Minimizing the complexity involved in Software Architecture recovery by using Bipartite graph

Shaheda Akthar

Assoc.Professor
Srimittapalli College of Engineering
JNTU,Kakinada

Dr.P.Thrimurthy

Professor
Dept of Computer Science&Engineering
Acharya Nagarjuna University

Abstract:
In pattern matching problem, Software Architecture Recovery is of immense importance, it uses A* Algorithm which runs in exponential time. In this article we proposed an algorithm which runs in linear time.

Key words:
Software Architecture recovery, reverse Engineering, Bipartite graph, Graph Matching Problem,AQL.

1. Introduction

At the present time, Software Architecture recovery [6][7] is exhibited as pattern matching problem, a constraint satisfaction problem, a clustering problem, a composition and visualization problem or a Lattice partitioning problem. The pattern matching problem is the best suitable method for Architecture recovery as observed, by the reverse Engineering Community [8] , because it uses Domain knowledge and system constraints and they can provide an user assisted environment.

1.1 Definition

First of all we have to get a basic idea of what is a Bipartite graph in order to get closer into this concept. We should also get closer to what is software Architecture recovery? To make this concept familiar to you, we have to follow these definitions.

1.2 Software Architecture recovery

A set of methods adopted for extraction of Architectural information from lower levels such as source code is called software Architecture Recovery.

1.3 Bipartite graph

In a non directed graph G whose set of vertices can be partitionised into two sets m and n in such a way that each edge joins a vertex in m to a vertex in n.

Manuscript received July 5, 2009
Manuscript revised July 20, 2009

In Graph1 vertices set |v|= {a, b, c, d} is divided into two disjoint sets v1 and v2 as

$$v1=\{a ,c\} \quad v2=\{b, d\}$$

In Graph2 every edge of G joins a vertex of v1 to a vertex of v2 . So G is a Bipartite graph.

Fig 1:(a)

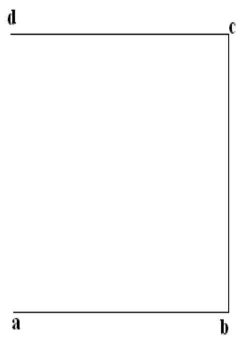
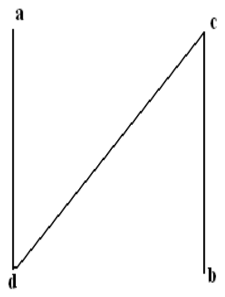


Fig 1:(b)



2. Similarities in Software Architecture Recovery Process and Graph Pattern Matching Process

A graph pattern matching problem is defined as Software architecture recovery, because both uses recursive graph equations that correspond to an iterative graph matching process.

2.1. Overview of matching process

Software Architecture recovery problem and pattern matching Problem (Graph Matching Problem)are similar to each other. But you have to know what a Graph Matching Problem is? In Graph Matching problem , we consider two graphs G1 and G2 by means of a function f that maps the nodes and edges of G1 onto the nodes and edges of G2 .The process of generating the pattern graph is exploratory in nature and the pattern graph is not a final graph in the proposed Software architecture recovery[6] [7]. In this way finding the exact matrix between the pattern graph and a sub graph of a Software system graph is not predictable. The aim is to identify a sub graph of the input graph that is similar to a given pattern graph. The search space for the matching process is provided by the source graph. This search space is separated into sub spaces using Data mining techniques and each sub space provides a sub set of initial search space. Each node in a source region is labeled with a similarity value to the main seed of the source region as a means for the matching process to operate on graphs of highly associated entities.

The Whole graph matching process [3] is done in $|N^q|$ iterations, where $|N^q|$ is the number of nodes in the query graph G^q. At each phase of development i where i takes values from 1 to $|N^q|$ the result of matching at previous phase G^m_{i-1} is used to build an input-graph G^I_i and a pattern graph G^P_i to be matched and produce a matched graph G^m_i, which in turn is used to build G^I_{i+1} and G^P_{i+1} for the next matching phase i+1 ,and so on. In this context,$G^m{}_0$ is defined as a Nil graph with zero number of nodes and edges , and when i = $|N^q|$ then $G^I_i = G^I$, $G^P_i = G^P$, $G^m_i = G^m$ and matching process terminates.

