Efficient Mining of Frequent Subgraph in the Presence of Isomorphism

Jun Huan, Wei Wang, Jan Prins
Department of Computer Science
University of North Carolina, Chapel Hill
{huan, weiwang, prins}@cs.unc.edu

Abstract

Frequent subgraph mining is an active research topic in the data mining community. A graph is a general model to represent data and has been used in many domains like cheminformatics and bioinformatics. Mining patterns from graph databases is challenging since graph related operations, such as subgraph testing, generally have higher time complexity than the corresponding operations on itemsets, sequences, and trees, which have been studied extensively. In this paper, we propose a novel frequent subgraph mining algorithm: FFSM, which employs a vertical search scheme within an algebraic graphical framework we have developed to reduce the number of redundant candidates proposed. Our empirical study on synthetic and real datasets demonstrates that FFSM achieves a substantial performance gain over the current start-of-the-art subgraph mining algorithm gSpan.

1. Introduction

Frequent pattern mining of structured data such as trees and graphs has attracted much research interest because of its varied applications. Recurring patterns in structured datasets can provide unique insight into the underlying relations being modeled and are the starting point for subsequent researches such as clustering and classification.

A graph is a particularly generic representation that is used in many domains. For example, the chemical structure of a given substance can be modeled by an undirected labeled graph in which each node corresponds to an atom and each edge corresponds to a chemical bond between atoms. In the Predicative Toxicology Evaluation Challenge (PTE) [10], the task is to find frequently occurring substructures that are correlated with toxic substances. Borgelt and Berthold [1] demonstrated the power of the graphical representation and frequent subgraph mining technique for this problem. Other applications of mining patterns from graph databases include video indexing [9], improving storage efficiency of semi-structured databases [3], efficient indexing [4] and web information management [12, 8].

There are on-going efforts to apply the frequent subgraph mining techniques to large and complex datasets. In the current popular applications such as PTE, the graphs are relatively small (on the order of 20 nodes and edges), have low average degree per node (≤ 4), and have a large number of node labels (> 60) [7, 11]. However, there are applications that have more challenging characteristics. Take the task of identifying frequently occurring residue motifs in proteins for example. In this case each graph represents a protein. The nodes of the graphs are residues (amino acids), of which there are only 20 varieties, and there is an unlabeled edge between two nodes when the associated residues are in close proximity. Residues that occur in a similar spatial relationship are thought to have functional significance, and hence an interesting problem is to mine graphical representations of proteins for frequently occurring subgraphs. Compared with the PTE application, the size of the individual graphs is much larger (on the order of 100 - 1000 nodes and edges), the degree per node can be much higher (6 - 20 in some instances), the number of labels is small (20), and the database contains a larger number of graphs.

At the core of any frequent subgraph mining algorithm are two computationally challenging problems 1) subgraph isomorphism: determining whether a given graph occurs in another graph and 2) efficient enumeration of all frequent subgraphs. Generally the number of possible isomorphisms/subgraphs increases with the size and number of graphs in a graph database. To design algorithms that scale to larger databases and

complex graphs, it is imperative to focus on efficient frequent pattern mining and isomorphism algorithms. This is the basic motivation for this work.

1.1 Related Work

Since frequent subgraph mining is computationally challenging, early research in this field focused either on approximation techniques such as SUBDUE [5] or methods applicable mainly for small database like Inductive Logic Programming [2].

Recent subgraph mining algorithms can be roughly classified into two categories. Algorithms in the first category use a level-wise search scheme like Apriori to enumerate the recurring subgraphs. Example algorithms in this category include AGM [6] proposed by Inokuchi *et al.* and FSG [7] proposed by Kuramochi and Karypis. AGM finds all frequent *induced* subgraphs in a graph database. An induced subgraph G' of graph G has nodes $V(G') \subseteq V(G)$ and contains all edges of G connecting nodes in $V(G')$. FSG, on the other hand, finds all frequent *connected* subgraphs in a graph database. As pointed out by [7], unconnected frequent subgraphs can be obtained using a frequent itemset mining algorithm once the frequent connected subgraphs have been identified.

Algorithms in the second category use a depth-first search for Finding candidate frequent subgraphs. Example algorithms in this category include *gSpan* [11], proposed by Yan and Han, and the algorithm of Borgelt *et al.* [1]. Both algorithms find frequent connected subgraphs in a graph database but differ in how they enumerate candidate subgraphs. In gSpan, a frequent subgraph G is *extended* to a candidate frequent subgraph G' by choosing a node v in G and adding an edge (v, w) where w is a node in G or not. Instead of enumerating all the subgraph isomorphisms, the method proposed by [1] keeps a list of all subgraph isomorphisms of a frequent subgraph G. This has several advantages. First, it avoids subgraph isomorphism testing, which generally becomes the performance limiting factor of gSpan when dealing with large and complex (denser graphs with less available labels) graphs. Second, given the list of all subgraph isomorphisms, we can turn previous costly graph operations, such as graph join [7], into much more efficient operations, as described in Section 3.3.

1.2 Our Contributions

In this paper, we present a new algorithm, FFSM (Fast Frequent Subgraph Mining) for finding all frequent connected subgraphs in a graph database. Our method follows the approach of the depth-first search schemes in [11, 1], but incorporates new techniques to improve efficiency.

The key features of our method are: (i) a novel graph canonical form and two efficient candidate proposing operations: FFSM-Join and FFSM-Extension, (ii) an algebraic graphical framework (suboptimal CAM tree) to guarantee that all frequent subgraphs are enumerated unambiguously and (iii) completely avoiding subgraph isomorphism testing by maintaining an embedding set for each frequent subgraph.

In an experimental study we investigate the performance of FFSM on two chemical benchmark datasets, for which [11] reports performance results of gSpan, the current state-of-the-art frequent subgraph mining algorithm. We also assess the performance of FFSM on synthetic graph datasets with different characteristics.

Our experimental study shows that FFSM is competitive with gSpan on all inputs and outperforms gSpan by a factor of seven on a commonly studied chemical compound benchmark.

