Clustering algorithms in PBS

MohammadTaghi Hajiaghayi
mhajiaghayi@uwaterloo.ca

Department of Computer Science
University of Waterloo

Abstract

A software system is typically comprised of several interconnected components, such as procedures, functions, etc. The software re-engineering attempts to deal with the difficult problems of understanding, re-documenting, and modifying the existing software systems. One of its task is to classify the components into subsystems and describe the interactions between these subsystems.

The interactions between subsystems may be inferred from the interaction between their member components. This task can be done by applications like PBS which find these interactions using compiling and linking information. After that, we construct an abstract model for this problem: A graph $G = (V, E)$ where V (set of nodes) is consisting the modules and E (set of edges) includes the relations between these modules. Finally, we construct the hierarchial structure for this graph which can be used for making *contain.rsf* file in PBS.

In this paper, we present some algorithms for the above problem from the literature and the algorithm which I have chosen for my project. In fact, my algorithm is the combination of two algorithms from the literature.

keywords: Software re-engineering, Hierarchial structure, Clustering algorithms, PBS.

1 Introduction

A software system is typically comprised of several interconnected components, such as procedures, functions, global variables, types, files, documentation, etc. The software engineering attempts to deal with the difficult problems of understanding, re-documenting, and modifying the existing software systems. One of its task is to classify the components into subsystems and describe the interactions between these subsystems[5]. This task provides both high-level abstraction of the organization and a very good documentation for a software.

The interactions between subsystems may be inferred from the interaction between their member components. This task can be done by applications like PBS which find these interactions using compiling and linking information. After that, we have some modules which can be procedures, files, etc. The crucial step is to classify these modules into subsystems, and find a hierarchal structure for the software. An abstract model for this problem is as follows: we have a graph $G = (V, E)$ where V (set of nodes) is consisting the modules and E (set of edges) includes the relations between these modules. Then we decide to classify the

nodes into subsets such that the cohesion between the nodes of each class be maximized and the coupling between the nodes of different classes be minimized. The number of vertices and edges of this graph is very large and so we cannot consider every clustering and then find the best of them. So we need a better approach.

The clustering problem also arises in other areas such as biology, social sciences, etc. Hence, so much research has been done in this field and there are so much solutions for this problem. In addition, Software clustering needs some more special considerations and deciding the best approach in this area is very hard job and needs a deep study. In this paper, we present some of the solutions from the literature in a stepwise manner and in each step talk about our approaches.

The structure of this paper is as follows: In section 2, we talk about modules and relations Identification. Then we talk about the different clustering algorithms in section 3. In this section we consider the general ideas of these algorithms and in some subsections talk more about each step of these general algorithms. In section 4, we talk about pattern driven approach for software clustering. We explain our algorithm and consider some case-studies in section 5 and finally we have a conclusion in section 6.

2 Modules and Relations Identification

First step in clustering algorithms is to find the modules and relations among them. We use these information to construct the graph for the next steps.

The definition of modules is different in the literature. One defined a module as a single function (a term we use interchangeably with *procedure*), another defined each source-code file as a module and the other defined a module as a group of files.

The first way to define a module uses a one-to-one transformation. However, this results in modules that consist of a single functions, with fixed(high) cohesion. This may result in an unnecessarily complex and misleading system design structure.

In the second defintion, we assume that the functions in a file are semantically related and form a cohesive logical module. Because it is usually true that programmers group related functions into a file, although with varying degrees of functional cohesion. It is reasonable to consider each file to be a module. Also most compilers use the file as the compilation unit, and other tools such as editors, debuggers, and pre-processors and post-processors use the file as their I/O medium. Finally when you choose to consider every source file as a module, you can readily query for other information, including its owner, latest author, latest revision, access conditions, file-path directory, and network file server, all of which can provide useful configuration management information.

In the last definition, we consider a group of files as a module but the problem here is how we must group the files to get one module. You may find some solutions for this problem in the literatures[14].

If you consider the procedures as modules then interactions between modules can be calling of a procedure, using a global variables or etc. You may see [13] for precise definitions of relations between the modules in the programming language C. But, if you consider the files (groups of files) as modules then the relations are using a procedure or variables in one file (group) which is defined in another file(group) .

Our final goal in this project is to provide one clustering algorithm for PBS. In PBS using compiling and linking information we can get all of static relations between procedure within one file or among more than one files (this information is saved in the *factbase.rsf*). But, the problem is that most of existing softwares are too large to consider one procedure or even part of a file as a module. So, in PBS, the modules are the files. The relations among files are refined in two more high-level files: *fileLevelFacts.rsf* and *highLevelFacts.rsf*. The latter is the most high-level file and has only this three relations among files:

- **useproc f1 f2:** says there is one function in f1 which calls the other function in f2.
- **usevar f1 f2:** says there is one function in f1 which uses a variable defined in f2.
- **implementby f1 f2:** says there is one header function in f1 which is implemented in f2 and mostly consists of header(.h) files and source code(.c) files.

The PBS uses the relations which are defined in highLevelFacts.rsf to show the relations among modules (files) in the screen. So, in this project, our modules are the files and the relations are those defined above and finally we use this information to construct our abstract model graph.

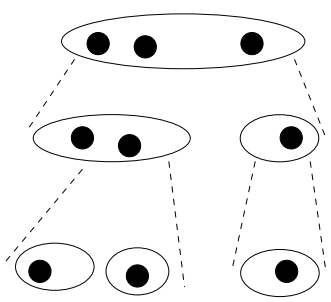
3 Clustering algorithms

3.1 Hierarchal Algorithms

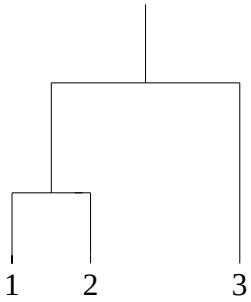
An organizations of a set of entities into subsets such that entities in the same subset are in *some sense* more similar to each other than to those in different subsets is often referred to as a *cluster*. *Subsystem classification* is another name for clustering in software engineering area. There are two major approaches for subsystem classification: *top-down* or *bottom-up*. In the top-down approach, first, the hierarchial algorithm create one subsystem which includes all modules and then, iteratively, decompose the current subsystems to create subsystems at lower levels. In the bottom-up approach first we consider each module as a subsystem and then iteratively merge subsystems to create those at higher levels.

