S_1)$. Moreover, $A(S' \cup \{x_j\}) = A(S') \implies A(S' \cup S_1 \cup \{x_j\}) = A(S' \cup S_1)$. So, node $S' \cup S_1$ can be extended using a vertex in $N_{op}(S' \cup S_1)$ while maintaining the same support, i.e., node $S' \cup S_1$ is also not closed.

□

Due to Lemma 6, we can prune the entire subtree rooted at node S' (see lines 15-17 in Algorithm 3). In Fig. 4, node $\{1,4\}$ in tree T_1 is pruned as it is covered by vertex 2, i.e., vertex 2 precedes vertex 4 in V_{ext} of the parent node $\{1\}$ and $A(\{1,4\} \cup \{2\}) = A(\{1,4\}) = \{a_4, a_5\}$. Similarly, node $\{1,6\}$ in T_1 is covered by vertex 2, node $\{2,3\}$ in T_2 is covered by vertex 1, node $\{3,4\}$ in T_3 is covered by vertex 2, and hence pruned.

Pruning rest

This is just a special case of the pruning non-anchor node technique, where we can apply the technique in batch. Let us consider a node $S \neq \{\}$. If there is a child $S \cup \{x_i\}$ of node S such that $A(S \cup \{x_i\}) = A(S)$, we can prune, at once, every child $S \cup \{x_j\}$ of node S, where x_j succeeds x_i in V_{ext} of node S (i.e., with $j > i$), as all these nodes will be covered by vertex x_i (line 21 in Algorithm 3). Please note that this pruning rest technique is applied just to speed up the application of pruning non-anchor node technique. If we do not apply this pruning rest technique, all the same nodes will still be pruned by pruning non-anchor node technique.

In Fig. 4, the branch leading to child nodes $\{1,2,6\}$ and $\{1,2,3\}$ of node $\{1,2\}$ in tree T_1 is pruned by this technique. As the node $\{1,2\}$ has a child $\{1,2,4\}$ such that $A(\{1,2,4\}) = A(\{1,2\}) = \{a_4, a_5\}$, every child of node $\{1,2\}$ that would be generated by extending with a vertex located after vertex 4 in its V_{ext} can be pruned at once. Similarly, node $\{1,2,4,3\}$ in tree T_1 is pruned by this technique.

Neighbor adding optimization

We can further optimize the algorithm if, while generating a child $S' = S \cup \{x_i\}$ from a node S in an enumeration tree, we append a neighbor $u \in \mathrm{N}(x_i)$ to V_{ext} of node S' only if $S' \cup \{u\}$ is cohesive, i.e., $|A(S' \cup \{u\})| \geq \delta$. We can do this because, if $S' \cup \{u\}$ is not cohesive, none of its supersets will be cohesive. This optional optimization is not shown in the Algorithms.

Identification of maximal and closed sets

Both CSMiner algorithms generate the same forest for a given minimum support δ, where every node in the forest represents a cohesive connected vertex set. We can incorporate the same pruning techniques and neighbor adding optimization into both CSMiner algorithms. The main difference between the two algorithms lies in the method of identifying maximal nodes versus closed nodes in a tree.

Let us assume that an enumeration tree has a node $S' = S \cup \{x_i\}$ with $V_{ext} = X'$ and $start = i + 1$, where $x_i = X'[i]$ is the extending vertex. According to Equation 5, $N_{op}(S') = X' \setminus S'$.

Maximality checking in algorithm 2

The node $S' = S \cup \{x_i\}$ is maximal if and only if it cannot be extended using a vertex in $N_{op}(S')$ while maintaining minimum support (Equation 9). So, to find out whether the node S' is maximal or not, we need to check if there is a vertex $x_j = X'[j]$ such that $x_j \in S'$ and $|A(S' \cup \{x_j\})| \geq \delta$. If we find such a vertex x_j located after vertex x_i (i.e., with $j > i$) in X', we mark the node S' as *not maximal* (line 14 in Algorithm 2). Otherwise, we check if there is such a vertex x_j located before x_i (i.e., with $j < i$) in X', and if we find such a vertex, we mark the node S' as *not maximal* (line 25 in Algorithm 2). Finally, if we do not find such a vertex x_j (neither after nor before x_i) in X', we output the node S' as *maximal* (see Fig. 6 and line 28 in Algorithm 2).

