Mining Graphs of Prescribed Connectivity

Natalia Vanetik

Shamoon College of Engineering, Beer Sheva, Israel
natalyav@sce.ac.il

Abstract. Many real-life data sets, such as social, biological and communication networks are naturally and easily modeled as large labeled graphs. Finding patterns of interest in these graphs is an important task, but due to the nature of the data not all of the patterns need to be taken into account. Intuitively, if a pattern has high connectivity, it implies that there is a strong connection between data items. In this paper, we present a novel algorithm for finding frequent graph patterns with prescribed connectivity in large single-graph data sets. We also show how this algorithm can be adapted to a dynamic environment where the data changes over time. We prove that the suggested algorithm generates no more candidate graphs than any other algorithm whose graph extension procedure we employ.

Keywords: Mining graphs, Graph connectivity.

1 Introduction

Representing large complex naturally occurring data structures as labeled graphs has gained popularity in the last decade due to the simplicity of the translation process and because such a representation is intuitive to users. The graph is now a standard format for representing social and biological networks, biochemical and genetic data, and Web and document structure. Frequent subgraphs that represent substructures of the dataset, which are characteristic to that dataset, are considered important and useful indicator of the nature of the dataset. Frequent subgraphs are used to build indices for graph datasets [16] that improve search efficiency, to facilitate classification or clustering for machine learning tasks [14], and to determine normal and abnormal structures within the data [7].

Not all of the frequent subgraphs are usually of interest to the user performing a specific search task, both because of the subgraph meaning in the particular database and because of the high complexity of the graph mining problem. This obstacle becomes especially disturbing when the dataset in question is represented by a single very large labeled graph, such as a Web or a DNA sequence. In this paper, we concentrate on the problem of finding frequent subgraphs that satisfy a user-defined constraint of minimum edge connectivity, that determines how many edges should be removed from a graph in order to separate in into two parts. A minimal edge connectivity requirement allows us to discard frequent graphs that do not characterize strong relations between data items in the native dataset. Moreover, the edge connectivity of a graph can be verified fairly easily (in polynomial time and space), unlike some of the other constraints, such as symmetry, maximum clique size etc.

A. Fred et al. (Eds.): IC3K 2011, CCIS 348, pp. 29–44, 2013.

While there is a number of algorithms for the task of general frequent subgraph mining exist (see, for instance [11]), the issue of finding frequent graphs that are subject to connectivity constraints has rarely been addressed in the literature. In [15], the authors address the issue of mining all closed frequent graphs with predefined edge connectivity and propose two algorithms that handle this problem. The algorithms do not address the issue of frequent patterns that have high connectivity but are not closed.

The authors of [13] have proposed the 'CODENSE' algorithm that finds coherent dense subgraphs – all edges in a coherent subgraph exhibit correlated occurrences in the whole graph set; these graphs naturally have high connectivity.

In this paper, we propose a novel graph mining algorithm that finds frequent subgraphs with a user-specified constraint on edge connectivity. Our algorithm uses the minimum cut structure of a graph in order to perform the task efficiently; this structure can be computed in low polynomial time (even linear, if one uses the randomized algorithm of [9]), which makes our algorithm especially suitable for databases consisting of a single large graph. The mincut structure of a graph also allows us to increase frequent patterns by more than just a node or an edge at a time than the standard approach. We also prove the optimality of this algorithm by showing that every frequent subgraph produced by our algorithm (even if it is only used as a building block for a supergraph satisfying edge connectivity constraints) has to be produced by a competing algorithm.

We present two extensions of our algorithm: the first one focuses on weaker connectivity constraint where frequent subgraphs with at least the required connectivity need to be found. This process can be performed effectively using a simple optimization of the CactusMiningAlgorithm and it is proven to be optimal as well. The second extension allows incremental maintenance of frequent subgraphs with prescribed connectivity in dynamic datasets where edge deletions may happen. Our approach eliminates the need repetitive mining and focuses on candidate subgraph structure instead, saving precious computational time.

This paper is organized as follows. Section 2 contains the basic definitions and graph theoretic facts required for our approach. Section 3 describes the algorithm and contains proofs of the algorithm's correctness. Section 4 contains the proof of algorithm's optimality. Section 5 describes the algorithm for weaker connectivity constraint and Section 6 describes the algorithm for incremental frequent subgraph maintenance.

2 Statement of the Problem

2.1 Basic Definitions

In this paper, we deal with undirected labeled graphs. In a graph $G = (V, E)$, V denotes the node set, $E \subseteq V \times V$ denotes the edge set, and each node $v \in V$ has a label $l(v)$. A graph $G' = (V', E')$ is called a *subgraph* of G, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$ and every edge in E' has both ends in V'. G' is an *induced subgraph* of G if it is a subgraph of G and for every pair of nodes $v, u \in V'$ such that (u, v) is an edge of G, (u, v) is also an edge of G'.

A graph $G = (V, E)$ is *disconnected* if there exists a partition V_1, V_2 of V so that no edge in E has one end in V_1 and another in V_2. If no such partition exists, G is called *connected*. G is called *k-edge-connected*, $k \in \mathbb{N}$, if G is connected and one has to remove at least k edges from E to make G disconnected.

