CHAPTER 4

INCREASING THE EFFICIENCY OF THE SOFTWARE ARCHITECTURE RECOVERY THROUGH SPANNING TREE BASED MAXIMAL GRAPH MINING TECHNIQUE

4.1. Introduction

In the present Chapter, to increase the efficiency of the software architecture a sophisticated graph mining technique called spanning tree based maximal graph is used. Query graph is first produced by making use of Architectural Query language [17, 48, 53, 60, 61]. A pattern graph is produced based on that query graph by using its Syntaxes and Semantics.

As the software grows the complexity also grows. Complexity of the big software system regarded as a major issue for the maintenance. Continuously using the software can make changes in its architecture. Reverse engineering process helps the project managers and code generators in understanding the software and components. Architecture recovery is one of the finest areas of software engineering, generally it adopts the reverse engineering process. Software recovery starts with the legacy software and it interpreted through the graphs. Bulk of papers’ are presented in this context many of them used graph as the main source of interpretation and some of them used the cluster based technique. Finding the frequent sub-graphs in the given main graph is one[1] important point in the graph based architecture recovery. In this approach each component is represented as the vertex and the relation between them as the edge. Vast literature as shown that to find the frequent data sets from the main graph used the Apriori based algorithm[1]. But there is significant disadvantage with A priori algorithm, first it needs the candidate generation step and two it creates the considerable overhead while joining two size-k and size-k+1 sub-graphs. In order over come from above two overhead we used an algorithm SPIN (Spanning tree based maximal graph mining technique) based on the
pattern growth approach to find maximum graphs with maximum association
[35].

4.2. Pattern Growth Approach and Overview of Spanning tree based
maximal graph mining
In this section once the graph is identified $g^1$, the recursion is continued until all
the frequent graphs are identified. Once there is no more graphs are generated
the recursion stops.

4.2.1 Labeled Graph
A labeled graph $G^l$ is a graph where each node and edge has an associated label.
We use $V^l$ and $E^l$ to denote the set of node labels and edge labels respectively.
Without loss of generality, we assume a total order $\geq$ on $V^l$ and $E^l$. The labeling
function $\theta$ defines the mappings $V \rightarrow V^l$ and $E \rightarrow E^l$

1. When we study recurring sub-graphs in graph databases, it is critical to know
whether a graph occurs in another graph, as defined below:

**Definition**: A labeled graph $G^l$ is **sub-graph isomorphic** to another graph $G^{11}$,
if there exists an injection $f : V[G^l] \rightarrow V[G^{11}]$ such that

* $\forall u \in V[G^l], (\theta(u) = \theta^1(f(u)))$,
* $\forall u, v \in V, ((u, v) \in E[G^l] \Rightarrow (f(u), f(v)) \in E[G^{11}])$, and
* $\forall(u, v) \in E[G^l], (\theta(u, v) = \theta^1(f(u), f(v)))$.

Where $V[G^l]$ and $E[G^l]$ denote the node set and edge set of a graph $G^l$. The
injection $s$ is a **sub-graph isomorphism** from $G^l$ to $G^{11}$. By a slightly abused
notation, we refer $G^l$ as a “sub-graph” of $G^{11}$, denoted by $G^l \subseteq G^{11}$ by omitting
the word “isomorphic”; similarly $G^{11}$ is referred to as a **super-graph** of $G^l$. A
labeled graph $G^l$ is defined to be **isomorphic** to another graph $G^{11}$ if $G^l$ and $G^{11}$
are mutually sub-graphs. Non- isomorphic sub-graph is referred to as a **proper
sub-graph**, denoted by $G^l \subset G^{11}$ and similarly $G^{11}$ is referred to as a **proper
super-graph** of $G^l$. Given a set $G^l$ of labeled graphs, the **support** of a graph $G^l$ is
the fraction of graphs in $G^l$ in which $G^l$ occurs.

4.3 Maximal Sub-graph Mining
In the following discussion, we present a novel framework for mining maximal
frequent sub-graphs. We show that we can unify tree mining and sub-graph
mining into one process where we first find all frequent trees from a graph
database and then construct frequent cyclic graphs from the mined trees. We developed two procedures to support this new framework the first one is a graph partitioning method where we group all frequent sub-graphs into equivalence classes based on the spanning trees they contain. First, tree related operations, such as isomorphism, normalization, and testing whether a tree is a sub-tree of another tree, are asymptotically simpler than the related operations for graphs, which are NP complete. Second, in certain applications such as chemical structure mining, most of the frequent sub-graphs are really trees. Last but not least, this framework adapts well to maximal frequent sub-graph mining, which is the focus of this paper. Using a chemical structure benchmark, we show 99% of cyclic graph patterns and 60% of tree patterns can be eliminated by our optimization technique in searching for maximal sub-graphs[35].