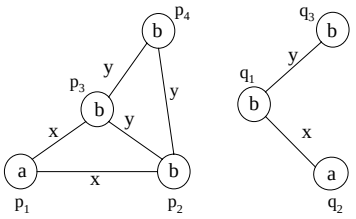
The remainder of this paper is organized as follows. In section 2 we provide some background for graph isomorphism and define the frequent subgraph mining problem. Section 3 presents the major data structure: the canonical adjacency matrix (CAM) representation of (sub)graphs and the canonical adjacency matrix tree. Section 3 also describes the enumeration strategy we developed for frequent subgraph mining. Section 4 presents our empirical performance evaluation of FFSM using both synthetic and real datasets.

2 Background

Definition 2.1 *A **labeled graph** G is a five element tuple $G = \{V, E, \Sigma_V, \Sigma_E, l\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of undirected edges. Σ_V and Σ_E are the sets of vertex labels and edge labels respectively. The labeling function l defines the mappings $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$. Without loss of generality, we assume that there is a total order $\geq$ on each label set Σ_V and Σ_E.*

Definition 2.2 *Given a pair of labeled graphs $G = (V, E, \Sigma_V, \Sigma_E, l)$ and $G' = (V', E', \Sigma'_V, \Sigma'_E, l')$, G is an **subgraph** of G' iff*

- $V \subseteq V'$,
- $\forall u \in V, (l(u) = l'(u))$,
- $E \subseteq E'$,
- $\forall (u, v) \in E, (l(u, v) = l'(u, v))$.



(1) Graph P (2) Graph Q

Figure 1. Examples of two labeled graphs and a subgraph isomorphism. Graph P has four nodes p_1, p_2, p_3, and p_4. Graph Q has three nodes q_1, q_2, and q_3. The labels of the nodes are specified within the circle and the labels of the edges are specified along the edge. Throughout this paper, we assume the following total order $a \geq b \geq x \geq y \geq 0$. The mapping $q_1 \rightarrow p_3$, $q_2 \rightarrow p_1$, $q_3 \rightarrow p_2$ represents a subgraph isomorphism from graph Q to P. There are a total of four subgraph isomorphisms from Q to P, nevertheless the support of Q in $GD = \{P\}$ is 1.

G' is also referred to as a **supergraph** *of G.*

Definition 2.3 *A labeled graph $G = (V, E, \Sigma_V, \Sigma_E, l)$ is* ***isomorphic*** *to another graph $G' = (V', E', \Sigma'_V, \Sigma'_E, l')$ iff there exists a bijection $f : V \rightarrow V'$ such that*

- $\forall u \in V, (l(u) = l'(f(u)))$,
- $\forall u, v \in V, ((u, v) \in E \Leftrightarrow (f(u), f(v)) \in E')$, *and*
- $\forall (u, v) \in E, (l(u, v) = l'(f(u), f(v)))$.

The bijection f is an *isomorphism* between G and G'. We also say that G is *isomorphic* to G' and vice versa.

Definition 2.4 *A labeled graph G is* ***subgraph isomorphic*** *to a labeled graph G', denoted by $G \subseteq G'$, iff there exists a subgraph G'' of G' such that G is isomorphic to G''.*

An example of labeled graphs and a subgraph isomorphism is presented in Figure 1.

Definition 2.5 *Given a set of graphs GD (referred as a* ***graph database****) and a threshold σ ($0 < \sigma \leq 1$), the* support *of a graph G, denoted by sup_G is defined as the fraction of graphs in GD to which G is subgraph isomorphic.*

$$sup_G = \frac{|\{G' \in GD \mid G \subseteq G'\}|}{|GD|}$$

G is ***frequent*** *iff $sup_G \geq \sigma$.*

The ***frequent subgraph mining*** problem is given a threshold σ and a graph database GD, finding all frequent subgraphs in GD.

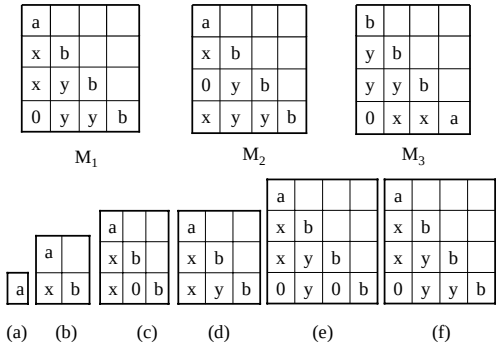


Figure 2. Top: three adjacency matrices for the graph P in Figure 1. After applying the total ordering, we have $code(M_1)$ = "$axbxyb0yyb$" $\geq$ $code(M_2)$ = "$axb0ybxyyb$" $\geq$ $code(M_3)$ = "$bybyyb0xxa$". For adjacency matrix M_1, the edge entry set is $\{m_{2,1}, m_{3,1}, m_{3,2}, m_{4,2}, m_{4,3}\}$ where $m_{2,1}, m_{4,3}$, and $m_{4,2}$ are the first, last, second-to-last edge entries of M, respectively. Bottom: examples of maximal proper submatrices. Matrix 3.a is the proper maximal submatrix of matrix 3.b, which itself is the proper maximal submatrix of 3.c and so forth.

3 Mining Frequent Subgraphs

3.1 Canonical Adjacency Matrix

In FFSM, we represent each graph by an adjacency matrix M such that every diagonal entry of M is filled with the label of the corresponding node and every off-diagonal entry is filled with the label of the corresponding edge, or zero if there is no edge. This is slightly different from the widely used adjacency matrix representation for unlabeled graphs.

One of the critical problems in graph mining is the graph isomorphic problem: given two graphs P and Q, determine whether P is isomorphic to Q. We solve this problem following a common theme such that first transforming each graph into its unique presentation (referred to as the *canonical form* of a graph) and secondly, comparing the transformed presentations. The canonical form is specially designed s.t. if two graphs are isomorphic to each other, their canonical forms are the same and vice versa. We will elaborate the canonical form further in the subsequent discussion.

Following convention, we use a capital letter to denote an adjacency matrix and use the corresponding lower case letter (augmented with subscripts) to denote an individual entry of the adjacency matrix. For instance, we use $m_{i,j}$ to denote the entry on the ith row and jth column of the adjacency matrix M, where $0 < j \leq i \leq n$. Note that the adjacency matrix is not

unique for a given graph. Since each diagonal entry represents a vertex in the graph, each permutation of the set of vertices corresponds to a different adjacency matrix. There may be $n!$ different adjacency matrices for a graph of n vertices. Figure 2 (top) shows three adjacency matrices for the labeled graph P in Figure 1[1]. In order to enable a unique representation for each graph, we define the code of adjacency matrices which provides a total order among all adjacency matrices.