Top-down approaches suffer from excessive computational complexity, as there is an exponential number of possible partitions at every step. This is the main reason that most of hierarchial methods use the bottom-up approach. The computation of bottom-up approaches may be abstracted by the algorithm template below:

```
INPUT: An interconnection graph G.
OUTPUT: A subsystem classification of the given graph G.
```



a): A hierrarchy of clustering for three entities



b): A dendrogram for the hierarchy of (a).

Figure 1: An example of dendrogram

```
S0: put each node of G in a subsystem by itself.
while G has more than one component do
  S1: identify a set (with at least two elements) of most similar subsystems.
  S2: create a new subsystem by merging the similar subsystems.
  S3: create a new graph G by replacing the similar subsystems by the new subsystems.
end-while
```

Starting with an interconnection graph these algorithms iteratively find a set of components that may be said to be similar (Step S1), place these components into the same subsystems (Step S2), and create a new interconnection graph by combining the two components (Step S3). Hierarchial algorithms differ in the strategies chosen at each of these steps.

One way of clustering called *hierarchial agglomerative clustering(HAC)*[3] is as follows: The input to HAC algorithm is a symmetric, directed graph G without self edges (i.e. there is no edge from a node to itself). The strategy at Step S1 depends upon whether the weights on the edges of G represent (a) similarity between two nodes or (b) dissimilarity between two nodes. Two nodes are *most similar* if they have either the highest similarity or the lowest dissimilarity. Step S2 of an HAC algorithm identifies a set of nodes that are pairwise most similar. After identifing of this set, then we can create only one cluster by taking the union of the most similar clusters, or create more than one cluster by taking the union of some of pairs of this set.

A common representation for a hierarchial structure is that of a *dendrogram.* Most of the HAC algorithms in step S2 create only one cluster by taking the union of the most similar clusters and so the resulted dendrogram will be a binary tree (a tree structure in which each node is a leaf or has two children). You can see one example in figure 1.

HAC algorithms traditionally choose from one of four strategies introduced in [3]: *single-link, complete-link, weighted-average-link,* and *unweighted-average-link* to create a new graph after merging a set of similar clusters to create a new clusters (Step 3). Let X be the set of similar clusters being combined in a given iteration, n be the new cluster replacing all the members of X, and e be another cluster (not in X). In the

single-link strategy, the smallest dissimilarity between any element of X and e is used as the dissimilarity between n and e. The complete-link strategy uses the largest such dissimilarity, and the other two strategies use weighted and unweighted averages of dissimilarities between e and members of X. Some others uses yet another strategy, termed *cumulative-link*, where the sum of the dissimilarities between e and all the elements of X is taken as the dissimilarity between e and n.

3.2 Measure of similarity and dissimilarity

As we have discussed, the first step of Hierarchial algorithms (Step 1) uses the similarity and dissimilarity of two nodes of graph G. In this section we talk about this subject and different approaches of computing this parameters.

Similarity and dissimilarity are two *distance functions* between two nodes. A distance function D over a set of E is a metric if and only if it satisfies the following conditions[4]:

1. $D(x, y) = 0$ if and only if $x = y$,
2. $D(x, y) \geq 0$ for all $x, y \in E$,
3. $D(x, y) = D(y, x)$ for all $x, y \in E$,
4. $D(x, y) \leq D(x, z) + D(z, y)$ for all $x, y, z \in E$.

Most of the similarity measures in the literature don't have the fourth metric property which is known as the *triangle inequality*, but, the best similarity and dissimilarity distance are the one that also has the fourth property. Also, note that similarity and dissimilarity measures can easily be computed as follows: $sim(i, j) = 1 - D(i, j)$ and $dis(i, j) = D(i, j)$. There are so many similarity and dissimilarity distance measures summarized in [9, 12] that we introduce some of them, here:

- **Association coefficient**

  Association coefficients or matching coefficients for two entities i and j are expressed in terms of the number of features which present for these entities. So association coefficients assume binary features. In the literature the table 2 is used for this purpose.

  In this table a represents the number of features for which both entities have the value 1, b represents the number of features present for entity i but absent for entity j, etc. The idea behind association metrics is very intuitive: the more relevant matches there are between the sets of present features of the two entities under comparison, the more similar the two entities are.

  Several choices are possible as to what features are relevant and more sophisticated coefficients use weighting. In the literature a lot of different association coefficient are presented (you may see [9] for more references). This diversity is caused by two factors:

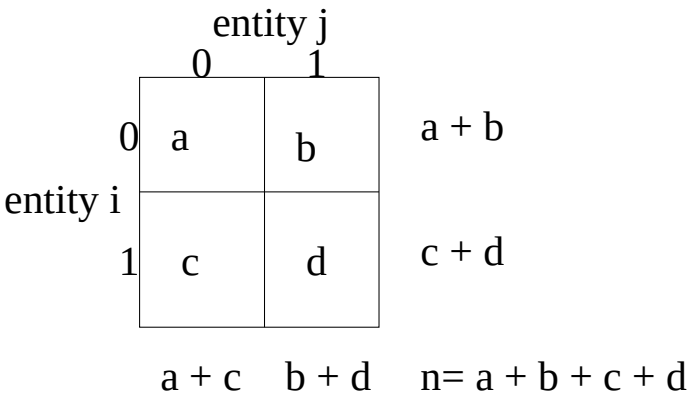


Figure 2: Association coefficient Table

- **The handling of 0-0 matches:** Should 0-0 matches positively influence the similarity measure and if so should they contribute with the same strength as 1-1 matches.
- **the weighting of matches and mismatches:** What is the best weighting for each matches?