A partition of edge set E into $X \subset E$ and $\overline{X} := E \setminus X$ is called a *cut*. Removing all edges having one end in X and another in $\overline{X}$ (called *$(X, \overline{X})$-edges*) from G disconnects the graph. The *size* of a cut $(X, \overline{X})$ is the number of $(X, \overline{X})$-edges, denoted $|(X, \overline{X})|$. The $(X, \overline{X})$-edges whose removal disconnects the graph are often also called a cut. A cut of minimum size is called a *minimum cut* or a *mincut*. The least size of a cut in a graph is the *edge connectivity* of a graph. In general, for two foreign subsets $X, Y \subset V$ we denote by $|(X, Y)|$ the number of edges in G with one end in X and another in Y.

We study the problem of graph mining in the following setting: our database is a single large undirected labeled graph G. We are given a user-supplied support threshold $S \in \mathbb{N}$ and a connectivity constraint k and we are looking for all k-edge-connected subgraphs of G with a count of at least S (these subgraphs are called *frequent*). The *count* of a graph in a database is determined by a function $count()$ that satisfies the *downward closure property*: for all subgraphs g_1, g_2 of any database graph D such that $g_1 \subseteq g_2$ we always have $count(g_1, D) \geq count(g_2, D)$. The main idea of our approach is to employ the special structure of mincuts in the database graph in order to make the search for frequent k-edge-connected subgraphs faster.

2.2 The Cactus Structure of Mincuts

An unweighted undirected multigraph is called a *cactus* if each edge is contained in exactly one cycle (i.e., any pair of cycles has at most one node in common). Dinitz, Karzanov and Lomonosov showed in [3] that all minimum cuts in a given graph with n vertices can be represented as a cactus of size $0(n)$. This cactus representation plays an important role in solving many connectivity problems, and we use it here for the efficient mining of graphs with connectivity constraints.

Formally, let $G = (V, E)$ be an undirected multigraph and let $\{V_1, ..., V_n\}$ be a partition of V. We denote the set of all minimum cuts of G by $Cuts(G)$. Let $R = (V_R, E_R)$ be a multigraph with node set $V_R := \{V_1, ..., V_n\}$ and edge set $E_R := \{(V_i, V_j) \mid (v_i, v_j) \in E,\ v_i \in V_i, v_j \in V_j\}$.

Definition 1. *R is a **cactus representation** of $Cuts(G)$ if there exists a one-to-one correspondence $\rho : Cuts(G) \to Cuts(R)$ such that for every mincut $(X, \overline{X}) \in Cuts(G)$ holds $\rho((X, \overline{X})) \in Cuts(R)$ and for every mincut $(X, \overline{X}) \in Cuts(R)$ $\rho^{-1}((X, \overline{X})) \in Cuts(G)$.*

Dinitz, Karzanov and Lomonosov [3] have proved that for any undirected multigraph, there exists a cactus representation (in fact, they showed that this is always true for any weighted multigraph). A *dual graph* to any cactus representation, if the cactus cycles are taken as nodes, is a tree. The size of a cactus tree is linear in the number of vertices in the original graph, and any cut can be retrieved from the cactus representation in time linearly proportional to the size of the cut. In addition, the cactus displays explicitly all nesting and intersection relations among minimum cuts. Note that a graph can have at most $\binom{n}{2}$ mincuts, where n is the size of graph's node set. The following definition and a fundamental lemma entirely describe the structure of a cactus representation.

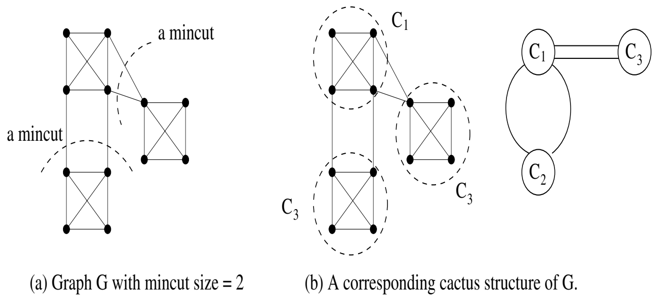


Fig. 1. A cactus structure of a graph

Definition 2. *Let λ be the size of a mincut in graph $G = (V, E)$. A* ***circular partition*** *is a partition of V into $k \geq 3$ disjoint subsets $\{V_1, ..., V_k\}$ such that*

1. $|(V_i, V_j)| = \lambda/2$ *when* $j - i = 1 \mod k$.
2. $|(V_i, V_j)| = 0$ *when* $j - i \neq 1 \mod k$.
3. *For* $1 \leq a < b \leq k$, $\cup_{i=a}^{b-1} V_i$ *is a mincut. Moreover, if any mincut* $(X, \overline{X})$ *is not of this form, then either* X *or* $\overline{X}$ *is contained in some* V_i.

