RECOVERY OF SOFTWARE ARCHITECTURE USING PARTITIONING APPROACH BY FIEDLER VECTOR AND CLUSTERING

7.1 Introduction

Software Architecture recovery includes the extraction of design patterns. Patterns may be found using many techniques such as fiedler vectors, using clustering methods, query languages etc. In this chapter, for evaluating design patterns clustering methods and the concept of fiedler vector are used.

A Software system is comprised of modules [26, 44] (which includes procedures, files, functions etc.). First these modules should be classified into subsystems. To classify this, construct a graph $G = (V, E)$ such that each vertex is consisting of the modules and each edge represents the relationship between these modules. Then we decide to classify the nodes into subsets such that cohesion between the nodes of class is maximized and the coupling between the nodes of different classes be minimized.

7.2 Modules Identification

Without identifying modules and relations, we cannot construct a graph, so the first step is to find the modules and relations among them.

There are many ways to find modules

1. The easiest way is to treat each file as a module as the functions in the file are semantically related.
2. We can consider group of files as a module. But here the question here is which files should be grouped?

3. We can also consider procedure as a module so, we are following the easier approach here i.e. considering each file as a module.

7.3 Relations

There will be three types of relations among files.

1. \textbf{f1 Useproc f2}: says there is one function in f1 which calls the other function in f2.

2. \textbf{f1 Usevar f2}: says there is one function in f1 which uses a variable defined in f2.

3. \textbf{f1 Implementby f2}: says there is one header function in f1 which is implemented in f2.

So, we can construct a graph, using these modules and relations among modules. The next step is to classify these modules into subsystems. So to divide the graph into sub graphs, use the concept of Fiedler vector. First of all we have to know what a Fiedler Vector is.

For the input graph G= (V, E)

1)calculate the Adjacency matrix

2)Calculate the Degree matrix

3)Calculate the Laplacian matrix

The adjacency matrix is

\[
A (i, j) = \begin{cases} 
1 & \text{if } (i, j) \in E \\
0 & \text{otherwise}
\end{cases}
\]
The degree matrix of the graph is the diagonal matrix of the row sum of the adjacency matrix

\[ D = \text{diag} \left( \text{deg} \left( i \right); \ i \in V \right) \]

\[ \text{Deg} \left( i \right) = \sum_{j \in V} A \left( i, j \right) \]

The Laplacian matrix \( L \) is the difference between diagonal matrix and Adjacency matrix [9]

\[ L = D - A \]

\[ L \left( i, j \right) = \sum_{(i,k) \in E} A \left( i,k \right) \quad \text{if} \quad i=j \]

\[ = - A \left( i,j \right) \quad \text{if} \quad i \neq j \text{ and } (i,j) \in E \]

\[ = 0 \quad \text{otherwise} \]

Now the matrix available is symmetric. The eigen vector \((1, 1 \ldots 1)^T\) corresponds to trivial zero. Eigen values [5]. Now with these Eigen values get the Fiedler vector. First arrange these Eigen values in the ascending order. The largest Eigen value and the second smallest Eigen value, whose corresponding Eigen vector is referred to as the Fiedler vector [81].

Now, Fiedler vector is known to us. Now decompose the graph into subgraphs. So, path sequence for the nodes should be calculated using a permutation \( \pi \). The sequence is that the elements of the edge weight matrix \( W \) decrease as the path is traversed if \( \pi \left( i \right) < \pi \left( j \right) < \pi \left( k \right) \) then \( W \left( i,j \right) > W \left( i,k \right) \) and \( W \left( j,k \right) > W \left( i,k \right) \)

Consider a vector \( x^* = (x_1, x_2, \ldots, x_{|V|}) \) of continuous variables \( x_i \). Calculate the penalty function

\[ g \left( x \right)^* = \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} W \left( i,j \right) \left( x_i - x_j \right)^2. \]

The constraints of this function are \( \sum_{i=1}^{|V|} \left( x_i \right)^2 = 1 \) and \( \sum_{i=1}^{|V|} x_i = 0. \)
7.4 Decomposing a graph into sub graphs

By using Fiedler vector concept, the graph is to be divided into sub graphs [43,62]. The neighborhood of the node \(i\) consist of its center node together with its immediate neighbors connected by edges in the graph.

\[ N_i^\wedge = \{i\} \cup \{u; (i,u) \in E\} \]

Assign to each node measure of significance as the center of the neighborhood. After assigning it traverse the path defined by the Fiedler vector. To traverse the path select center nodes on the basis of this measure. Assign weights to nodes based on the rank-order in the permutation. The weight assigned to the node is \(i \in V\) is \(w = \text{Rank}(\pi(i))\). After assigning weight to each node calculate the score function. By giving score to each node, the significance of the node will be known. Score can be calculated by using a function [8,57]

\[ S_i = C_1 (\text{deg}(i) + |N_i \cup P|) + C_2/ W_i \]

Where \(C_1\) and \(C_2\) are threshold values that were detected heuristically. \(P\) is the set of nodes on the perimeter of the graph. The first term depends on the degree of the node and its proximity to the perimeter. So the nodes will be sorted according to their distance from the perimeter. This was proposed because it is better to decompose from the outermost layer first. The second term says that the first ranked nodes in the Fiedler vector are visited first. We use the scoring function to locate the non overlapping neighborhoods of the graph \(G\). We traverse this list until we find a node \(K\) which is neither in the perimeter and also the calculated score should not exceed of its neighbors. If this condition gets satisfied then the node \(K\) together with its neighbors represents the first sub graph. This process will be repeated for all the nodes that satisfies the condition.
Find the sub graphs that are overlapping with its neighborhood. By performing this step the sub graphs are found. Input the overlapping sub graphs to a clustering algorithm. There are two major approaches for subsystem classification.