[4] presents a list of possible coefficients by mechanically applying combinations of possible choices for these two factors. Some of the derived coefficients are widely used and are known under a specific name, others have no sensible meaning however. In this article we will only discuss the most frequently used coefficient is defined as:

$simple(i,j) = \frac{a+d}{n}$

This coefficient treats 1-1 matches and 0-0 matches equally, both contribute to the similarity. Matches and mismatches are weighted equally. The *Russel and Rao* coefficient excludes the 0-0 matches. The *Jaccard* coefficient doesn't take 0-0 matches into account at all:

$Jaccard(i,j) = \frac{a}{a+b+c}$

therefore this measure is very well suited for asymmetric binary features. The difference in weighting between matches and mismatches is usually done by a factor 2. The *Dice* coefficient applies double weighting of 1-1 matches to the Jaccard coefficient. A lot of other weighting are possible and are presented in the literature(see e.g. [4]).

- **Correlation coefficients** Correlation coefficients are originally used to correlated features. They are applied to the correlation of entities as well although it makes no statical sense to obtain the mean value across different feature types rather than across entities. The most popular coefficient of this sort is the Pearson product-moment correlation coefficient. The value of a correlation coefficient lies in the range from -1 to 1. A value of 0 means that two entities are not related at all.

  In [9] two formulae are presented to convert correlation coefficients to dissimilarity measures. Both formulae treat entities with a high positive correlation as very similar. The difference lies in the way negative correlations are handled.

- **Probabilistic measures**

Probabilistic measures are based on the idea that agreement on rare features contributes more to the similarity between two entities than agreement on features which are frequently present. So probabilistic coefficients take into account the distribution of the frequencies of the features present over the set of entities, When this distribution is known for each feature a measure of information can be computed.

The above similarity measures are based on different features that one object provides. But, in software clustering we also use other kind of similarity which is simpler than the above type. In this case, the problem can be represented as a graph, where the nodes are the subsystems and the edges are the relations. If we have more than one relation, then the graph will have multiple kinds of edges.

Common similarity measures that deal with cases like this are based on the number of edges connecting two objects, the length of shortest path between two objects, or the weight that different kinds of edges might have. Whether the graph is directed or undirected is also a factor.

Note that from the first measure of similarity we can easily find the second measure. Because we can construct a graph where its nodes are the subsystems and edges have weight which are the similarity between modules. This graph is a weighted undirected graph. We can also get an undirected graph without weight by taking only those edges which have at least certain amount of similarity between submodules and remove all other edges and all weights of remaining edges.

Also using the given graph, we can compute the similarity and dissimilarity of two objects [6]. One way that captures the intuitive notion is that if a 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 of the degree of vertices and the common neighbors of two vertices. That is, let p be de dissimilarity matrix defined by $p(i,j) = \frac{deg(i)+deg(j)-2*b(i,j)}{deg(i)+deg(j)-b(i,j)}$ where $deg(i)$ is the degree of vertex i in the graph and $b(i,j)$ is the number of common neighbor of vertices i and j. Since $deg(i) + deg(j) - b(i,j)$ is the number of all vertices connected to i or j and $deg(i) + deg(j) - 2 * b(i,j)$ is the number of all vertices connected to exactly one of i and j. Note that if $deg(i) = deg(j) = b(i,j)$ 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.

So, using above measures of similarity we can get a graph in each step of the above algorithm, and so we only need some algorithm for clustering. Also note that if we decide to merge only two vertices with the most similarity we get a binary tree, but it is not good for our purpose of finding the hierarchial structure of a system and constructing the contain.rsf file. In the next subsection we present some algorithm for finding a real hierarchial structure not just a binary tree.

3.3 Graph clustering algorithms

In this section we suppose that graph $G = (V, E)$ is given where the edges can be directed or undirected, weighted or without weight. We briefly talk about some algorithms and also mention their weak points.

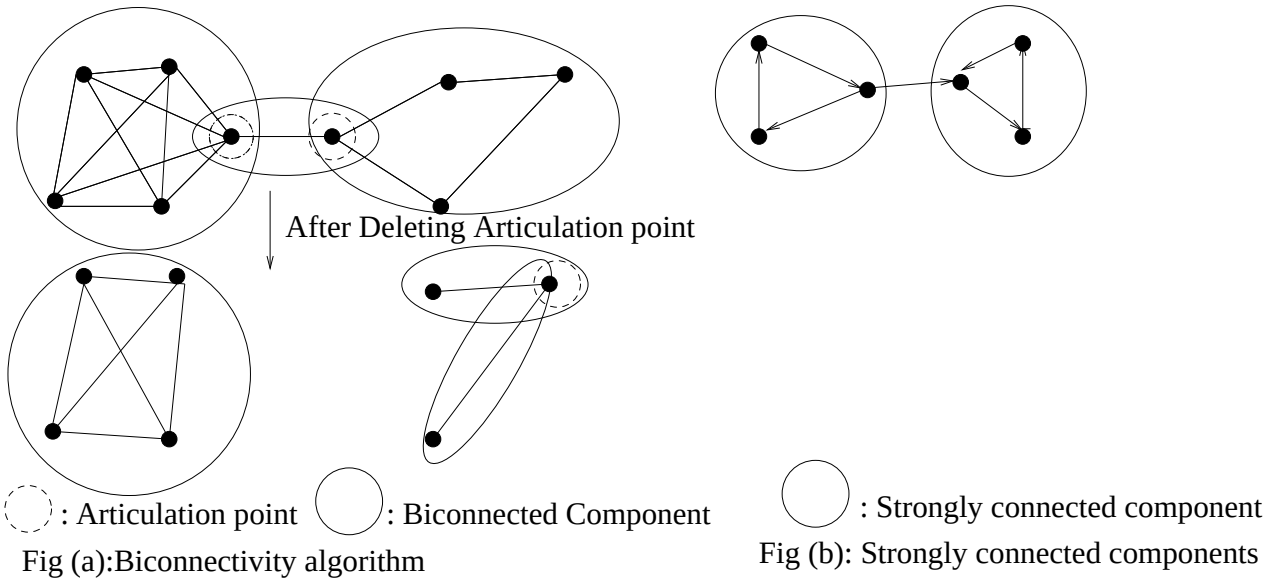


Figure 3: biconnectivity and strongly connectivity

3.3.1 Choi and Scacchi algorithm

To explain the algorithm presented in [8], we must first define two terms:

Definition 1 *The* control visibility *of subsystem S is the set of all descendent modules of that subsystem (which are controlled by it) in the hierarchial structure of a subsystem.*

Definition 2 *The* alteration distance *between modules is a measure of the distance between an altered module and the affected module. The alteration distance is zero if one subsystem controls both the altered and the affected module. Otherwise, the distance is the length of the path between the altered and affected module. The alternation distance of a system or subsystem is the sum of the alteration distances of all the modules of a system or subsystem.*

Now the algorithm presented in [8] accommodated both the minimum-coupling and minimum-alteration-distance objectives. First they begin with a graph in which each module is a node and each relation is an edge. The initial system is an undirected graph which we assume is connected. After that, the algorithm apply a biconnectivity algorithm, which divides the graph into subgraphs connected only by articulation points, the nodes which disconnect the graph if we delete them. Each of these subgraphs are corresponded to subsystems of a bigger subsystem, then for each of this subgraphs we delete the articulation points and recursively do the same thing to construct the overall hierarchial structure of the whole of system (see figure 3.a). The worstcase running time of this algorithm is $O(n^2)$ where n is the number of nodes in the graph.

I have implemented this algorithm, but it dose not seem a good algorithm for large systems. Since, in large systems there are few articulation points and it causes that after one iterations of the algorithm the system

cannot break down into small subsystems and so resulted subsystems are very big and we cannot understand the inside of each subsystem well.

3.3.2 A slight different algorithm

In the Choi algorithm we have assumed that the graph is undirected. However, if the graph is directed there is a similar algorithm. We can find the *strongly connected components* of the graph. The strongly connected component of a graph G, is a set C of nodes such that for each $u, v \in C$ there is a directed path form u to v in the graph G (see figure 3.b). Then overall algorithm is the same as Choi algorithm with this difference that we find the strongly connected components in each step. Again the worstcase running time of this algorithm is $O(n^2)$ where n is the number of nodes in the graph (see [2] for the implementing of finding strongly connected components)

I have also implemented this algorithm, but it has some drawbacks:

- It also recognizes big components similar to the problem which we had in Choi algorithm.
- There is no good thing like articulation points in Choi algorithm that we are able to remove them from the graph and find the structure of each components

Instead, in this algorithm, we have directed graph and the resulted clustering has better structure in respect with undirected case.

3.3.3 Optimizing algorithm

Mancoridis and et al. in [7] have introduced another approach for automatic clustering of an undirected software system's graph. They regard *Intra-Connectivity(A)* to be a measure of the connectivity between the components that are grouped together in the same cluster. A high degree of intra-connectivity indicates good subsystem partitioning because the modules grouped within a common subsystem share many software-level components. By maximizing the intraconnectivity measurement we increase the likelihood that change made to a module are localized to the subsystem that contains the module. They define the intra-connectivity measurement A_i of cluster i consisting of N_i components and m_i intra-edge dependencies as:

$A_i = \frac{m_i}{N_i^2}$

They also regard *inter-connectivity* to be a measurement of connectivity between two distinct clusters. A high degree of inter connectivity is an indication of poor subsystem partitioning. They also define the inter-connectivity E_{ij} between clusters i and j consisting on N_i and N_j components, respectively, with m_{ij} inter-edge dependencies as:

$E_{ij} = \frac{m_{ij}}{2N_iN_j}$ if $i \neq j$ otherwise it is 0.

After that they gave the below algorithm for minimizing inter-connectivity and maximizing intra-connectivity.

1. Let $S = \{M_1, M_2, \cdots, M_n\}$, where each M_i is a module in the software system.
2. Let MDG be the graph representing the relationships between the modules in S.
3. Generate an initial random partition P of set S.
4. Repeat
   - Randomly select a better neighboring partition bNP which is a partition resulted from P by moving one module from one set to another set or decompose one set to two disjoint sets and also it has $MQ(bNP) > MQ(P)$ where $MQ(P)$ is the sum of the intra-connectivity minus the sum of inter-connectivity of the partition P.
   - If a bNP is found, then let $P = bNP$.

   Until no further "improved" neighboring partitions can be found.
5. Partition P is the sub-optimal solution.

The above algorithm find a local optima point. They overcome the local optima problem of hill-climbing algorithms by choosing to go "downhill" for a while in the hope of finding a taller "hill" in the next few steps.

I have also tried to implement this algorithm, but again because if the system is large then the number of ways of partitioning is very huge, the local solution is not a good approximation for the optimal solution.

3.3.4 Muller algorithm

Muller has introduced *(k, 2)-partite graph* in [11] and also construct a tool called Rigi which uses (k, 2)-partite graphs to manipulate software system's graph[10]. A (k, 2)-partite graph consists of a series of graph levels called *layers*. Layers are connected by means of *level edges*; however, level edges may only connect adjacent layers. Also the number of vertices per layer is bounded by k. In their algorithm, the actual composition of layers is to be done by the user, the toll may be of help however, in producing alternative clusterings. Measures of composition used are interconnection strength (in terms of the size of interfaces) and clients/suppliers. Composition operations provided by the tool are: removal of omnipresent nodes(nodes with a high number of direct clients), composition by interconnection strength, common clients /suppliers, centricity and by name. All these operations are performed if the respective measures lie between specified thresholds.