Lemma 1. *[3,1] If X_1 and X_2 are crossing cuts in G (have a non-trivial intersection as sets), then G has a circular partition $\{V_1, ..., V_k\}$ such that each of $X_1 \cap X_2$, $\overline{X_1 \cup X_2}$, $X_1 \setminus X_2$ and $X_2 \setminus X_1$ equal $\cup_{i=a}^{b-1} V_i$ for appropriate choices of a and b.*

Corollary 1. *[3] Every graph has a cactus representation.*

Corollary 2. *[5] Every graph on n vertices has a cactus representation with no more than $2n - 2$ vertices.*

Figure 1 shows a 2-edge-connected multigraph and its cactus representing all three mincuts that exist in the graph. In this example, there is a one-to-one correspondence between the cycles of the cactus and the circular partitions of G.

2.3 Cactus Construction Algorithms

The earliest well-defined algorithm for finding all minimum cuts in a graph uses maximum flows to compute mincuts for all pairs of vertices (see [6]). Karzanov and Timofeev [10] outlined the first algorithm to build a cactus for an unweighted graph. A randomized algorithm by Karger [8] finds all minimum cuts in $O(n^2 \log n)$ time. Fleischer in [5] describes an algorithm that arranges the minimum cuts into an order suitable for a cactus algorithm which runs in $O(nm + n^2 \log n)$ time. Finally, Karger and Panigrahi proposed a near-linear time randomized algorithm in [9].

3 Finding Frequent k-connected Graphs

In this section, we present the CactusMining algorithm for finding all frequent k-connected graphs in a graph database. For simplicity, we assume here that the database is a single large graph; when a database consists of two or more disconnected graphs, graph decomposition and support counting should be performed once for each transaction. The CactusMining algorithm searches for all frequent k-connected subgraphs in a bottom-up fashion and relies on the search space reduction that is implied by the cactus structure of the database.

3.1 Computing the Cactus Structure of a Graph

To compute the cactus structure for a given graph G and a connectivity bound k, we employ a cactus-constructing algorithm, denoted as $BuildCactus()$ (for instance, the one described in [5]).

3.2 Basic Properties

In this section, we describe several useful properties of a cactus mincut structure.

Property 1. Let g be a $(k+1)$-connected subgraph of G. Then g is entirely contained in some V_i, $1 \leq i \leq k$. The converse is not true, i.e. non-k-edge-connected subgraphs of V_i may exist. □

Property 2. Let g be a k-edge-connected subgraph of G. Let C be a minimal subcactus of G with circular partition $\{V_1, ..., V_k\}$ containing g as a subgraph. Then either g contains all the (V_i, V_j)-edges or it contains no such edges.

Proof: This property is trivial since removing a (V_i, V_j)-edge decreases the edge connectivity of a subcactus C containing g.

Corollary 3. *In Property 2, subgraph $g \cap V_i$ contains all the nodes incident to the (V_i, V_j)-edges of a circular partition.*

Proof: Follows from the fact that g contains all (V_i, V_j)-edges.

3.3 Growing Subgraphs

In this section, we describe how an instance of a candidate subgraph can be grown from an existing frequent subgraph instance without violating connectivity constraints.

The intuition behind our subgraph extension approach relies on properties of its location within the cactus structure. Let $T = (V_T, E_T)$ be a dual cactus structure of $(k+1)$-cuts in database D with nodes V_T being the cactus cycles and the adjacency relation E_T determining whether two cactus cycles share a node. Each cactus cycle $C \in V_T$ is a graph, denoted by $C = (V_C, E_C)$, with the structure of a simple cycle. In C, the nodes of V_C are the basic $(k+1)$-connected components of D, i.e. the components that contain no edge of a k-cut in D. Two such components $c_1, c_2 \in V_C$ are adjacent if there exist edges of D that belong to a k-cut and are incident to nodes in c_1 and c_2 (there are precisely $\lceil \frac{k}{2} \rceil$ such edges). To simplify the notation, we say that (c_1, c_2) denotes the set of these edges.

Our goal is to extend instances of frequent graphs gradually, while complying with the following rule:

– do not produce an extension whose cactus structure in C does not ensure k-connectivity.

In order to achieve the objective, our extension procedure depends strongly of the location of an instance within the database cactus structure. Moreover, our approach allows to extend an instance by more than one node.

Let $f \subset D$ be a frequent subgraph instance that we are currently extending. We introduce several additional parameters of f that are updated by our mining algorithm:

1. $f.type$ can assume the values $node$, $cycle$ and $tree$,
2. $f.cycle$ denotes the node $t \in V_T$ containing f as a subgraph (if one exists),
3. $f.tree$ denotes the subtree of T containing f as a subgraph; $|f.tree|$ denotes the number of nodes in the said subtree.

For each value of $f.type$, we propose a separate extension procedure. The first two procedures extend the subgraph instance within its own type; they can fail to extend either because no extension is possible at all or because the type of extension needs to be changed. For $f.type = cycle$ and $f.type = tree$, additional precaution needs to be taken in order to ensure a better search space reduction. In this case, a subgraph of such an instance contained within a $(k+1)$-connected component of the database graph may cause a cut of size less than k to appear in the extended instance. We apply the Contraction() procedure (see Section 3.3) that determines exactly if these subgraphs produce a smaller than required edge cut or not.