Definition 3.1 *Given an $n \times n$ adjacency matrix M of a graph G with n vertices, we define the* **code** *of M, denoted by code(M), as the sequence formed by concatenating lower triangular entries of M (including entries on the diagonal) in the order: $m_{1,1}m_{2,1}m_{2,2}...m_{n,1}m_{n,2}...m_{n,n-1}m_{n,n}$ where $m_{i,j}$ is the entry at the ith row and jth column in M ($0 < j \leq i \leq n$). We assume that the rows in M are numbered 1 through n from top to bottom and the columns are numbered 1 through n from left to right.*

For an adjacency matrix M, each diagonal entry of M is referred to as a *node entry* and each off-diagonal none-zero entry in the lower triangular part of M is referred to as an *edge entry*. We order edge entries according to their relative positions in the code of the matrix and refer to the *first* edge entry of M as the left-most edge entry that appears in *code*(M) and the *last* edge entry as the one appears rightmost in *code*(M).

Figure 2 shows codes and edge entries for the labeled graph P showing in Figure 1.

We use standard lexicographic order on sequences to define a total order of two arbitrary codes p and q. Given a graph G, its *canonical form* is the maximal code among all its possible codes. The adjacency matrix M which produces the canonical form is defined as G's *canonical adjacency matrix* (CAM). For example, the adjacency matrix M_1 shown in Figure 2 is the CAM of the graph P shown in Figure 1, and *code*(M_1) is the canonical form of the graph.

Notice that we use maximal code rather than the minimal code used by [7, 6] in the above canonical form definition. This definition provides important properties for subgraph mining, as explained below.

3.2 CAM's Suboptimal Property and the CAM Tree

In this section, we focus on a single undirected connected graph. We return to unconnected graphs and graph databases in section 3.4.

[1]Only the lower triangular part of an adjacency matrix is drawn since the upper half is a mirror of the lower half

A key property of the canonical form is that a "prefix" of the canonical form is also maximal, which is stated in the following theorem.

Theorem 3.1 *Given a connected graph G and a subgraph H of G, let A and B be the CAMs of G and H respectively. We have code(A) $\geq$ code(B).*

Given a $n \times n$ matrix N and a $m \times m$ matrix M, let $m_{l,k}$ be the last edge entry of M, and assume that M has at least two edge entries in the last row. N is the *maximal proper submatrix* of M iff:

$n = m$, and

$$n_{i,j} = \begin{cases} m_{i,j} & 0 < i,j \leq n \wedge (i \neq l \wedge j \neq k) \\ & \wedge (i \neq k \wedge j \neq l) \\ 0 & \text{otherwise} \end{cases}$$

Similarly, if M has only one edge entry in the last row, N is the *maximal proper submatrix* of M iff $n = m-1$, and $n_{i,j} = m_{i,j}$ $(0 < i,j \leq n)$.

Since M represents for a connected graph, it is not necessary to consider a situation where there is no edge entry in the last row of M.

Several examples of the maximal proper submatrices are given at the bottom of Figure 2.

Corollary 3.2 *Given a CAM M of a connected graph G and M's submatrix N, N represents a connected subgraph of G.*

Proof outline: since N must represent a subgraph of G, it is sufficient to show the subgraph N represents is connected. To prove this, it is sufficient to show that in N there is no such row i (with the exception of the first row) that i doesn't contains any edge entry. The detailed proof is in Appendix A.1.

Corollary 3.3 *Given a connected graph G with CAM M, M's submatrix N, and a graph H which N represents, N is the CAM of H.*

Proof outline: Let's first assume M has only one edge entry in the last row. It is trivial to show that *code*(N) is a prefix of *code*(M). According to theorem 3.1, we can not find a subgraph (including the graph N represents) such that its code might be greater than any equal length prefix of *code*(M). This suggests that we have *code*(N) $\geq$ *code*($CAM(H)$), where $CAM(H)$ stands for the canonical adjacency matrix of graph H. Therefore N must be the CAM of the graph H.

For a CAM with at least two edge entries in the last row, a similar proof could be used. Interested reader might verify this result using the examples we give out at Figure 2.

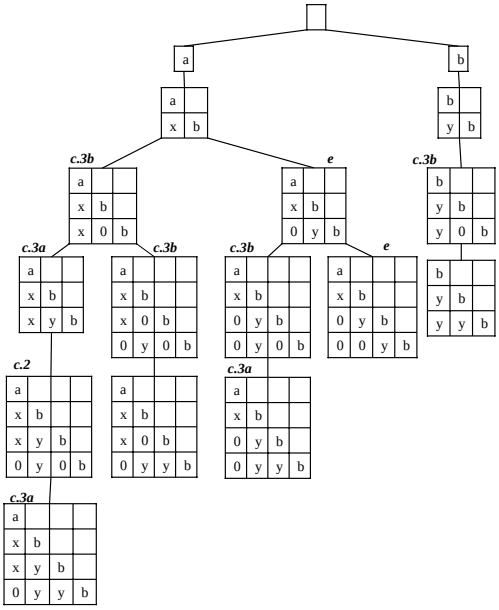


Figure 3. The CAM Tree of the graph P in Figure 1. Every matrix obtained by a join operation is specified by a label starting with **c.** and then the type of the join operation e.g. **c.3a** stands for join case3a. A CAM obtained by an extension operation is labeled with **e**. The join and extension operations are discussed in section 3.3.1 and 3.3.2, respectively. CAMs (size ≥ 3) without label are explained in section 3.3.3 where suboptimal CAMs are discussed. CAMs with one/two edges are obtained by an initial step (discussed in section 3.4) which involves directly scanning nodes/edges in a graph database and hence no label provided.

If we let an empty matrix be the maximal proper submatrix of any matrix with size 1, we can organize all the CAMs of connected subgraphs of a graph G into a rooted tree as follows:

- The root of the tree is an empty matrix;
- Each node in the tree is a distinct connected subgraph of G, represented by its CAM;
- For a given none-root node (with CAM M), its parent is the graph represented by M's maximal proper submatrix;

The tree obtained in this fashion is denoted as the *CAM tree* of the graph G. Figure 3 shows the CAM tree of the graph P from Figure 1.