The main drawback of this algorithm is that it is interactive and for large system is too hard to find the (k, 2)-partite structure of a graph.

There are also some other algorithm for clustering that you may see in [9], but now, we finish this section by explaining the main algorithm named Markov Cluster algorithm(MCL) which is introduced in [15] and I have used it in final version of my project and it seems the best among others.

3.3.5 MCL graph clustering

In this subsection we assume that we are given the undirected, unweighted graph G and we want to find the clustering of this graph, but what are natural clustering? This is in general a difficult problem, but within the framework of graphs there is a single notion which governs many proposals. This notion can be worded in different ways. Let G be a graph possessing cluster structure, then alternative wording are the following:

1. The number of higher-length paths in G is large for pairs of vertices lying in the same dense cluster, and small for pairs of vertices belonging to different clusters.
2. A random walk in G that visits a dense cluster will likely not leave the cluster until many of its vertices have been visited.
3. Considering all shortest paths between all pairs of vertices of G, links between different dense clusters are likely to be in many shortest paths.

Note that these three notions are strongly related to each other. Now, the idea is to measure or sample any of these-higher-length paths, random walks, shortest-paths and deduce the cluster structure from the behavior of the sampled quantities. The cluster structure will manifest itself as a peaked distribution of the quantities, and conversely, a lack of cluster structure will result in a flat distribution. The distribution should be easy to compute, and a peaked distribution should have a straightforward interpretation as a clustering.

k-path clustering:

For $k = 1$, the k-path clustering method coincides with generic single link clustering introduce in section 3.1. For $k > 1$ the method is a straightforward generalization which refines the similarity coefficient associated with 1-path clustering. Let $G = (V, E)$ be a graph, where $V = \{v_1, \cdots, v_t\}$, let $M = M_G$ be the associated matrix of G. For each integer $k > 0$, a similarity coefficient $Z_{k,G}$ associated with G on the set V is defined by setting

$Z_{k,G}(v_i, v_j) = \inf$ if $i = j$ and $Z_{k,G}(v_i, v_j) = (M^k)_{ij}, i \neq j$.

Note that the values $(M^i)_{pp}$ are disregarded. The quantity $(M^k)_{pq}$ has a straightforward interpretation as the number of paths of length k between v_p and v_q; this is the exact situation if G is a simple graph.

If G has dense regions separated by sparse boundaries, it is reasonable to conjecture that there will be relatively many path connections of length k with both ends in single dense regions, compared with the number of path connections having both ends in different dense regions. For weighted graphs, the interpretation is

in terms of path capacities rather than paths, and the formulation is now that the path capacities between different dense regions are small compared with the path capacities within a single dense region.

Also if we add loops(self edges) to each node of the graph G then M^k(M is the adjacency of graph G) is the number of paths of length at most k (not exactly k) between vertices and it would be better for showing the dependency.

Random walks and graphs

The standard way to define a random walk on a simple graphs is to let a Young Walker take off on some arbitrary vertex. After that, he successively visits new vertices by selecting arbitrarily one of the outgoing edges. This will be the starting point for the MCL algorithm. An excellent survey on random graphs is [1].

Also, to define the random walks on weighted graphs in general, the weight function of a graph has to be changed such that the sum of the weight of all outgoing edges equals one. This is achieved by a generic rescaling step.

Definition 3 *Let G be a graph on n nodes, let $M = M_G$ be its associated matrix. The* markov matrix *associated with a graph G is denoted by T_G and is formally defined by letting its qth column be the qth column of M normalized. To this end, let d denote the diagonal matrix that has diagonal entries the column weight of M, thus $d_{kk} = \sum_i M_{ik}$, and $d_{ij} = 0, i \neq j$. Then T_G is defined as $T_G = M_G d^{-1}$. The Markov matrix T_G corresponds to a directed weighted graph G', which is called the* associated markov graph *of G.*

In fact, Given a graph G and its associated markov matrix $T = T_G$, the value T_{pq} now indicated 'how much is the vertex q attracted to the vertex p, and this is meaningful only in the context of the other values found in qth column.

Now consider this generalization of normalization operation:

Definition 4 *Given a matrix $M_{k\times l}$, $M \geq 0$, and a real nonnegative number r, the matrix resulting from rescaling each of the columns of M with power coefficient r is called $\Gamma_r M$, and Γ_r is called the* inflation *operator with power coefficient r. Formally, the action of $(\Gamma_r M)_{pq} = (M_{pq})^r / \sum_{i=1}^{k} (M_{iq})^r$. If the subscript is omitted, it is understood that the power coefficient equals 2.*

Iterating expansion and infaltion

Suppose that we are given the adjacency matrix M of a graph G (suppose that each vertex has one loop to itself). Now we iterate the process of alternately expanding information flow via normal matrix multiplication and contracting information flow via application of Γ_r. Thus, the matrix $\Gamma_r M^2$ is squared, and the inflation operator is applied to the result. This process is repeated ad libitum. The invariant of the process is that flow in dense region profits from both the expansion and the inflating step. The intersting thing is that in [15] is proved after a few number of iterations(e.g. less than 10 times for graph G with 1000 vertices), the