2.2 Problem involved in A* Algorithm

Previously Software Architecture is recovered using A* (Approximation Graph Matching Algorithm). But this algorithm suffers from Exponential time complexity because it uses Breadth first search method [4] for the temporary storage

This Search Algorithm generates a search tree that corresponds to the recovery of each module $_{Mi}$ in AQL (Architectural Query Language)

It Consists of a

i) root node for matching the main seed of the Source region with the first place holder $n_{i,1}$ in the pattern region $G_i^{pr.}$

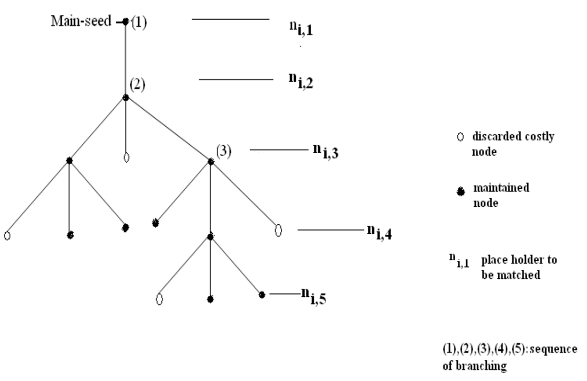
ii) A number of non-leaf tree-nodes at different levels of the search-tree that correspond to different alternative matching of the place holders in the Pattern region with nodes in the source region

iii) Leaf tree-nodes that correspond to solution paths where the placeholders have been matched and constraints have been met.

Each phase sets a place holder for the process of matching by the search tree which will be divided into number of phases. This act of composing property allows managing the complexity of the matching process of a large Source graph as shown in the figure below.

In this way, the whole matching process is divided into K incremental phases so that the recovery process performs a Multiphase matching. Each Partial Matching at phase i where i takes values from 1,2,3,……k generating a search tree which is a part of Multiphase Search space as shown in the above fig.

Fig:2(a)



Note: In this algorithm, the result of each phase is stored in a queue by discarding the previous result that was stored.

If the Current phase i of the matching process fails to identify a matched graph G^m_i then this algorithm should backtrack by