In Fig. 4, node $\{4, 5\}$ in tree T_4 is not maximal because there is a vertex 2 in its V_{ext} such that $A(\{4, 5\} \cup \{2\}) \geq \delta$. However, node $\{1, 2, 4, 6, 3\}$ in tree T_1 and node $\{2, 5, 4\}$ in tree T_2 are maximal as neither of these two nodes can be extended using a vertex in V_{ext} while maintaining minimum support.

Closedness checking in algorithm 3

The node $S' = S \cup \{x_i\}$ is closed if and only if it cannot be extended using a vertex in $N_{op}(S')$ while maintaining the same support (Equation 10). So, to find out whether the node S' is closed or not, we need to check if there is a vertex $x_j = X'[j]$ such that $x_j \notin S'$ and $|A(S' \cup \{x_j\})| = |A(S')|$.

If $S' = S \cup \{x_i\}$ is the anchor node (i.e., $S = \{\}$ and x_i is the anchor vertex), there is no vertex before vertex x_i in X'. Now let us consider the case where S' is a descendant of the anchor node (i.e., $S \neq \{\}$). In this case, if there existed a vertex x_j located before vertex

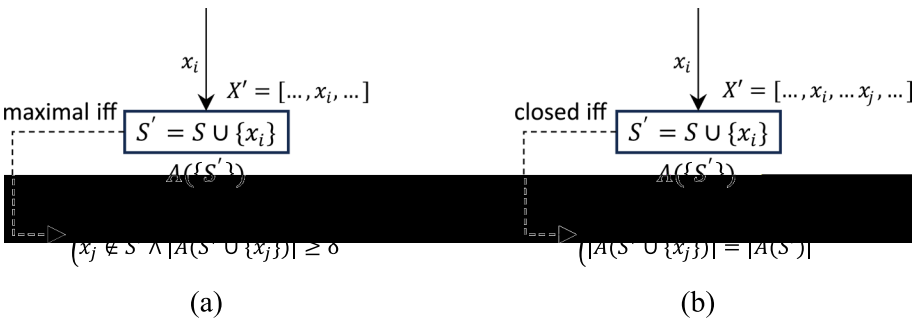

Fig. 6 (**a**) Maximality checking. (**b**) Closedness checking

x_i (i.e., with $j < i$) in X' such that $x_j \notin S'$ and $|A(S' \cup \{x_j\})| = |A(S')|$, node S' would be pruned (i.e., not generated) by the algorithm (see Lemma 6 and its applications). Assuming that the node S' is generated (i.e., not pruned) by the algorithm, we can assert without any checking that there is no such vertex x_j located before vertex x_i in X'.

So, to find out whether the node S' is closed or not, we just need to check if there is a vertex x_j located after vertex x_i (i.e., with $j > i$) in X' such that $x_j \notin S'$ and $|A(S' \cup \{x_j\})| = |A(S')|$. However, we also do not need to check if the condition $x_j \notin S'$ is satisfied in this case, as the algorithm has not added any vertex located after the extending vertex x_i in X' to the node S'. Finally, if we find a vertex x_j located after vertex x_i in X' such that $|A(S' \cup \{x_j\})| = |A(S')|$, we mark node S' as *not closed*, otherwise we output node S' as *closed* (see Fig. 6 and lines 20 and 23 in Algorithm 3).

In Fig. 4, node $\{1, 2\}$ in tree T_1 is not closed because there is a vertex 4 located after the extending vertex 2 in its V_{ext} such that $A(\{1, 2\}) = A(\{1, 2, 4\})$. However, node $\{2, 5\}$ in tree T_2 is closed as it cannot be extended using a vertex located after the extending vertex 5 in its V_{ext} without reducing its support.

Results

We implemented our algorithm in C++. For runtime comparison with existing algorithms, we used the C++ implementations provided by the respective authors. We used the GNU gcc compiler with -O3 optimization to compile all programs. For measuring runtime, we ran each program on a Linux computer with Intel Xeon E5-2670 v2 processor and recorded the total CPU time spent by the process in both user and kernel modes.