3.3 Exploring the CAM Tree: Join, Extension and Suboptimal CAMs

The current methods for enumerating all the subgraphs might be classified into two categories: one is the join operation adopted by FSG and AGM [6, 7]. A join operation takes two joinable frequent k-edge graphs G_1 and G_2 and produces a $(k+1)$-edge graph candidate G such that both G_1 and G_2 are subgraphs of G. Two k-edge graphs are *joinable* if they share a common $(k-1)$-edge subgraphs. The join operation is expensive, as shown in [7], in that a single join operation might generate multiple graph candidates and one candidate might be redundantly proposed by multiple distinct join operations.

On the other hand, [1, 11] use an extension operation to grow an existing frequent graph. An extension operation produce a $(k+1)$-edge graph candidate from a frequent k-edge graph G by adding an additional edge to G (with/without introducing an additional node). This operation is also costly since for a given graph, there are many nodes in the graph that an additional edge might be attached to.

Keeping all the existing algorithms in mind, we list some of the key design challenges to achieve efficient subgraph enumeration:

(i) Can we design a join operation such that every distinct CAM is generated only once?

(ii) Can we improve the join operation such that only a few CAMs can be generated from a single join operation (say at most two)?

(iii) Can we design an extension operation such that all the edges might be attached to only one node in a graph represented by a CAM?

In order to meet these challenges, we augment the CAM tree with a set of *suboptimal canonical adjacency matrices*, and introduce two new operations: FFSM-Join and FFSM-Extension. These are discussed in the following sections.

3.3.1 FFSM-Join

As discussed before, a join operation is "superimposing" two graphs to generate a new graph candidate. Depends on the different characteristics of the graphs, the join operation might produce one or two graph candidates. Giving the details of the join operation is the main focus in this section.

Given an adjacency matrix A of a graph G, we denote A as an "*inner*" matrix iff A has at least two edge entries in the last row. Otherwise, we denote A

as a "*outer*" matrix. Given two adjacency matrices A ($m \times m$) and B ($n \times n$) sharing the same maximal proper submatrix, let A's last edge be $a_{m,f}$ and B's last edge be $b_{n,k}$. We define *join(A, B)* by the following three cases [2]:

join case 1: both A and B are inner matrices

1: **if** $f \neq k$ **then**
2: $join(A, B) = \{C\}$ where C is a $m \times m$ matrix and

$$c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m, i \neq n \wedge j \neq k \\ b_{i,j} & \text{otherwise} \end{cases}$$

3: **else**
4: $join(A, B) = \Phi$
5: **end if**

join case 2: A is an inner matrix and B is an outer matrix

$join(A, B) = \{C\}$ where C is a $n \times n$ matrix and

$$c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m \\ b_{i,j} & \text{otherwise} \end{cases}$$

join case 3: both A and B are outer matrices

1: let matrix D be a $(m+1) \times (m+1)$ matrix where (case 3b)

$$d_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m \\ b_{m,j} & i = m+1, 0 < j < m \\ 0 & i = m+1, j = m \\ b_{m,m} & i = m+1, j = m+1 \end{cases}$$

2: **if** $(f \neq k \wedge a_{m,m} = b_{m,m})$ **then**
3: C is $m \times m$ matrix where (case 3a)

$$c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m, i \neq n \wedge j \neq k \\ b_{i,j} & \text{otherwise} \end{cases}$$

4: $join(A, B) = \{C, D\}$
5: **else**
6: $join(A, B) = \{D\}$
7: **end if**

In join case 3, when joining two outer matrices M_1 and M_2 (both with size m), we might obtain a matrix with the same size. We refer this join operation as *case3a*. It is also possible that we obtain a matrix having size $(m+1)$ and this case is referred as *case3b*.

We notice that the join operation is symmetric w.r.t. A and B with only one exception join case 3b. In another word, join(A, B) = join(B, A) in join case 1, 2 and 3a and join(A, B) ≠ join(B, A) in join case3b. In order to remove the potential duplications resulting from this symmetry, we require that $code(A) \geq code(B)$ in all join cases except join case 3b. Equality is permitted since self-join is a valid operation. If the inequality is not satisfied ($code(A) < code(B)$), a join operation produces an empty set.

[2] we only define the lower triangular part of the matrix by the same reason we stated before

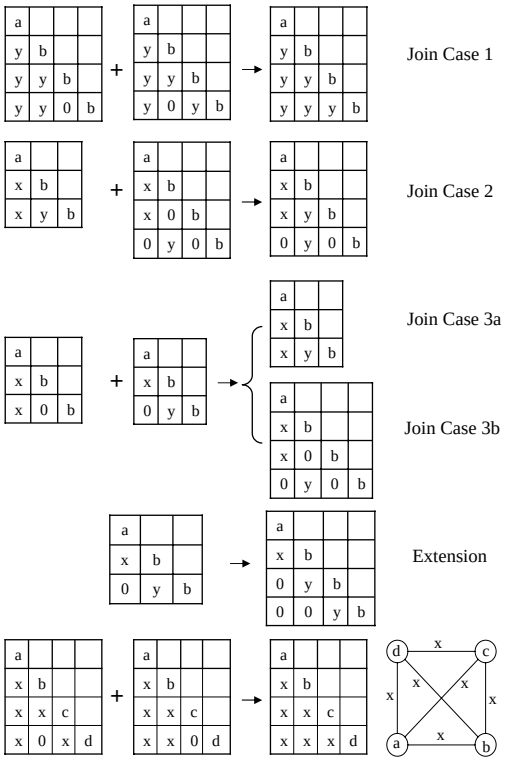


Figure 4. Examples of the join/extension operation

Figure 4 shows examples for the join operation for the four cases (case1, case2, case3a and case3b). At the bottom of Figure 4 shows a case in which a graph might be redundantly proposed by FSG $\binom{6}{2} = 15$ times (joining of any two distinct five-edge subgraphs G_1, G_2 of the graph G will restore G by the join operation proposed by FSG). As shown in the graph, FFSM-Join completely removes the redundancy after "sorting" the subgraphs by their canonical form.

However, the join operation is not "complete" in the sense that it may not enumerate all the subgraphs in the CAM tree. Interested readers might find such examples in the CAM tree we presented in Figure 3. Clearly we need another operation, which is discussed below.