The Contraction Procedure. The Contraction() procedure, described in Algorithm 1, receives as an input a subgraph g contained in a $(k+1)$-connectivity component of a cactus structure T, and contracts all the parts of T that are not adjacent or incident to g into single nodes. For each subtree $t \in T$ adjacent to g, $t \setminus g$ is turned into a single node. In fact, every such subtree is turned into a two-node cycle with t and $t \setminus g$ as nodes, and the cycle edges incident to g as edges.

Algorithm 1. Contraction()

Input: subgraph g,
subcactus T containing g.
Output: contraction of g

```
K := a node of T containing g;
for all subtrees t ∈ T incident to K do
    replace t \ {g, cycle edges incident to g}
    with a single node;
end for
return T;
```

Figure 2 gives two examples of applying the Contraction() procedure. The following claim ensures correctness of the procedure.

Claim. Let $T = (V_T, E_T)$ be a cactus structure of k-cuts in a graph G and let a subgraph $f \subset G$ span the $(k+1)$-connectivity components of T and contain all the cycle

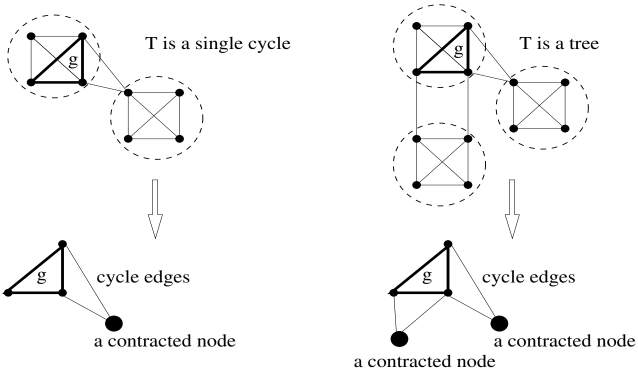


Fig. 2. Contracting subgraphs in $(k+1)$-connected components

edges of the cactus structure. Then there exists a $(k+1)$-connectivity component K of G such that Contraction($f \cap K, T$) is not k-connected if and only if f is not k-connected.

Proof: The "only if" direction is trivial since the contraction described in Algorithm 1 does not reduce the connectivity of f.

For the "if" direction, let us assume an f that is not k-connected. Then there exists a partition V_1, V_2 of its node set so that $|(V_1, V_2)| < k$. Since the number m of cactus components $C_1, ..., C_m$ in T is at least 2, there exists $i \in [1, m]$ s.t. $|(V_1, V_2) \cap C_i < \frac{k}{2}$. Thus, we have a partition U_1, U_2 of a node set of $f \cap C_i$ so that $|(U_1, U_2)| < \frac{k}{2}$.

Let us denote by $f' :=$Contraction($f \cap C_i, T$). Since there are at most k cactus edges incident to $f \cap C_i$, w.l.o.g. there are at most $\frac{k}{2}$ cactus edges incident to U_1 in f', which we denote by E'. Then $(V_1, V_2) \cup E'$ is an edge cut of f' of size less than k, and f' is not k-connected.

Algorithm 2. ExtendNodeType()

Input: Cactus $T = (V_T, E_T)$ of $(k+1)$-cuts in D,
frequent subgraph f.
Output: extensions of f.

$Ext(f)$:=basic-extend(g);
for all $h \in Ext(f)$ **do**
  if $h \cap f.cycle \neq f.cycle$ **then**
    $Ext(f) := Ext(f) \setminus \{h\}$;
  else
    $h.type = node$;
    $h.cycle = f.cycle$;
  end if
end for
return $Ext(f)$;

Extension Procedures. The procedure for $f.type = node$ is described in Algorithm 2. This procedure simply adds a node or an edge to an existing subgraph instance within a $(k+1)$-connectivity component of D. It uses a basic pre-existing extension procedure

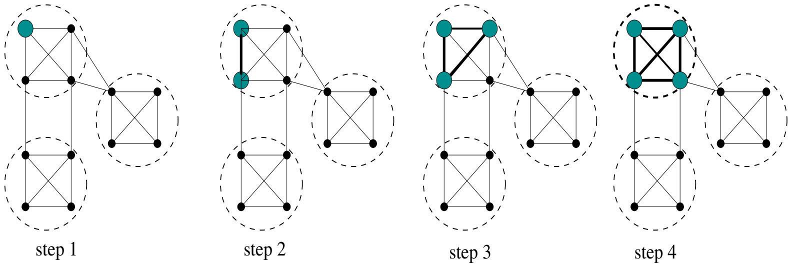


Fig. 3. Extending frequent subgraphs of type *node*

basic-extend(), for instance such as the one in FSG [11]. An example of ExtendNodeType() procedure is given in Figure 3 (extended subgraphs in bold).

The procedure for $f.type = cycle$ is described in Algorithm 3. It extends an instance f of a frequent graph within a single cycle $C = (V_C, E_C)$ that forms a node of the cactus dual tree structure. The main concern is to extend a subgraph so as not to create a not-k-connected instance, and for this purpose the edges E_C must be present in an extension. Therefore, a frequent graph instance that is a subgraph of V_C must be added to f. An example of joining subgraphs of type $node$ into subgraphs of type $cycle$ is given in Figure 4 (extended subgraphs in bold).