1) **Top down approach**

2) **Bottom up approach.**

In the top down approach, creation of subsystem includes all modules and then iteratively decomposes the current subsystems to create subsystems at lower levels. In the bottom up approach, consider each module as a subsystem and then iteratively merge subsystems to create those at higher levels.

Top down approaches suffer from exponential complexity as in A* algorithm. So, follow the bottom up approach.

1) For clustering, calculate the similarity between two nodes.

2) Identify a set of nodes that are pair wise most similar. After identifying, create a cluster by taking the union of the most similar cluster or creation of more than one cluster is also possible by taking the union of some of pairs of this set.

7.5. **Similarity/ Dissimilarity measures**

Two nodes are said to be similar if they have either the highest similarity measure [44] or lowest dissimilarity.

If the component of the system is entirely connected to just one other component, that connection should be computed as a lower dissimilarity than any connection that is not complete. It is based on the percentage degree of vertices common neighbors of two vertices. That is, let \( p \) be the dissimilarity matrix and is defined by
\[ p(i,j) = \text{deg}(i) + \text{deg}(j) - 2 \times \frac{b(i,j)}{\text{deg}(i) + \text{deg}(j)} - b(i,j) \]

where \( \text{deg}(i) \) is the degree of the vertex \( i \) in the graph and \( b(i,j) \) is the number of common neighbor of vertices \( i \) and \( j \). Since \( \text{deg}(i) + \text{deg}(j) - b(i,j) \) is the number of all vertices connected to exactly one of the \( i \) and \( j \). Note that if \( \text{deg}(i) = \text{deg}(j) = b(i,j) \) then \( p(i,j) = 0 \) and so \( i \) and \( j \) are completely similar. Note also if \( i \) and \( j \) have no common neighbor then \( p(i,j) = 1 \) and so \( i \) and \( j \) are completely dissimilar.

After clustering now there is a need to optimize the solution.

So consider the measurement of intra-connectivity and inter-connectivity.

7.5.1 Intra-connectivity

It is a measure of connectivity between the component that are grouped together in the same cluster. For good subsystem partitioning, the degree of intra-connectivity should be high because many software level features are shared by the modules grouped with in a common subsystem.

\[ A_i = \frac{m_i}{N_i^2} \]

Where \( A_i \) is the intra-connectivity measurement

\( N_i \) is number of components

\( m_i \) intra-edge dependencies.

7.5.2 Inter-connectivity

It is a measure of connectivity between two distinct clusters. Inter-connectivity should be very less. Inter-connectivity is denoted by \( E_{ij} \)

\[ E_{ij} = \frac{m_{ij}}{2N_iN_j} \quad \text{if} \; i \neq j \]

\[ = 0 \quad \text{if} \; i = j \]
i and j are clusters consisting of \( N_i \) \( N_j \) components. \( m_{ij} \) is inter dependencies. The clusters will be derived. To apply the clustering techniques to software architecture recovery and reengineering, the object-attribute data matrix should be converted to object-object data matrix, so that the input reflects the interconnectivity of components. The clustering techniques are then used to minimize interconnections among components. Here we explain how the clustering technique could be used to support the identification of a pattern.

There are some client classes that are accessing some subsystem classes. With existing software architecture recovery assistants, especially file names based approaches, the result may look perfect for the subsystem. In other words, the architecture recovered through this type of technique is close to or the same as the modules that are partitioned by the designer.

Certainly, architecture capture is important and valuable. But we are also concerned with ways to improve the architecture rather than simply capture it. Besides, in reality, the directory structures already often reveal the high level components of a system. Simply capturing software architecture at a higher level abstraction often has limited benefits. By applying the clustering technique to this example, we get very different partitions. In fact, the subsystem does not exist anymore, since many subsystem classes are directly accessed by or related to client classes. In other words, the clustering technique reveals that some classes in the subsystem are more closely related to client classes, which contradicts the design concept. Ideally, the subsystem classes should be grouped together as one unit. Clustering techniques could be used in this type of situations to enforce the architect to reason ways to keep the subsystem classes in a more cohesive manner. Facade pattern provides common interfaces to
subsystem classes and facilitates separation of concerns. The subsystem classes in the new pattern-based design are grouped in the same unit according to the clustering method, which is consistent with the original design. In this example, the clustering technique helps the adoption of a design pattern to reduce the coupling between the subsystem and the clients.