Data set

We used the BioGRID human PPI network, version 4.4.200 [12], as the input graph $\mathcal{G}$. The network has 25, 239 genes (vertices) and 588, 719 unique physical interactions (edges). We associated a set of different cancers (attribute-terms) with each gene, where an association between a gene (a vertex) and a cancer (an attribute-term) indicates that the gene is differentially expressed with $|log(FC)| \geq 2$ in that cancer. We generated these gene-disease associations for 17 cancers from the TCGA research network. The list of the cancers along with the number of samples and dysregulated genes is shown in Table 1.

Performance comparison

We measured the runtime of the proposed CSMiner-closed algorithm to enumerate the set of closed cohesive connected vertex sets in the vertex-attributed BioGRID human PPI network with varying minimum support, ranging from 1 to 12. We compared the result with the runtime of CSMiner-maximal and RSSP-maximal algorithms [10, 11]. Note that both CSMiner-maximal and RSSP-maximal algorithms enumerate the set of maximal cohesive connected vertex sets $\mathcal{M}_{\mathcal{G},\delta}$, which is a subset of the set of closed patterns (i.e., $\mathcal{M}_{\mathcal{G},\delta} \subseteq \mathcal{C}_{\mathcal{G},\delta}$). The set of patterns reported by CSMiner-closed is more comprehensive than those reported by CSMiner-maximal and RSSP-maximal. Moreover, we could generate the maximal patterns from the set of closed patterns reported by CSMiner-closed by retaining only the patterns that do not have any superpattern in the set.

Table 1 Dysregulated genes from cancer data

Sl No.	Cancer	Total No. of Samples	No. of Control Samples	No. of Tumor Samples	No. of Dysregulated Genes
1	TCGA-UCEC	586	35	551	2,481
2	TCGA-ESCA	172	11	161	1,371
3	TCGA-PRAD	550	52	498	679
4	TCGA-KIRC	610	72	538	2,206
5	TCGA-HNSC	544	44	500	1,827
6	TCGA-KICH	89	24	65	2,779
7	TCGA-READ	176	10	166	2,497
8	TCGA-THCA	560	58	502	1,130
9	TCGA-LIHC	421	50	371	1,981
10	TCGA-CHOL	45	9	36	3,933
11	TCGA-KIRP	320	32	288	2,288
12	TCGA-BLCA	433	19	414	2,188
13	TCGA-BRCA	1,215	113	1,102	2,039
14	TCGA-COAD	519	41	478	2,293
15	TCGA-LUSC	551	49	502	3,169
16	TCGA-STAD	407	32	375	1,353
17	TCGA-LUAD	592	59	533	2,265

Table 2 Runtime comparison in the PPI network $\mathcal{G}$

δ	$\lvert\mathcal{M}_{\mathcal{G},\delta}\rvert$	$\lvert\mathcal{C}_{\mathcal{G},\delta}\rvert$	RSSP-maximal (t_r sec)	CSMiner-maximal (t_{cm} sec)	CSMiner-closed (t_{cc} sec)	t_r/t_{cc}	t_{cm}/t_{cc}
1	1060	85,262	6,350.10	655.50	653.80	9.71	1.00
2	1647	84,978	3,489.30	432.70	430.10	8.11	1.01
3	2727	84,435	1,963.30	307.40	306.00	6.42	1.00
4	5136	83,067	1,050.80	219.37	218.40	4.81	1.00
5	9695	79,422	513.20	150.20	149.80	3.43	1.00
6	15,341	71,272	256.74	93.49	93.65	2.74	1.00
7	18,974	57,495	102.19	51.49	51.40	1.99	1.00
8	17,844	40,027	43.97	24.30	24.18	1.82	1.00
9	12,853	23,587	15.47	10.06	9.95	1.55	1.01
10	7391	11,931	4.95	3.82	3.79	1.31	1.01
11	3740	5490	1.77	1.51	1.50	1.18	1.01
12	1798	2453	0.81	0.77	0.78	1.04	0.99