3.3.2 FFSM-Extension

Another basic enumeration technique is an extension operation which involves proposing a k+1-edge graph candidate G from a k-edge graph G_1 by introducing one additional edge. The newly introduced edge might connect two existing nodes or connect an existing node and a node introduced together with the edge. A simple way to perform the extension operation is to introduce every possible edge to every node in a graph G. This method clearly has the complexity of $O(\Sigma_V \times \Sigma_E \times |G|)$ where Σ_V, Σ_E stand for the set of available vertex and edge labels for a graph G, respectively. It suffers from the large size of graph candidates as well as the large amount of available node/edge labels.

gSpan [11] developed an efficient way to reduce the total number of nodes need to be considered. In gSpan, the extension operation is only performed to nodes on the "rightmost path" of a graph. Given a graph G and one of its depth first search trees T, the *rightmost path* of G with respect to T is the rightmost path of the tree T. gSpan chooses only one depth first search tree T which produces the canonical form of G for extension. We refer to [11] for further details about the extension operation.

In FFSM, we further improve the efficiency by always choosing a single fixed node in a CAM and attach an newly introduced edge to it together with an additional node. As proved by Theorem **3.4**, this extension operation, combined with the join operation, unambiguously enumerates all the nodes in the CAM tree.

In the following pseudo code for the extension operation.

FSM-Extension [3]

input: a $n \times n$ adjacency matrix A
output: a set S of adjacency matrices B (with size $(n+1)$) s.t. A is B's maximal proper submatrix.

1: **if** (A is an outer adjacency matrix) **then**
2: **for** $(n_l, e_l) \in \Sigma_V \times \Sigma_E$ **do**
3: create an $n \times n$ matrix B s.t.
4:

$$b_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq n \\ 0 & i = n+1, 0 < j < n \\ e_l & i = n+1, j = n \\ n_l & i = n+1, j = n+1 \end{cases}$$

5: $S \leftarrow S \cup \{B\}$
6: **end for**
7: **else**
8: $S \leftarrow \Phi$
9: **end if**

[3] This is not fully optimized yet since we still enumerate set $\Sigma_V \times \Sigma_E$. For an optimized version of this operation, refer to section 3.5 after we introduce embedding set

3.3.3 Suboptimal CAM Tree

Using the CAM tree of the graph P in Figure 3, we can see that join and extension operations, even combined together, can not enumerate all subgraphs in P. We investigated this and found this problem can be solved by introducing the suboptimal canonical adjacency matrices, as defined below.

Definition 3.2 *Given a graph G, a **suboptimal Canonical Adjacency Matrix** (simply, suboptimal CAM) of G is an adjacency matrix M of G such that its maximal proper submatrix N is the CAM of the graph N represents.*

By definition, every CAM is a suboptimal CAM (Corollary 3.3). We denote a *proper suboptimal CAM* as a suboptimal CAM that is not the CAM of the graph it represents. Several suboptimal CAMs (the matrices with dotted boundaries) are shown in Figure 5. Clearly, all the suboptimal CAMs of a graph G could be organized in a tree in a similar way to the construction of the CAM tree. One such example for the graph P in Figure 1 is shown in Figure 5.

With the notion of suboptimal CAM, the suboptimal CAM tree is "complete" in the sense that all vertices in a suboptimal CAM tree can be enumerated using join and extension operations. This is formally stated in the following theorem.

Theorem 3.4 *For a graph G, let $C_{k-1}(C_k)$ be set of the suboptimal CAMs of all the $(k-1)$-vertex (k-vertex) subgraphs of G ($k \geq 3$). Every member of set C_k can be enumerated unambiguously either by joining two members of set C_{k-1} or by extending a member in C_{k-1}.*

Proof outline: let A be a $m \times m$ suboptimal CAM in set C_k. We consider the following five cases according to the edge entries in A's last row and second-to-last row:

- TypeA M has three or more edge entries in the last row;
- TypeB M has exactly two edge entries in the last row;
- TypeC M has exactly one edge entry in the last row and more than one edge entries in the second-to-last row;
- TypeD M has exactly one edge entry $e_{m,n}$ in the last row and one edge entry in the second-to-last row and $n \neq m-1$;

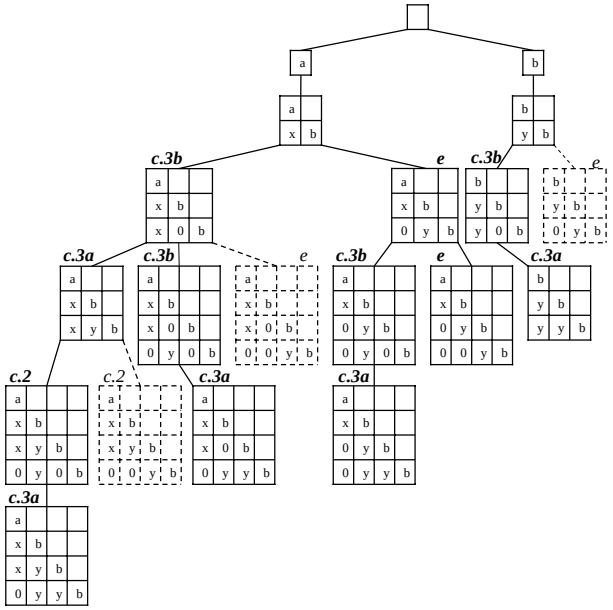


Figure 5. the Suboptimal CAM Tree for the graph P showing in the Figure 1. Matrices with solid boundary are CAMs and those with dashed line boundary are proper suboptimal CAMs. The label on top of an adjacency matrix M indicates the operation by which M might be proposed from its parent. The labeling follows the same way as we present in the Figure 3

- TypeE M has exactly one edge entry $e_{m,n}$ in the last row and one edge entry in the second-to-last row and $n = m - 1$;

As shown in the Appendix, a TypeA suboptimal CAM might be produced by two suboptimal CAMs following the join case1. Similarly, a TypeB suboptimal CAM might be produced following the join case3a, TypeC corresponds to join case2, TypeD corresponds to join case3b, and TypeE corresponds to the extension operation. The formal proof of the completeness of the suboptimal CAM tree is lengthy and is deferred to Appendix A.2.