Algorithm 3. ExtendCycleType

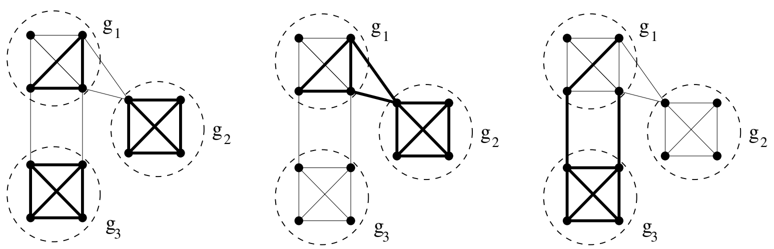
Input: frequent subgraph instances F
of type $node$, frequent subgraph f.
Output: extensions of f.
let $C = (\{c_1, ..., c_n\}, E_C) := f.cycle$;
let $f \subseteq c_j$;
$Ext(f) := \emptyset$;
$E =: \cup_{i,j}\{(c_i, c_j) \in E_C\}$ (as edge sets);
$F_{\text{good}} := F \cap C$; {frequent subgraphs in C}
for all $f \in F_{\text{good}}$ **do**
  $f' :=$ Contraction(f, C);
  if f' **is not** k-connected **then**
    $F_{\text{good}} := F_{\text{good}} \setminus \{f'\}$;
  end if
end for
for all $f_i \in F_{\text{good}} \cap c_i$,
$1 \leq i \leq n, i \neq j$ **do**
  $h := \cup_{1 \leq i \leq n,\, i \neq j} f_i \cup E$;
  if h is a graph **then**
    $Ext(f) := Ext(f) \cup \{h\}$;
    $h.type = cycle$; $h.cycle = f.cycle$;
    $h.tree = h.cycle$;
  end if
end for
return $Ext(f)$;

The final procedure for $f.type = tree$ is described in Algorithm 4. In this case, f is contained in a subtree of the dual cactus tree structure and it is extended by an instance of a frequent subgraph of type $cycle$. In order not to generate the same instance twice, we assume that the tree T is a directed out-tree and that extending a subtree of T by a node is possible only in the direction of T's edges. An example of joining two subgraphs

Algorithm 4. ExtendTreeType

Input: database cactus structure
$T = (V_T, E_T)$,
frequent subgraph instances F
of type $cycle$, frequent subgraph f.
Output: extensions of f.

$T' = (V_{T'}, E_{T'}) := f.tree$;
$Ext(f) := \emptyset$;
for all $C \in V_T$ and $C' \in V_{T'})$ **do**
  if $(C', C) \in E_T$ **then**
    for all $g \in F$ such that $g \cap C = f \cap C$ **do**
      $T'' := T' \cup C$;
      $g' :=$ Contraction$(g \cap C, T'')$;
      if g' **is** k-connected **then**
        $h := f \cup g$;
        $Ext(f) := Ext(f) \cup \{h\}$;
        $h.type = tree$;
        $h.tree = (V_{T'} \cup \{C\}, E_{T'} \cup \{(C', C)\})$;
      end if
    end for
  end if
end for
return $Ext(f)$;



(a) frequent subgraphs of type node

(b) joining g_1 and g_2 into a subgraph of type cycle

(c) joining g_1 and g_3 into a subgraph of type cycle

Fig. 4. Constructing frequent subgraphs of type $cycle$

of type $cycle$ into a subgraph of type $tree$ is given in Figure 5 (extended subgraphs in bold).

3.4 The CactusMining Algorithm

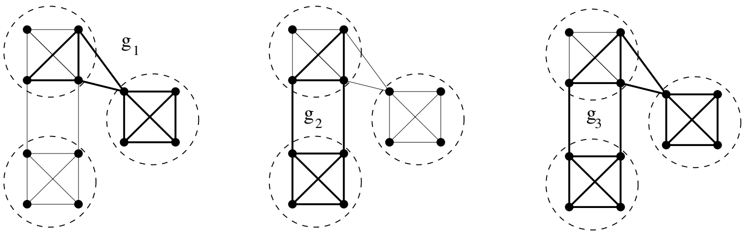
The CactusMining algorithm extends each frequent graph until it spans beyond connectivity component of the database, at which point the extension must include a non-trivial subcactus. If such an extension is not possible, the frequent subgraph must be abandoned. The existence of a counting procedure for subgraphs, denoted count(), is assumed.

3.5 Proof of Correctness

The aim of this section is to show that every maximal frequent k-edge-connected subgraph g of G is generated by the above algorithm at some point (completeness), and no subgraph that is not k-edge-connected is added to a candidate set (soundness).

Claim. The CactusMining algorithm is sound.

Proof: This claim is trivial since step 39 of Algorithm 5 filters out all not-k-connected graphs. □



(a) subgraph g_1 of type=cycle.

(b) subgraph g_2 of type=cycle.

(a) subgraph g_3 of type=tree ersulting from join of g_1 and g_2

Fig. 5. Constructing a frequent subgraph of type $tree$

Claim. The CactusMining algorithm is complete.