Table 2 shows the runtime comparison for this experiment. There was no significant difference in runtime between the CSMiner-closed and CSMiner-maximal algorithms, which is reasonable as both algorithms generate the same enumeration forest for a given minimum support. While both have the same running time, CSMiner-closed reports a much larger set of patterns than CSMiner-maximal. Moreover, when minimum support δ was high ($\delta \geq 10$), there was not much difference in the runtime among the three algorithms as the enumeration trees cannot generate deeper nodes while maintaining a large node support. The speedup in runtime of CSMiner-closed over RSSP-maximal becomes evident for small thresholds where the algorithms had to explore deeper nodes in the

enumeration trees. For $\delta = 1$ and $\delta = 2$, the CSMiner-closed algorithm was approximately 10 and 8 times faster than the RSSP-maximal algorithm respectively.

Analysis of reported gene sets

For each minimum support δ, ranging from 1 to 12, the CSMiner-closed algorithm reports a collection of gene sets $\mathcal{C}_{\mathcal{G},\delta}$ from the vertex-attributed BioGRID human PPI network $\mathcal{G}$. From each gene set collection $\mathcal{C}_{\mathcal{G},\delta}$, we removed gene sets that contained less than 3 genes, and the resulting collection is referred to as $\mathcal{C}^*_{\mathcal{G},\delta}$. Each of these gene sets induce a subgraph and we computed some topological properties of the subgraphs induced by the gene sets in each collection $\mathcal{C}^*_{\mathcal{G},\delta}$. Moreover, to assess the biological relevance of the genes in each gene set, we performed biological enrichment analysis that validates if a gene set overlaps with known biological processes and pathways.

Topological properties

Table 3 summarizes the average of the topological properties, order $|V_{sub}|$, size $|E_{sub}|$, and density ρ_{sub}, of the subgraphs induced by the reported gene sets in each collection $\mathcal{C}^*_{\mathcal{G},\delta}$. As we increased δ, the average order $\overline{|V_{sub}|}$ decreased. For a large δ, it is unlikely that a cohesive connected induced subgraph can be extended much while maintaining the minimum support δ. It is also unlikely to find a large number of connected vertices that satisfy the attribute constraint. Moreover, as we increased δ, average size $\overline{|E_{sub}|}$ also decreased. However, we did not observe any significant trend in the average density $\overline{\rho_{sub}}$ with change in δ. Please note that the algorithm does not impose any separate constraint on the number of edges or density of the output subgraphs. However, in order to remain connected, an output subgraph with n vertices must have at least $n-1$ edges, thus resulting in a minimum density of $\frac{n-1}{n*(n-1)/2} = \frac{2}{n}$.

Table 3 Topological properties and enrichment analysis of gene sets in $\mathcal{C}^*_{\mathcal{G},\delta}$

Topological properties					"% of closed gene sets enriched with signatures					
δ	$\lvert\mathcal{C}^*_{\mathcal{G},\delta}\rvert$	$\overline{\lvert V_{sub}\rvert}$	$\overline{\lvert E_{sub}\rvert}$	$\overline{\rho_{sub}}$	KEGG	CGN	CM	CC	MF	C6
1	77,400	122	319	0.09	90	86	96	88	97	91
2	77,378	121	315	0.09	90	86	96	88	97	91
3	77,214	119	308	0.09	90	86	96	88	97	91
4	76,350	115	297	0.09	90	86	96	88	97	91
5	73,334	108	282	0.09	90	86	96	88	97	91
6	65,958	101	266	0.09	91	86	97	87	97	91
7	52,953	94	251	0.09	92	88	97	87	98	92
8	36,291	88	238	0.09	93	89	97	87	98	94
9	20,618	82	225	0.10	93	91	97	87	99	95
10	9667	76	208	0.11	93	92	96	87	99	95
11	3824	68	186	0.13	92	92	94	88	98	93
12	1303	60	162	0.15	92	93	91	90	97	91

Gene set enrichment analysis

Gene set enrichment (or over-representation) analysis uses a predefined collection P of gene sets that have been grouped together based on their biological property such as involvement in the same biological pathway, sharing the same process, sharing proximal location on a chromosome, etc. [13]. We refer to a gene set in such a predefined collection as a *module*. A module $p \in P$ is called enriched (or over-represented) in a gene set q, reported by the algorithm, if genes in p are present significantly more than expected in q. For a module p and a reported gene set q, the enrichment analysis assesses the over-representation of the genes of the module p in the genes of the reported gene set q. We used hypergeometric test with a p-value = 0.05 for significant over-representation. In this experiment, we used the following signature gene set collections from Molecular Signatures Database (MSigDB) [13, 14] as a predefined collection.