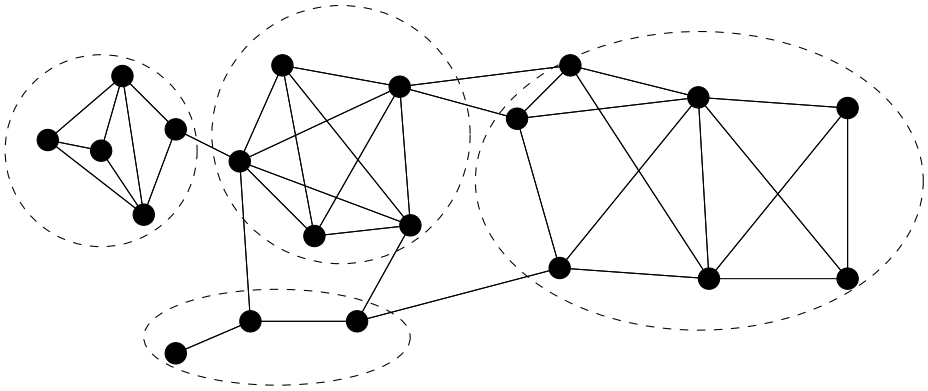


Figure 4: MCL clustering of a graph

matrix will not change anymore and it is a very sparse matrix, such that each vertex usually attracted by only one vertex, and vertices of dense regions will attract themselves, so if we find the connected componet of the graph associated to this matrix, we can easily find the the clusters. In the nest section the MCL process is formally described.

Formal description of the MCL algorithm

The basic design of the MCL algorithm is given below; it is extremely simple and provides basically an interface to the MCL process, introduced below. The main skeleton is formed by alternation of matrix multiplication and inflation in a for loop. In the kth iteration of this loop two matrices labelled T_{2k} and T_{2k+1} are computed. The matrix T_{2k} is computed as the previous matrix T_{2k-1} taken to the power of e. The matrix T_{2k+1} is computed as the image of T_{2k} under Γ_r.

MCL(G, e, r)

1. Add loops to Graph G;
2. $T_1 = T_G$;
3. **while** there are some changes in Matrix T_{2k} in respect with T_{2k-2} **do**
   (a) $T_{2k} = Exp_e(T_{2k-1})$;
   (b) $T_{2k+1} = \Gamma_r(T_{2k})$;
   (c) $k := k + 1$;
4. Interpret connected components of graph T_{2k+1} as clustering.

For example by calling the $MCL(G, 2, 2)$ for graph figure 4 we get the resulted clusters.

For our clustering algorithm with experience of some values , I finally get the $r = 4$ and $e = 2$ as the best values.

The most time-consuming part in the above algorithm is the matrix multiplication which is $O(n^3)$. Also, because the number of iteration of the mail loop is constant(less than 10 for 1000 vertices) the overall running time of this algorithm is also $O(n^3)$.

4 Pattern-driven approach

Most of algorithms presented in the software clustering literature identify clusters by utilizing certain intuitive criteria, such as the maximization of cohesion, the minimization of coupling, or some combination of the two or other parameters. Such criteria can definitely produce meaningful clusterings. However, it is a well-known fact that any clustering problem does not have a single answer and it is more important to produce a clustering that will be helpful to the developers trying to understand the software system, rather than to try to find the clustering that maximize the value of some metric.

Experiences indicates that certain patterns emerge time and again when humans create manual decompositions of software systems they are knowledgeable of. This suggests that if a decomposition contains these patterns, it would be easier to understand.

Tzerpos in his thesis[14] introduced some patterns refer to familiar subsystem structure that frequently appear in manual decompositions of large industrial software systems. They are as follows:

- **Source file pattern:** It says that usually we would group all the procedures and variables contained in the same source file into one cluster.
- **Directory structure pattern:** It says that usually we group the files contained in the same directory into one cluster.
- **Body-header pattern:** It says that usually we group the declaration file(header) and definition file(body) together.
- **Leaf collection pattern:** This pattern groups together resources that are leaves in the dependency graph of the software system
- **Library and dispatcher patterns:** These patterns group resources of large in-degree (or out-degree) into one subsystem.
- **Subgraph dominator pattern:** It looks for a particular type of subgraph in the system's graph.

The ACDC (Algorithm for Comprehension-Driven Clustering) introduced in [14] seeks for this special patterns and after finding it, reduces the nodes of this pattern into one super-node in the system's graph and iterate this operations till find the overall hierarchy. We have also used this approach in our project.

5 Our algorithm

For the algorithm of finding the hierarchial structure of a software subsystem, after experiencing of many ways, I have chosen this way:

Firstly, we find some of the patterns introduced in section before such as source file, body-header, leaf collection, library and dispatcher patterns and reduce the nodes of these patterns into one super-node in the system's graph. After that, using the general approach for getting hierarchial structure described in section 3, in each step I find the clustering using the MCL algorithm and after that I got a new graph by choosing each cluster as a node and there is an edge between cluster i and j if and only if there is an edge between one vertices in cluster i to cluster j. In fact, my algorithm is the combination of pattern-driven algorithm and MCL algorithm and it uses good properties of both of these algorithms: First, it uses the comprehensivity and second, it uses a general approach that can be adopted for every graph even if it doesn't have a good architecture and I think it is complement of both of them.

To run my algorithm, first we need to have the *factbase.rsf* file. After that, using *pbs-build*, we can generate the *highLevelfacts.rsf* file. Then, we can run our program to obtain the *contain.rsf* file.

I have also run my program for some software systems as case studies and got the below results.

5.1 Case study 1: C488

C488 is a sample compiler which is one of the examples of PBS. It is a somewhat small project and consists of approximately 20 files. Using my clustering program, I got the following *contain.rsf* file from *factbase.rsf* and *highLevelfacts.rsf* files:

```
contain gLib.ss <stdio.h>
contain gLib.ss <string.h>
contain gLib.ss <malloc.h>
contain gLib.ss <assert.h>
contain gLib.ss <stdarg.h>

contain sub8.ss main.c

contain sub7.ss codegen.c
contain sub7.ss machine.h
contain sub7.ss codegen.h
contain sub7.ss semanticext.h
contain sub7.ss machine.c

contain gLib1.ss scanner.c

contain sub5.ss symbol.h
contain sub5.ss symbol.c
contain sub5.ss common.h
contain sub5.ss machineDef.h

contain gLib1.ss scanner.h

contain sub3.ss parser.h
```