Proof: Let G be a frequent k-connected subgraph of D, where D has the k-connectivity cactus structure T. Then every instance of G is contained in a k-connectivity component of D. Let us assume that G is not generated by the Algorithm 5 and let G be minimal in the number of nodes and vertices. We show that every instance g of G is generated by one of the procedures ExtendNodeType, ExtendCycleType() or ExtendTreeType().

If $g.type = node$, it is generated by the ExtendNodeType() procedure that filters out nothing. Otherwise, $g.type \in \{cycle, tree\}$ and by Claim 3.3 g is not generated only if g is not-k-connected - a contradiction. □

Algorithm 5. CactusMining()

Input: graph database D, support S, connectivity bound k.
Output: frequent k-edge-connected graphs.

F_1 := frequent nodes of D;
$D := D \setminus \{\text{non-frequent nodes}\}$;
$BuildCactus(D, k-1)$;
$D \leftarrow k$-connectivity components of D;
$T := BuildCactus(D, k)$;
F_1 := frequent nodes; {`type=node`}
$i := 1$;
while $F_i \neq \emptyset$ **do**
  $C_{i+1} := \emptyset$;
  for all $f \in F_i$ **do**
    $C_{i+1} := C_{i+1} \cup$ ExtendNodeType(T, f);
  end for
  $i := i + 1$;
  F_i := frequent graphs from C_i;
end while
$F_{\mathrm{I}} := \cup_{j=1}^{i-1} F_j$;
$F_1 := F_{\mathrm{I}}$; {`type=cycle`}
$i := 1$;
while $F_i \neq \emptyset$ **do**
  $C_{i+1} := \emptyset$;
  for all $f \in F_i$ **do**
    $C_{i+1} := C_{i+1} \cup$ ExtendCycleType(F_{I}, f);
  end for
  $i := i + 1$;
  F_i := frequent graphs from C_i;
end while
$F_{\mathrm{II}} := \cup_{j=1}^{i-1} F_j$;
$F_1 := F_{\mathrm{II}}$; {`type=tree`}
$i := 1$;
while $F_i \neq \emptyset$ **do**
  $C_{i+1} := \emptyset$;
  for all $f \in F_i$ **do**
    $C_{i+1} := C_{i+1} \cup$ ExtendTreeType(T, F_{II}, f);
  end for
  $i := i + 1$;
  F_i := frequent graphs from C_i;
end while
$F_{\mathrm{III}} := \cup_{j=1}^{i-1} F_j$;
remove $< k$-connected graphs from $F_{\mathrm{I}} \cup F_{\mathrm{II}} \cup F_{\mathrm{III}}$;
return $F_{\mathrm{I}} \cup F_{\mathrm{II}} \cup F_{\mathrm{III}}$

4 Candidate Graphs Generated by the Algorithm

In this section, we prove that the CactusMining algorithm is optimal w.r.t. the set of candidate subgraphs generated by it. We show that any algorithm based on pattern extension must produce a superset of candidate subgraphs generated by the Cactus-Mining algorithm.

Theorem 1. *Let AnyAlgorithm be a graph mining algorithm based on pattern extension. Then CactusMining algorithm produces no more candidates than AnyAlgorithm.*

Proof: We show that any frequent candidate subgraph g produced by the CactusMining algorithm has to be produced by AnyAlgorithm. We denote the database graph by D and its cactus tree structure by $T = (V_T, E_T)$. We denote the $(k+1)$-connectivity of the cactus structure by $C_1, ..., C_n$. Let us assume that g is produced by the CactusMining algorithm but not by AnyAlgorithm as a candidate subgraph.

If $g \subseteq g'$, where g' is a k-connected frequent graph in D, then g is produced by AnyAlgorithm (we can assign labels to g''s nodes so that g is the lexicographically minimal extension of g) – a contradiction. Therefore, g is neither k-connected nor is it a subgraph of a frequent k-connected graph in D. Let therefore V_1, V_2 be a partition of g's nodes so that $|(V_1, V_2)| < k$ (i.e. an edge cut of size $< k$ separates V_1 and V_2). We assume first that g has a non-empty intersection with cactus nodes $C_1, ..., C_m$, where $m > 1$. Then g contains all the cactus edges connecting $C_1, ..., C_m$, for otherwise it is not produced by the CactusMining algorithm (as $g.type = cycle$ or $g.type = node$). We denote the edge set (V_1, V_2) separating g by $E_{1,2}$. Since $m > 1$ and $|E_{1,2}| < k$, there exists i s.t. $|C_i \cap E_{1,2}| < \frac{k}{2}$. Thus, we have a partition U_1, U_2 of a node set of $g \cap C_i$ so that $|(U_1, U_2)| < \frac{k}{2}$. Let g' :=Contraction(g, C_i). Since there are at most k cactus edges incident to $g \cap C_i$, w.l.o.g. there are at most $\frac{k}{2}$ cactus edges incident to U_1 in g', which we denote E'. Then $E_{1,2} \cup E'$ is an edge cut of g' of size less than k – in contradiction to step 8 of Algorithm 3 or to step 8 of Algorithm 4. Then g is not produced by the CactusMining algorithm – a contradiction.