1. **KEGG**: KEGG canonical pathway, a collection of 186 modules.
2. **CGN**: computational cancer gene neighborhoods, a collection of 427 modules.
3. **CM**: computational cancer modules, a collection of 431 modules.
4. **CC**: gene ontology cellular component, a collection of 996 modules
5. **MF**: gene ontology molecular function, a collection of 1,708 modules
6. **C6**: oncogenic signature gene sets, a collection of 189 modules

A gene set q is enriched with modules of a predefined collection P if at least one of the modules in P is enriched in q. Table 3 shows the percentage of the gene sets in $\mathcal{C}^*_{\mathcal{G},\delta}$ that were found enriched with modules of a predefined collection. This analysis shows that most of the reported gene sets were enriched with multiple predefined signature module collections. Note that a module in a predefined collection can be enriched in many gene sets in $\mathcal{C}^*_{\mathcal{G},\delta}$, and similarly, a gene set in $\mathcal{C}^*_{\mathcal{G},\delta}$ can be enriched with many modules in a predefined collection. We sorted the modules in a predefined collection by the number of gene sets in $\mathcal{C}^*_{\mathcal{G},\delta}$ that they were enriched in. Table shows the top five modules in each predefined collection that were enriched in the highest number of gene sets in $\mathcal{C}^*_{\mathcal{G},12}$, where N indicates the number of gene sets in $\mathcal{C}^*_{\mathcal{G},12}$ in which a module was enriched. For example, in C6 oncogenic signature collection, the module rps14_dn.v1_dn was enriched (over-represented) in the highest number of gene sets (1,123 in total) reported by our algorithm for $\delta = 12$ (i.e., $\mathcal{C}^*_{\mathcal{G},12}$). This module contains a total of 186 genes that are down-regulated in CD34+ hematopoietic progenitor cells after knockdown of rps14 by RNAi [13, 14]. Recall that multiple closed patterns can share a common subgraph core, and if the core has significant overlap with a module in a collection, then the module will likely be enriched in all the closed patterns.

We also sorted the gene sets by the number of modules (in a predefined collection) that they were enriched with. Table 4 show the top five enriched modules for each predefined collection from MSigDB. Figure 7 shows some gene sets that were enriched with the highest number of modules in a predefined collection.

Discussion and conclusions

Integrating gene expression profile with protein-protein interaction network analysis provides a powerful approach for discovering functional biological pathways and complexes.

In this work, we proposed an algorithm to enumerate the set of all closed cohesive connected vertex sets whose vertices share common attributes. The proposed approach returns a representative set of the cohesive patterns. To overcome the large search space of all cohesive subnetworks, we proposed strategies for pruning the branches in the enumeration trees that do not produce any target node. These pruning strategies improve the performance of the algorithm. Runtime comparison with existing algorithms shows the efficiency of our proposed approach compared to algorithms that report a smaller set of maximal patterns.

The patterns extracted by our algorithm correspond to connected interaction subnetworks whose genes share similar dysregulation profiles. Biological enrichment analysis of the reported patterns shows that the gene sets have high biological significance as they overlap with standard datasets of curated biological pathways and complexes.

In future work, we plan to extend our algorithm to consider real-valued attributes and mine subnetworks with similar attribute values. Moreover, we plan to investigate integrating interaction constraints, e.g., degree or density, while mining cohesive subnetworks. We plan to develop a parallel implementation of this algorithm to facilitate mining patterns from large-scale networks with high-dimensional gene data.