![Graph and related Sub Graphs](image)

**Fig:4.1** Source Graph and related Sub Graphs

### 4.4 Canonical Spanning Tree of a Graph

We define a *sub-tree* of an undirected graph $G^i$ as an acyclic connected sub-graph of $G^i$. A sub-tree $T^i$ is a *spanning tree* of $G^i$ if $T^i$ contains all nodes in $G^i$. There are many spanning trees for a given graph $G^i$. We define the maximal one according to a total order defined on trees as the *canonical spanning tree* of $G^i$, denoted by $T^i \ (G^i)$. First picking up maximal labeled nodes in $G^i$ as a group of single node trees. It iteratively grows those trees by attaching an additional node to each of them in all possible ways. The outcomes of these sub-trees are
properly checked and sub-tree with maximum peer value taken for next iteration. This procedure is guaranteed to converge to the canonical spanning tree of a graph. Since every tree is a graph, the procedure can be applied to obtain canonical representations of trees.

![Graph and canonical tree](image)

**Fig 4.2** All Spanning tree of graph G with Canonical tree

### 4.5 Tree-based Equivalence Class

In this section based on the canonical spanning tree we introduced, we introduce a graph partitioning. We outline a new frequent sub-graph search algorithm based on the graph partitioning method discussed hereinbefore with two steps: (1) mine all the frequent trees from a graph database and (2) for each such frequent tree $T$, find all frequent sub-graphs whose canonical spanning trees are isomorphic to $T$. Maximal frequent sub-graphs can be found subsequently among frequent ones.

![Classes of trees](image)

**Fig 4.3** Divide the above trees into equivalent Classes
Now generate the candidate edge from above one class

Class 1 \[ C : \{ e_1, e_2, e_3, \ldots, e_n \} \]

Fig 4.5  Convert all trees of Class 1 into Graphs

4.6 Source Graph Decomposition

4.6.1 Decomposed Source Graph

From the above procedure we decompose a source graph into sub graphs of most associate relationship generally called offline phase. For example \( G_s = \{ V_s, E_s \} \) be the source graph which has been divided into the number of frequent graphs with maximum association. Let the sub-graphs be \( \sum G_s^i \) denotes the graphs of most frequent datasets. Based on the above graph mining, find the sub-graphs with same canonical from can be under one class. Let \( \sum C_i \) be the set of classes of graphs which are obtained from the domain knowledge, system documents and other analysis tools this is done under the online phase. Here we call \( G_{c_i}^s \) be the sub-graph belong to the class.

4.7 Architecture Query Language (AQL)

A query containing a number of abstract components and connectors where each component imports a place holder and exports a matched place holder if any is known as an AQL[36,37,38,39].

Abstract Component: A collection of place holder here a place holder is one which retrieves the searching mechanism is called Abstract component.
**Abstract Connector:** A connection among the abstract components is called an abstract connector.

**Fig 4.6 The notations of abstract component and abstract connector in AQL,**

### 4.7.1. Query Generation

By applying any one of the following methods, we can generate an initial Pattern.

(i) Comparing the source code of the system with its architecture.

(ii) Applying a clustering technique

(iii) Using the available system architecture document

The objective of these methods is to extract small groups of system entities that represent the core functionality of the system modules. These groups are used to generate an initial Query graph of an AQL query [36,37,38,39]

Now, we got an idea of the AQL specification. Now, the textual specification of an AQL query should be transformed into a graph representation. The groups of graphs that are generated from the query-graph during the iterative matching process are defined. The generated graphs are related by recursive graph algebraic equations.
Now the query is generated. Generate a pattern graph based on that Query. Therefore the textual specification of an AQL query should be transformed into a graph representation. To generate a graph pattern we have to get a basic idea of source graph and a source region. Let the graph obtained from the AQL be $G_p^s$. Here the query graph is iteratively processed to get more and more graph generations.

![Diagram](image)

**Fig 4.7 Transforming textual specification of AQL to a query graph**

### 4.8 Graphing matching process

In this section we compare the sub-graph obtained from the section 3(source graph) with graph obtained from the AQL from the section 4. Here we used our previously proposed algorithm bipartite graph pattern matching. The experiments done on the c source code it is observed that graph mining with pattern growth approach is more efficient than other.
Fig 4.8 Comparison Graph