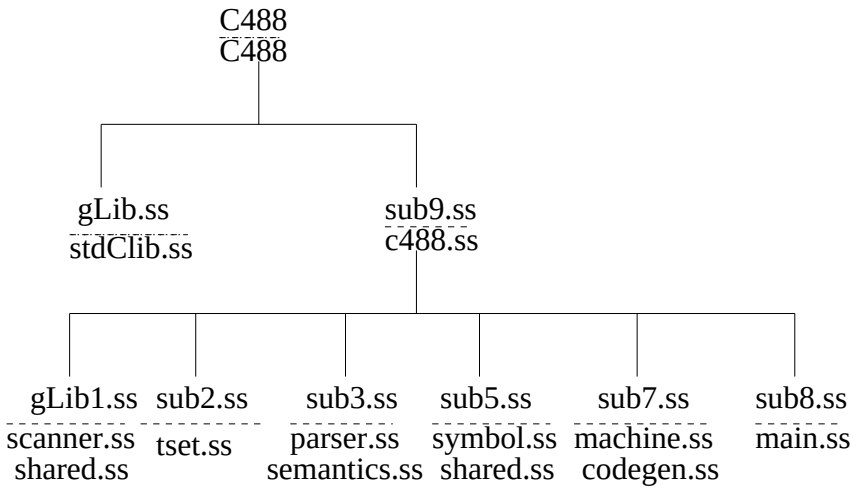


Figure 5: Resulted clustering for C488 project

```
contain sub3.ss semantics.c
contain sub3.ss parser.c
contain sub3.ss semantics.h
contain sub3.ss parsetables.h
contain sub3.ss parsecodegen.h
contain sub3.ss tokenDef.h
contain sub3.ss parsesemantics.h
contain sub3.ss scanparse.h

contain sub2.ss tset.h
contain sub2.ss tset.c

contain gLib1.ss globalvars.h

contain sub9.ss sub8.ss
contain sub9.ss sub7.ss
contain sub9.ss sub5.ss
contain sub9.ss sub3.ss
contain sub9.ss sub2.ss
contain sub9.ss gLib1.ss

contain C488 gLib.ss
contain C488 sub9.ss
```

The overall hierarchy of this system is shown in figure 5. In this figure, we have also compared the resulted clustering with the actual subsystem classification. In fact, this diagram is the combination of the two hierarchial structures. The subsytems above dotted lines are the generated subsystems and the ones bellow dotted lines are the actual subsystems. You can see that some of the resulted clusters contain more than one actual subsystems such as sub3.ss or sub7.ss and some of the actual subsystems are contained in more than one clusters i.e. shared.ss, however, the overall structure is very close to each other.

The PBS diagram is also shown in figure 6.