3.4 Frequent Subgraph Mining of a Graph Database

In the above discussions, we present a novel data structure (CAM tree) for organizing all (connected) subgraphs of a single connected undirected graph. This, however, can be easily extended to a set of graphs (connected or not) (denoted as a graph database) and a single CAM tree can be built for a graph database. If we have such a tree built in advance (regardless of the required space and computational capacity), any traversal of the tree reveals the set of distinct subgraphs of the graph database. For each of such subgraph, its support can be determined by a linear scan of the graph database and therefore frequent ones can be reported. This method clearly suffers from the huge number of available subgraphs in a graph database and therefore very unlikely scale to large databases.

In the following pseudo code, we present an algorithm which takes advantage of the following simple fact: if a subgraph G is not qualified for frequent (support of G is less than a user posted threshold), none of its supergraphs are qualified. This suggest that we can stop building a branch of the tree as early as we find the current node doesn't have sufficient support.

In the pserudo code below, symbol $CAM(G)$ denotes the CAM of the graph G. $X.isCAM$ is a boolean variable indicate whether the matrix X is the CAM of the graph it represents or not.

FFSM

input: a graph database GD and a support threshold f $(0 < f \leq 1)$
output: set S of all G's connected subgraphs.

1: $S \leftarrow$ { the CAMs of the frequent nodes and edges }
2: $P \leftarrow$ { the CAMs of the frequent edges }
3: FFSM-Explore(P, S);

FFSM-Explore
input: U, a suboptimal CAM list and W, a set of frequent connected subgraphs' CAMs
output: set W contains CAMs of all frequent subgraphs searched so far.

1: **for** $X \in P$ **do**
2: **if** $(X.isCAM)$ **then**
3: $W \leftarrow W \cup \{X\}, C \leftarrow \Phi$
4: **for** $Y \in P$ **do**
5: $C \leftarrow C \cup$ FFSM-Join(X, Y)
6: **end for**
7: $C \leftarrow C \cup$ FFSM-Extension(X)
8: remove CAM(s) from C that is either infrequent or not suboptimial
9: FFSM-Explore(C, W)
10: **end if**
11: **end for**

3.5 Performance Enhancement using an Embedding List

One of the key challenges for efficient subgraph enumeration is the scheme to efficiently count the support

of a particular subgraph G. A brute force way to do this is to perform subgraph matching and count how many graphs in a graph database are the supergraph of G. This scheme involves a subgraph matching procedure which is known to be NP-complete in the general case.

From a practical point of view, the above scheme can be significantly sped up by recording the previous matched vertices (denoted as an *embedding*) and basing the subsequent search on the previous recorded information. To further explain this point, we define an embedding as follows:

Definition 3.3 *Given an arbitrary $n \times n$ adjacency matrix A and a labeled graph $G = (V, E, \Sigma_V, \Sigma_E, l)$, a vertex list $L = u_1, u_2, \ldots, u_n \subset V$ is an* ***embedding*** *of A in G iff:*

(i) $\forall\ i, (a_{i,i} = l(u_i))$;

(ii) $\forall\ i, j (a_{i,j} \neq 0 \Rightarrow a_{i,j} = l(u_i, u_j))$;

where $0 < j < i \leq n$.

The set of all embeddings of a matrix in a database is defined as its *embedding set*. Given two suboptimal CAMs and their embedding sets, we could modify the join and extension operations we presented before to let them not only propose candidates but also calculate candidates' embedding sets. Take the join case 1 for example. Given two inner suboptimal CAMs P and Q, a suboptimal CAM $\{A\} = join(P, Q)$, and a list L of nodes in a graph G which is a embedding of A in G, clearly L is a embedding of both P and Q in G. This is because 1) the condition (i) automatically gets satisfied and 2) Condition(ii) must also hold since A contains all the none-zero entries of both P and Q. On the other hand, if we have an list of nodes L which is an embedding of both P and Q, the same list must be a embedding of A (noticing A contains no more none-zero entries than those found in either P or Q).

From the above analysis, we conclude that for the embedding set O_A of a suboptimal CAM A, which is joined by two suboptimal CAMs P and Q through join case 1, we have $O_A = O_P \cap O_Q$, where O_P and O_Q are the embedding sets of suboptimal CAM P and Q, respectively.

Similarly, for join case 2, ($\{A\} = join(P, Q)$, P is an inner matrix and Q is an outer matrix), we have $O_A = \{L \mid L = u_1, u_2, \ldots, u_{n-2}, u_{n-1}, u_n, L \in O_q, L' = u_1, u_2, \ldots, u_{n-2}, u_{n-1} \in O_P\}$. For join case 3a ($\{A\} = join(P, Q)$, both P and Q are outer matrices, A has the same size as P and Q after join), we have $O_A = O_P \cap O_Q$. For join case 3b ($\{A\} = join(P, Q)$, both P and Q are outer matrices, and A has size one greater than that of P and Q after join), we have $O_A = \{L \mid L = u_1, u_2, \ldots, u_n, L' = u_1, u_2, \ldots, u_{n-3}, u_{n-2}, u_{n-1} \in O_p, L'' = u_1, u_2, \ldots, u_{n-3}, u_{n-2}, u_n \in O_q\}$.

For the extension operation, we present the embedding set calculation as a pseudo code below:

FFSM-Extension-Embedding

input: A suboptimal $n \times n$ CAM A
output: A set S of all suboptimal CAMs extended from A and their embedding sets

1: **for** embedding $L = u_1, u_2, \ldots, u_n \in O_A$ **do**
2: let G be the graph contains vertices in L
3: **for** node $v \in V[G] \wedge v \notin L \wedge (v, u_n) \in E[G]$ **do**
4: create a $(n+1) \times (n+1)$ Matrix B where

$$b_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq n \\ 0 & i = n+1, 0 < j < n \\ l_G(v, u_n) & i = n+1, j = n \\ l_G(v) & i = n+1, j = n+1 \end{cases}$$

5: $O_B \leftarrow O_B \cup \{(g_i, Lv)\}$
6: $S \leftarrow S \cup \{B\}$
7: **end for**
8: **end for**

where l_G is the mapping used by G to map the vertices and edges to their labels. Lv is the list concatenated by list L and a single node v.

4 Experimental Study

We performed our experimental study using a single processor of a 2GHz Pentium PC with 2GB memory, running RedHat Linux 7.3. The FFSM algorithm is implemented using the C++ programming language and compiled using g++ with O3 optimization. The gSpan executable, which was compiled in a similar environment, was kindly provided by X. Yan and J. Han in University of Illinois, .