Let us assume now that $g \subseteq C_i$ for some $i \in [1, m]$. Since g is minimal in the node and edge set, there exists a node or an edge x such that $g - x$ is either a k-connected frequent graph in D or is a subset of a k-connected frequent graph g' in D. We can assume that AnyAlgorithm is locally optimal, and, since Algorithm 2 can use any function as basic-extend(), that the same function is used. Therefore, g is produced by both AnyAlgorithm and the CactusMining algorithm – a contradiction. □

5 Weaker Connectivity Constraints

Sometimes, users of graph mining systems (network analysts or biologists) may not know the precise connectivity of frequent subgraphs in a large dataset that they look for. Often, only the minimal requirement of connectivity equal to or bigger than the specified constraint is known. In his case, the CactusMining algorithm can be easily modified into an algorithm that finds all such patterns. In fact, if we are given a connectivity bound k and we are looking for all frequent subgraphs that are *at least k-connected* in dataset D, all these subgraphs fall into two categories:

– subgraphs can be $(k + 1)$-connected and are located inside cactus nodes of k-connectivity cactus structure of D;

– subgraphs that are k-connected but not $(k+1)$-connected that span the cactus edges of D's cactus structure. This property follows from the fact that any frequent subgraph of type $tree$ or $node$ has a k-cut.

In order to find patterns that are at least $(k+1)$-connected we modify the CactusMining algorithm in order to focus the search process on subgraphs within cactus nodes of dataset cactus structure. The algorithm ExtendedCactusMining is presented in Algorithm 6.

The ExtendedCactusMining algorithm, much like CactusMining algorithm, produces the subset of candidate subgraphs of any other mining algorithm. Indeed, if a subgraph has connectivity at least $(k+1)$ that is can only be of type=$node$ for otherwise it contains a k-cut. Inside cactus nodes, both algorithms employ the most efficient extension procedure than it available. It is necessary to produce all the candidate patterns inside cactus nodes as such a node can be, for instance, a complete graph of large order. We obtain the following simple corollary from Theorem 1.

Corollary 4. *Let AnyAlgorithm be a graph mining algorithm based on pattern extension. Then ExtendedCactusMining algorithm produces no more candidates than AnyAlgorithm.* □

Algorithm 6. ExtendedCactusMining()

Input: graph database D, support S, connectivity bound $k+1$.
Output: frequent graphs that are at least $(k+1)$-connected.

$F_1 :=$ frequent nodes of D;
$D := D \setminus \{$non-frequent nodes$\}$;
$T := BuildCactus(D, k)$;
$D \leftarrow (k+1)$-connectivity components of D;
$F_1 :=$ frequent nodes; {`type=node`}
$i := 1$;
while $F_i \neq \emptyset$ **do**
    $C_{i+1} := \emptyset$;
    for all $f \in F_i$ **do**
        $C_{i+1} := C_{i+1} \cup$ ExtendNodeType(T, f);
    end for
    $i := i + 1$;
    $F_i :=$ frequent graphs from C_i;
end while
$F = \bigcup_i F_i$;
remove $< (k+1)$-connected graphs from F;
return F

6 Connectivity Constraints and Dynamic Datasets

The next issue we address is the fact that many single-graph datasets have dynamic nature, i.e. edges may appear and disappear over time. Social networks, web and communication/routing networks are natural examples; protein structure also changes over time while protein folds or moves from one conformation to another. A trivial approach to handling the task of finding all frequent k-connected subgraphs in a dynamic dataset would be to run the mining algorithm anew every time a change occurs. This approach, however, is computationally infeasible. We suggest an easier solution, keeping in mind the fact that computing the cut structure of a pattern is an easy polynomial task. Our approach consists of following steps.

- We run the CactusMining algorithm once and keep the set of candidate subgraphs and their instances in the database for future use. One should notice that the instances are usually kept in the system anyway since they are used by data miners and analysts for insight and verification.
- For each k-connected candidate subgraph we compute its k-connectivity structure (the cactus).
- When a change occurs in the dataset, we look at the exact place this change happened and act accordingly. Our algorithm handles edge deletion only, since addition of an edge may cause subgraphs previously discarded by the mining process to become valid again. In this case, a re-computation may be required.

The DynamicCactusMining algorithm depicted in Algorithm 7 describes a simple procedure for incremental maintenance of frequent k-connected patterns in case of edge deletion. The DynamicCactusMining algorithm performs no mining; it uses previously computed cactus structure of frequent subgraph instances, denoted by $f.cactus$, in order to determine whether the change is dangerous or not. In the former case, the instance needs to be discarded while in the latter case the instance is still k-connected. However, the subgraph represented by this instance may change. If this happens, the count of subgraph in question may need to be updated. Thus, even when k-connectivity is preserved, frequency may change and so can the cactus structure of an instance.

Algorithm 7. DynamicCactusMining()

Input: instances of frequent k-connected subgraphs I,
deleted edge $e = (v, u)$,connectivity bound k, support S.
Output: frequent k-edge-connected graphs.