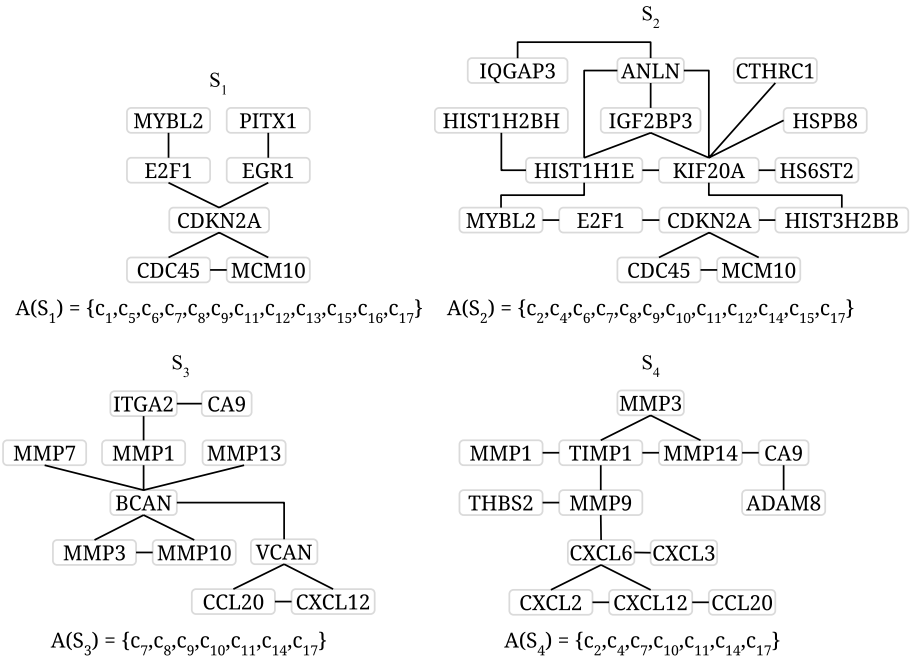

Fig. 7 Some of the most enriched gene sets (i.e., their induced subgraphs) reported by our algorithm from the vertex-attributed BioGRID human PPI network. Gene sets S_1 and S_2, among all gene sets in $\mathcal{C}^*_{\mathcal{G},12}$, were the most enriched with the KEGG modules. Gene sets S_3 and S_4 in $\mathcal{C}^*_{\mathcal{G},7}$, were the most enriched with the C6 modules. The attribute-terms C_1, C_2, etc. refer to the cancers in Table 1 with serial no. 1, 2, etc. respectively

Table 4 Top five enriched modules in each predefined collection from MSigDB (for $\delta = 12$)

KEGG	N	CGN	N
cell_cycle	1,095	gnf2_cdc2	1,178
p53_signaling pathway	854	gnf2_cenpf	1,092
oocyte_meiosis	846	gnf2_bub1	989
progesterone_mediated_oocyte_maturation	846	gnf2_bub1b	983
pancreatic_cancer	360	gnf2_espl1	982
CM	N	CC	N
module_57	1,107	cell_division_site	1,136
module_54	1,088	midbody	973
module_439	1,080	contractile_ring	971
module_52	1,075	kinesin_complex	855
module_451	1,074	intercellular_bridge	851
MF	N	C6	N
dna_replication_origin_binding	1,114	rps14_dn.v1_dn	1,123
motor_activity	942	prc2_ezh2_up.v1_dn	1,094
microtubule_binding	862	prc2_eed_up.v1_dn	1,090
tubulin_binding	858	gcnp_shh_up_early.v1_up	1,018
protein_serine_threonine_kinase_activity	850	csr_late_up.v1_up	1,008

Acknowledgements
The results shown here are partly based upon data generated by the TCGA Research Network: https://www.cancer.gov/tcga.

Author contribution
SS and RH discussed the idea and designed the algorithm. RH wrote the code for the algorithm. SS and RH worked on experimental design and subgraphs topological and biological analysis. SS and RH wrote the paper.

Funding
The authors express their gratitude to the funding provided to support this study from National Science Foundation (NSF) EPSCoR RII Track-2 Program under the grant number OIA-2119691. The findings and opinions expressed in this article are those of the authors only and do not necessarily reflect the views of the sponsors.

Data availability
Materials availability The software tool will be made available at https://www.cs.ndsu.nodak.edu/~ssalem/multirelation.html

Code availability
https://www.cs.ndsu.nodak.edu/~ssalem/multirelation.html.

Declarations

Ethics approval and consent to participate
Not Applicable

Consent for publication
All the authors read and approved the final manuscript.