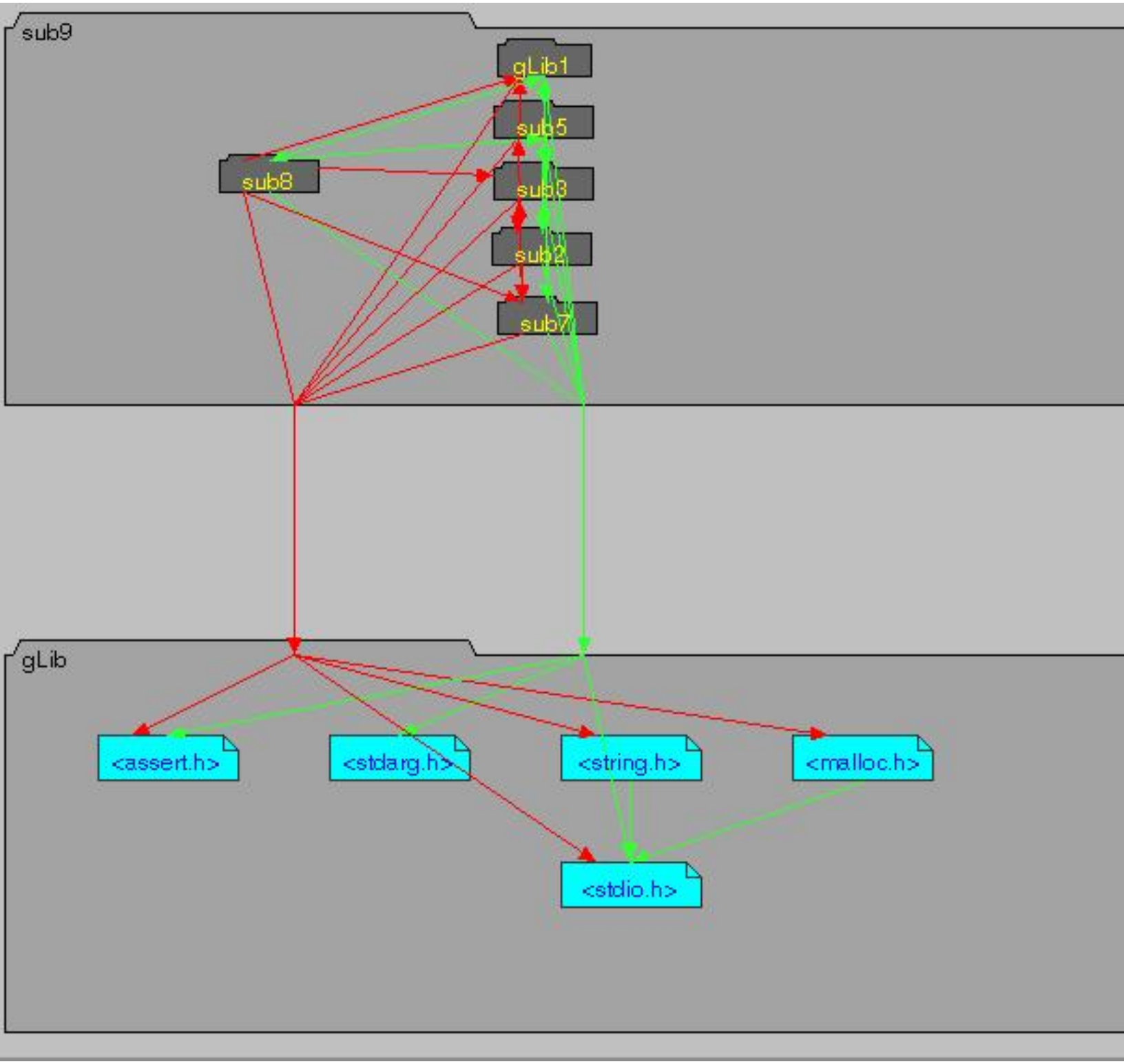


Figure 6: PBS diagram of C488 project

5.2 Case study 2: AolServer

In the next case study we consider AolServer which is a web server. This system consists of more than 320 files and it is a big project. I have received the *factbase.rsf* file of this project from Ahmed Hassan and using my clustering algorithm I have generated the *contain.rsf* file.

Then, I have compared the resulted hierarchy with the the hierarchy which Ahmed Hassan done by hands. The number of subsystems and the the overall structure is the same(both of them have around 25 subsystems), however, some of the resulted clusters in automatic hierarchy is grouped into one subsystem in the manual subsystem classification. Also the modules in one of the clusters(sub56.ss) are so many related and so the clustering algorithm recognize all the files as a one group, but in manual construction, this subsystem decomposed further.

Also, the running time of my algorithm for this project is so fast (approximately on-line). The number of subsystem are too large to mention here but you can find the PBS diagram at *http://swag.uwaterloo.ca/ nautilus/pbs/C488/R10 and R11.*

I have also tried another experience for this software system: in the current project, I have treated the *implementedby* links similar to other links , so it is possible that two modules in two different subsystems have a *implementedby* link between themselves, but, I have changed my program slightly such that if two modules have a *implementedby* link, they necessarily place in one subsystem. This approach seems logical, however, the result was not so interesting, because one of the subsystem (sub14.ss) contained so many modules and the other subsystems contained somewhat few modules . You may see PBS diagram at *http://swag.uwaterloo.ca/ nautilus/pbs/C488/R20.*

5.3 Case study 3: Apache

I have also run my algorithm for Apache Web server which consists of about 150 files. My algorithm was on-line for this system. Again, the overall structure is similar to the manual subsystem classification but still there are some inconsistencies. You can also find the PBS diagram at *http://swag.uwaterloo.ca/ nautilus/pbs/C488/R13.*

6 Conclusions

In this paper, we have considered clustering algorithms for software systems. We have reviewed the solutions for this problem from the literature and finally we have chosen the combination of two of them for our purpose of generating *contain.rsf* file. Before choosing the best one, I have also implemented some others and see their results.

Still, there are some drawbacks in our algorithm as follows:

- We have used undirected and unweighted graph G as our abstract model of the software system. However, using direction and weight (number of relations between two modules), we can certainly get better results. But, I couldn't find a good algorithm for clustering in these cases, however, I think it is possible to generalize the MCL algorithm to solve these cases. But, it needs more theory work before.
- In the MCL part of algorithm we have used the Matrix Multiplication and its running time is $O(n^3)$ where n is the number of vertices of our abstract graph. This operation is time-consuming, and takes so much time for software with more than 1000 modules. However, it is possible to parallelize this operation and get the time $O(\log(n)^2)$ and so the process would be fast even for very large softwares.
- I have not used some patterns such as directory structure and subgraph dominator patterns in my algorithm. I think, we can get a better result by using them.

Finally, I think, by solving the above drawbacks, our algorithm will be the best among all other algorithms in the literature.

References

[1] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[2] J. A. McHugh, *Algorithmic graph theory*, Prentice-Hall, Englewood Cliffs, 1990.

[3] N . Jardine, R. Sibson, *Mathematical Taxonomy*, John Wiley and Sons Inc., New York, 1971.

[4] M. R. Anderberg, *Cluster Analysis for Applications*, Academic Press, New York, 1973.

[5] D. Garlan, M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering, Volume 1*, World Scientific Publishing Co., 1993.

[6] D. H. Hutchens, V. R. Basili, "System structure analysis: Clustering with data bindings", *IEEE Transactions on Software Engineering, August 1985, 11(8)*, 749-757.

[7] S. Mancoridis, B. S. Mitchell, et al., "Using automatic clustering to produce high-level system organizations of source code ", *IEEE proceedings of the 1998 International Workshop on Program Comprehension*, IEEE Computer Society Press, 1998.

[8] S. C. Choi, W. Scacchi, "Extracting and restructuring the design of large systems", *IEEE Software, January 1990*, 66-71.

[9] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization", *Proceedings of the Fourth Working Conference on Reverse Engineering, October 1997*, IEEE Computer Society Press, 33-43.

[10] H. A. Muller, O. A.Mohmet, "Reverse Engineering Approach to Subsystem Identification", *Software Maintenance and Practice 5(1993)*, 181-204.

[11] H. A. Muller, J. S. Uhl, "Composing subsystem structures using (K,2)-partite graphs", *Proceedings of the Conference on Software Maintenance, Nov. 1990*, 12-19.

[12] A. Lakhotia, "A Unified Framework for Expressing Software Subsystem Classification Techniques", *J. of systems software 36(1997)*, 211-231.

[13] R. Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution", *Doctoral Thesis, Institut fur Informatik, Universitat Stuttgart, 2000.*

[14] V. Tzerpos, "Pattern-driven Software Clustering", *Doctoral Thesis, Department of Computer Science, University of Toronto, 2000.*

[15] S. M. van Donge, "Graph clustering by flow simulation", *Doctoral Thesis, Department of Computer Science, Proefschrift Universiteit Utrecht, Netherlands , 2000.*