$NewI := \emptyset$;
for all $f \in I$ **do**
    $C := f.cactus$;
    if $u, v \in$ same node of C **then**
        $f := f \setminus e$;
        $count(f)$++;
        $count(f \cup e)$- -;
        $NewI := NewI \cup \{f\}$;
        $f.cactus$:=*BuildCactus*(f, k);
    end if
end for
$F' := \emptyset$.
for all $f \in NewI$ **do**
    if $count(f) \geq S$ **then**
        $F' := F' \cup \{f\}$;
    end if
end for
return F'.

Claim. DynamicCactusMining algorithm finds all k-connected patterns in the dataset with edge e removed.

Proof: Let our dataset be denoted D. Each k-connected candidate subgraph in $D \setminus e$ is a candidate subgraph in D since edge deletion can only decrease connectivity. The case where a candidate subgraph f is k-connected but $f \setminus e$ is not can only happen when the edge e is a cactus edge. Indeed, if $f \setminus e$ is not k-connected, then e lies in a k-cut X, which by definition contains only edges from $f.cactus$. Moreover, since every edge $g \in f.cactus$ lies in some k-cut X of f, removing e from f creates a cut $X \setminus e$ in $f \setminus e$ of size $(k-1)$. Therefore, lines 5–9 of the DynamicCactusMining algorithm keep a subgraph instance if and only if it remains k-connected. Since the graph changes after edge deletion, instance counts need to be updated as it happens in lines 6–7 of the algorithm. Finally, due to line 14 only frequent subgraphs are kept in the final set. □

7 Conclusions

In this paper, we have presented the CactusMining algorithm for mining frequent k-edge connected subgraphs in a graph database, where k is a user-defined integer constant. The method presented here is defined and described for a single graph database case, but is adapted trivially to multiple graph databases. Our method relies on the Dinitz-Karzanov-Lomonosov cactus minimum cut structure theory and on the existence of efficient polynomial algorithms that compute this structure. Our algorithm employs the pattern-growth approach, and the cactus structure of mincuts allows us to grow frequent subgraphs by more than a node or an edge at a time. We have proved that the CactusMining algorithm is sound and correct, and have also shown that the set of frequent patterns it produces is the least possible, i.e. a competing graph mining algorithm will produce all the candidate patterns that our algorithm produces. We demonstrated how our approach can be adapted to the case of weaker connectivity constraints (ExtendedCactusMining algorithm) and the case of dynamic dataset where edge deletion happen (DynamicCactusMining algorithm).

References

1. Bixby, R.E.: The minimum number of edges and vertices in a graph with edge connectivity n and m n-bonds. Networks 5, 253–298 (1975)
2. De Vitis, A.: The cactus representation of all minimum cuts in a weighted graph. Technical Report 454, IASI-CNR (1997)
3. Dinits, E.A., Karzanov, A.V., Lomonosov, M.V.: On the structure of a family of minimal weighted cuts in a graph. In: Fridman, A.A. (ed.) Studies in Discrete Optimization, pp. 290–306. Nauka, Moscow (1976)
4. Fiedler, M., Borgelt, C.: Support computation for mining frequent subgraphs in a single graph. In: International Workshop on Mining and Learning with Graphs (2007)
5. Fleischer, L.: Building Chain and Cactus Representations of All Minimum Cuts from Hao-Orlin in the Same Asymptotic Run Time. In: Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z. (eds.) IPCO 1998. LNCS, vol. 1412, pp. 294–309. Springer, Heidelberg (1998)

6. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. J. Soc. Indust. Appl. Math. 9(4), 551–570 (1991)
7. Horváth, T., Ramon, J.: Efficient frequent connected subgraph mining in graphs of bounded tree-width. Theor. Comput. Sci. 411(31-33), 2784–2797 (2010)
8. Karger, D.R., Stein, C.: A new approach to the minimum cut problem. Journal of the ACM 43(4), 601–640 (1996)
9. Karger, D.R., Panigrahi, D.: A near-linear time algorithm for constructing a cactus representation of minimum cuts. In: SODA 2009, pp. 246–255 (2009)
10. Karzanov, A.V., Timofeev, E.A.: Efficient algorithms for finding all minimal edge cuts of a nonoriented graph. Cybernetics 22, 156–162 (1986); Translated from Kibernetika 2, 8–12 (1986)
11. Kuramochi, M., Karypis, G.: Frequent Subgraph Discovery. In: ICDM 2001, pp. 313–320 (2001)
12. Nagamochi, H., Kameda, T.: Canonical cactus representation for minimum cuts. Japan Journal of Industrial Appliel Mathematics 11, 343–361 (1994)
13. Papadopoulos, A., Lyritsis, A., Manolopoulos, Y.: Skygraph: an algorithm for important subgraph discovery in relational graphs. Journal of Data Mining and Knowledge Discovery 17(1) (2008)
14. Seeland, M., Girschick, T., Buchwald, F., Kramer, S.: Online Structural Graph Clustering Using Frequent Subgraph Mining. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part III. LNCS, vol. 6323, pp. 213–228. Springer, Heidelberg (2010)
15. Yan, X., Zhou, X.J., Han, J.: Mining Closed Relational Graphs with Connectivity Constraints. In: ICDE 2005, pp. 357–358 (2005)
16. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: EDBT 2009, pp. 192–203 (2009)