4.1 Chemical Compound Datasets

We used three chemical compound datasets to evaluate the performance of the FFSM algorithm. The first dataset we used if the PTE [10] which can be downloaded from `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE/`. This dataset contains 337 chemical compounds and each of which is modeled by an undirected graph. There are a total of 66 atom types and four bond types (single, double, triple, aromatic bond) in the dataset. The atoms and bonds information are stored in two separated files and we follow exactly the same procedure described in [11] for building graphs from the dataset.

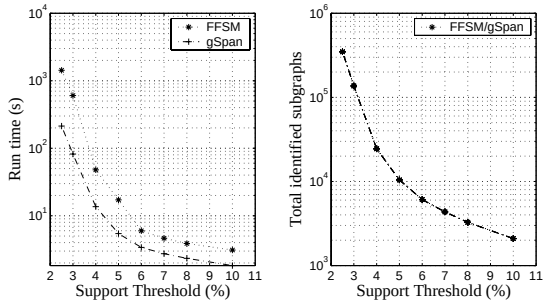


Figure 6. Left: FFSM and gSpan performance comparison under different support values for DTP CM dataset. Right: Total frequent pattern identified by the algorithms.

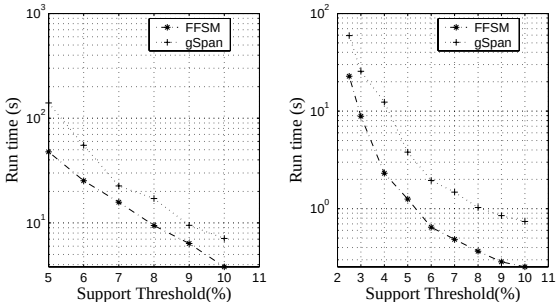


Figure 7. FFSM and gSpan performance comparison under different support values for two different datasets. Left: DTP CA dataset. Right: PTE.

The next two we used are from the DTP AIDS Antiviral Screen dataset from National Cancer Institute. In this dataset, chemicals are classified into three classes: confirmed active (**CA**), confirmed moderately active (**CM**) and confirmed inactive (**CI**) according to experimentally determined activities against AIDS virus. There are total 423, 1083, and 42115 chemicals in the three classes, respectively. For our own purposes, we formed two datasets consisting of all CA compounds and of all CM compounds and refer to them as DTP CA/DTP CM thereafter. The DTP datasets can be downloaded from `http://dtp.nci.nih.gov/docs/aids/aids_data.html`.

We evaluated the performance of FFSM using various support threshold. The result is summarized by the Figure 6 and 7. We can see for DTP CM dataset, FFSM have a maximal 7 fold speedup over gSpan. For DTP CA and PTE dataset, FFSM usually have a 2 to 3 fold speedup from gSpan.

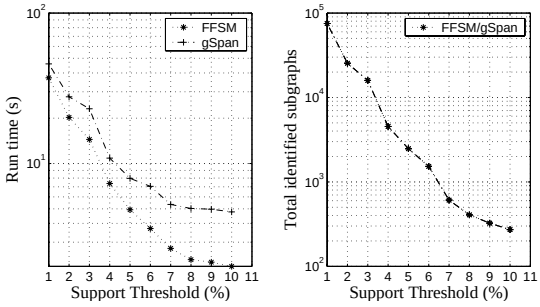


Figure 8. FFSM and gSpan performance comparison under different support value. Parameters used: D10kT20I9L200E4V4.

4.2 Synthetic Datasets

We used a graph generator, kindly offered by M. Kuramochi and G. Karypis in University of Minnesota, to generate all synthetical graph databases with different characteristics. There are six parameters to control the set of graphs generated: (i) $|D|$, total graph transactions generated, (ii) $|T|$, average graph size for the generated graphs, in terms of number of edges, (iii) $|L|$, the total number of the potentially frequent subgraphs, (iv)$|I|$, the size of the potentially frequent subgraphs, in terms of number of edges, (v)$|V|$, total number of available labels for vertices, and (vi)$|E|$, total number of available labels for edges.

We write a single string to describe the parameter settings, e.g. "$D10kT20L200I9V4E4$" represents a synthetic graph database which contains a total of $|D| = 10$k (10000) graph transactions. Each graph averagely contains $|T| = 20$ edges with up to $|V| = 4$ vertex labels and $|E| = 4$ edge labels. There are total of $|L| = 200$ potential frequent patterns in the database with average size $|I| = 9$.

In Figure 8, we show how the FFSM algorithm scales with increasing support. The total number of identified frequent subgraphs is also given.

At the left part of the Figure 9, we show performance comparison between FFSM and gSpan under different average graph sizes (left) or different number of node/edge labels (right). From almost all circumstances, FFSM is faster than gSpan though the value of the speedup varies from dataset to dataset.

5 Conclusion

We presented a new algorithm FFSM for the frequent subgraph mining problem. Comparing to existing algorithms, FFSM achieves substantial per-

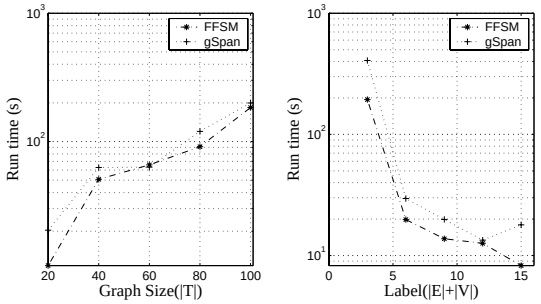


Figure 9. FFSM and gSpan performance comparison under different graph sizes ($|T|$) ranging from 20 to 100 (left) or different total labels ($|V| + |E|$) ranging from 3 to 18 (right). The ratio of the $|V|$ to $|E|$ is fixed to 2:1 for any given total number of labels. For example, if there are total 15 labels, we have 10 vertices label and 5 edge label. Other parameters setting: D10kI7L200E4V4 (left) and D10kT20I7L200 (right). The support threshold is fixed to 1% at both cases

formance gain by efficiently handling the underlying subgraph isomorphism problem, which is a time-consuming step and by introducing two efficient subgraph enumeration operations, together with an algebraic graphical framework developed for reduce the number of redundant candidates proposed. Performance evaluation using various real datasets demonstrated a wide margin performance gain of FFSM over gSpan. The efficiency of FFSM is further confirmed using the synthetic datasets.