Competing interests
The authors declare no competing interests.

Received: 22 July 2024 Accepted: 16 October 2024
Published online: 14 November 2024

References

1. Chuang H-Y, Lee E, Liu Y-T, Lee D, Ideker T. Network-based classification of breast cancer metastasis. Mol Syst Biol. 2007;3(1):140. https://doi.org/10.1038/msb4100180.
2. Maxwell S, Chance MR, Koyutürk M. Efficiently Enumerating All Connected Induced Subgraphs of a Large Molecular Network. In: Dediu A-H, Martín-Vide C, Truthe B, editors. Algorithms for Computational Biology. Cham: Springer; 2014. p. 171–82. https://doi.org/10.1007/978-3-319-07953-0_14.

3. Georgii E, Dietmann S, Uno T, Pagel P, Tsuda K. Enumeration of condition-dependent dense modules in protein interaction networks. Bioinformatics. 2009;25(7):933–40. https://doi.org/10.1093/bioinformatics/btp080.
4. Moser F, Colak R, Rafiey A, Ester M. Mining Cohesive Patterns from Graphs with Feature Vectors. In: Proceedings of the 2009 SIAM International Conference on Data Mining (SDM). Proceedings. Society for Industrial and Applied Mathematics, 2009. p. 593–604. https://doi.org/10.1137/1.9781611972795.51.
5. Chowdhury SA, Nibbe RK, Chance MR, Koyutürk M. Subnetwork state functions define dysregulated subnetworks in Cancer. In: Berger B, editor. Research in computational molecular biology. Berlin, Heidelberg: Springer; 2010. p. 80–95. https://doi.org/10.1007/978-3-642-12683-3_6.
6. Ideker T, Ozier O, Schwikowski B, Siegel AF. Discovering regulatory and signalling circuits in molecular interaction networks. Bioinformatics. 2002;18(suppl-1):233–40. https://doi.org/10.1093/bioinformatics/18.suppl_1.S233.
7. Wernicke S. Efficient detection of network motifs. IEEE/ACM Trans Comput Biol Bioinf. 2006;3(4):347–59. https://doi.org/10.1109/TCBB.2006.51.
8. Elbassioni K. A polynomial delay algorithm for generating connected induced subgraphs of a given cardinality. J Graph Algorithms Appl. 2015;19(1):273–80. https://doi.org/10.7155/jgaa.00357.
9. Uno T. Constant time enumeration by amortization. In: Dehne F, Sack J-R, Stege U, editors. Algorithms and data structures. Cham: Springer; 2015. p. 593–605. https://doi.org/10.1007/978-3-319-21840-3_49.
10. Alokshiya M, Salem S, Abed F. A linear delay algorithm for enumerating all connected induced subgraphs. BMC Bioinf. 2019;20(12):319. https://doi.org/10.1186/s12859-019-2837-y.
11. Hakim R, Salem S. Efficiently mining rich subgraphs from vertex-attributed graphs. In: Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. BCB '20. Association for Computing Machinery, New York, NY, USA, 2020. https://doi.org/10.1145/3388440.3412423.
12. Oughtred R, Stark C, Breitkreutz B-J, Rust J, Boucher L, Chang C, Kolas N, O'Donnell L, Leung G, McAdam R, Zhang F, Dolma S, Willems A, Coulombe-Huntington J, Chatr-aryamontri A, Dolinski K, Tyers M. The BioGRID interaction database: 2019 update. Nucleic Acids Res. 2018;47(D1):529–41. https://doi.org/10.1093/nar/gky1079.
13. Subramanian A, Tamayo P, Mootha VK, Mukherjee S, Ebert BL, Gillette MA, Paulovich A, Pomeroy SL, Golub TR, Lander ES, Mesirov JP. Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. Proc Natl Acad Sci. 2005;102(43):15545–50. https://doi.org/10.1073/pnas.0506580102.
14. Liberzon A, Subramanian A, Pinchback R, Thorvaldsdóttir H, Tamayo P, Mesirov JP. Molecular signatures database (MSigDB) 3.0. Bioinformatics. 2011;27(12):1739–40. https://doi.org/10.1093/bioinformatics/btr260.

Publisher's Note