Acknowledgement We thank Mr. Michihiro Kuramochi and Dr. George Karypis in University of Minnesota for providing the synthetic data generator. We thank Mr. Xifeng Yan and Dr. Jiawei Han in University of Illinois at Urbana Champaign for providing the gSpan executable.

References

[1] C. Borgelt and M. R. Berhold. Mining molecular fragments: Finding relevant substructures of molecules. *Proc. ICDM02*, 2002.

[2] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical componuds. *Proc. of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–6, 1998.

[3] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. *in SIGMOD*, pages 431–442, 1999.

[4] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, 1997.

[5] L. B. Holder, D. J. Cook, and S. Djoko. Substructures discovery in the subdue system. *Proc. AAAI'94 Workshop Knowledge Discovery in Databases*, pages 169–180, 1994.

[6] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *In Proc. of the 4th European Conf. on Principles and Practices of Knowledge Discovery in Databases (PKDD)*, pages 13–23, 2000.

[7] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *In ICDM'01*, 2001.

[8] S. Raghavan and H. Garcia-Molina. Representing web graphs. *In Proceedings of the IEEE Intl. Conference on Data Engineering*, 2003.

[9] K. Shearer, H. Bunks, and S. Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–91, 2001.

[10] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. *In Proc. of the 15th International Joint Conferrence on Artificial Intelligence (IJCAI)*, pages 1–6, 1997.

[11] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *ICDM'02*, 2002.

[12] M. J. Zaki. Efficiently mining frequent trees in a forest. *SIGKDD'02*, 2002.

A APPENDIX

A.1 maximal proper Submatrix's Property

In this section, we give a detailed proof of the Corollary 3.2, which states that given a connected graph's CAM M and M's maximal proper submatrix N, N represents a connected graph.

As discussed in the proof outline in Corollary 3.2, it is sufficient to show that in N there is no such row i (with the exception of the first row) that i does not contain any edge entry. We prove this by two steps. First we show every row in M has at least one edge entry and second, we show that this property (every row has at least one edge entry) is an invariant as we driving N from M.

We use a proof by contradiction to prove the first step. Let us assume i ($i > 1$) is the least number such that $m_{i,1} = m_{i,2} = ... = m_{i,i-1}$=0. Since G is connected, there must exist at least a j ($j > i$) such that $\max\{m_{1,j}, m_{2,j}, \ldots, m_{i-1,j}\} > 0$. Otherwise, no node is connected to the first $i-1$ nodes from the remaining $n-(i-1)$ nodes and the graph must be unconnected. Given the existence of such (i, j) pairs, if we permute M such that i and j exchange their positions,

and keep every other node the same, we must have a code greater than the original code we have. That leads to the contradiction to the assumption that M is in canonical form. Thus we can not find a i in M satisfying $m_{i,1} = m_{i,2} = \ldots = m_{i,i-1} = 0$.

Given that every row in M (except the first row) has an edge entry, after removing the last edge, the obtained adjacency matrix must represent a connected graph.

A.2 Completeness of Suboptimal CAM Tree

In this section, we give a detailed proof of Theorem 3.4. We prove the theorem following the method of construction. For each type(A to D) of matrix M, we construct a pair of suboptimal CAMs and prove that pair might be joined to produce M. Intuitively, one suboptimal CAM in such pair is the maximal proper submatrix of M. This is true and another one is so-called secondary submatrix, defined below. For a typeE suboptimal matrix M, it is straightforward to prove that M can and only can be proposed by an extension operation and therefore is not discussed below.

Given a graph G and its CAM A $(m \times m)$, we define a $n \times n$ matrix B as the *secondary submatrix* of A by the following three cases:
Case 1: assume M satisfy one of the following conditions:

(i) A has three or more edge entries in the last row (TypeA);

(ii) A has exactly two edge entries in the last row (TypeB);

(iii) A has exactly one edge entry in the last row and more than one edge entries in the second-to-last row (TypeC);

Let the second-to-last edge of A be $a_{l,k}$. Now B is the secondary submatrix of A iff:
n = m, and

$$b_{i,j} = \begin{cases} a_{i,j} & 0 < i,j \leq n, i \neq l \wedge j \neq k \\ 0 & \text{otherwise} \end{cases}$$

Case 2: assume A has exactly one edge entry in both the last row and the second-to-last row of A. Let the second-to-last edge entry of A is $a_{l,k}$, and the last edge entry of A is $a_{p,q}(q \neq m-1)$ (TypeD). B is the *secondary submatrix* of M iff:
n = m-1, and

$$b_{i,j} = \begin{cases} a_{i,j} & 0 < i,j \leq n-1 \\ a_{m,j} & i = n, 0 < j \leq n-1 \\ a_{m,m} & i = n, j = n \end{cases}$$

The rest of the proof is in two steps. First we need to prove that both the maximal proper submatrix and the secondary submatrix of a suboptimal CAM are suboptimal CAMs. This is the result of the following theorem A.1. Second, we need to prove relation between the types of CAMs and the join/extension operation they might be produce.

Theorem A.1 *for any CAM M we have the following properties:*

(i) M's maximal proper submatrix is a CAM (and suboptimal CAM) of the connected graph it represents (Corollary 3.2)

(ii) M's secondary submatrix is a suboptimal CAM of the connected graph it represents

Similarly, for any proper suboptimal CAM N we have:

(i) N's maximal proper submatrix is a CAM(and suboptimal CAM) of the connected graph it represents (by definition)

(ii) N's secondary submatrix is a suboptimal CAM of the connected graph it represents

Proof outline: to prove that a secondary submatrix represents a connected graph, we take advantage of the result of Appendix A.1 such that every row of a suboptimal CAM has an edge entry. This is an invariant when we derive the secondary submatrix from a suboptimal CAM in all cases. Given a CAM M and its secondary submatrix N, to prove N is a suboptimal CAM, we notice N's maximal proper submatrix is the maximal proper submatrix of L, where L is the maximal proper submatrix of M. Then the theorem could be proved by applying Corollary 3.2 twice.

Given both the secondary submatrix and the maximal proper submatrix of a suboptimal CAM are suboptimal CAMs, it is trivial to show the connection between the types of the join and the type of the suboptimal CAM the join produces.

Figure 4 shows all the join operations and interested reader might verify the results there.