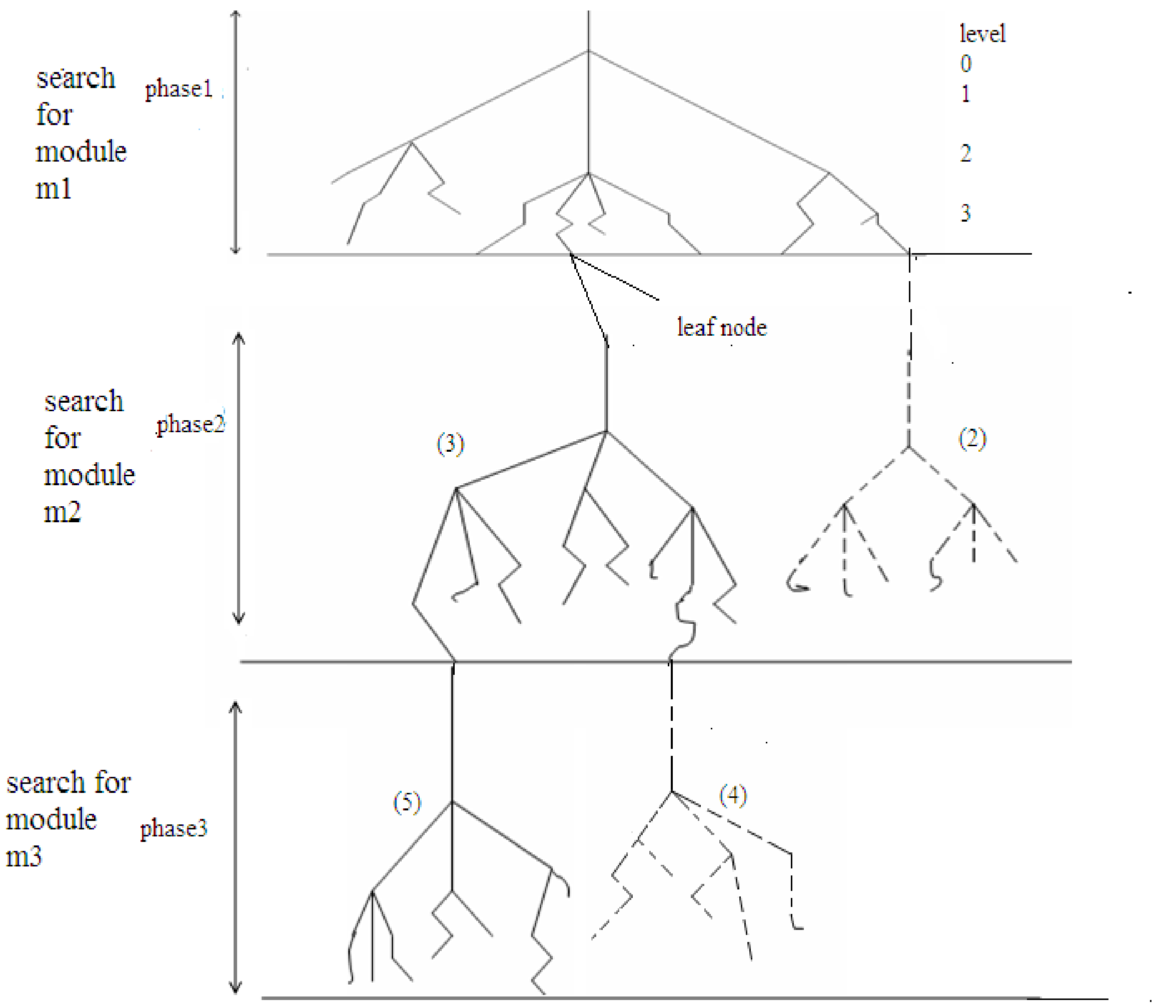
i) Discarding the result that was stored queue in its previous phase G^m_{i-1}
ii) Restoring the search tree for previous phase i-1
iii) Expanding the search tree to find another solution G^{m1}_{i-1}
iv) Advancing to the current phase i and generating a new search tree from G^{m1}_{i-1}

In our pattern matches in the nth phase, we are backtracking to the root n times in this algorithm.

So in this way we can conclude time complexity increases by an exponential order.

In the very first phase, if the pattern matches, then this algorithm is best suited for software architecture recovery.

We are proposing a new algorithm to reduce the complexity involved in this algorithm for recovering Software Architecture. We are dividing the source graph into two sub graphs in this algorithm by using graph Bipartition method [1]. As analysed above divide the graph into two subgraphs by selecting vertices [2]. The major problem is the selection of vertices. It is clear that there will be a problem while dividing.



3. Bi Partition Problem

The bipartition problem is described in this way:
Input:

(1) an undirected graph $G=(V,E)$ with $n=|V|$ vertices;

(2) two distinct vertices $s_1,s_2 € V, s_1 \neq s_2$;

(3) two natural numbers n_1 and n_2 such that $n_1+n_2 = n$.

Output:

A partition (V_1,V_2) of vertex set V such that

(1) $s_1 €$ V1 and $s_2 € V_2$;

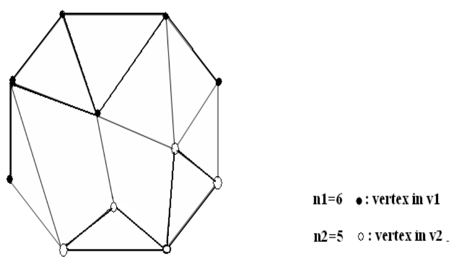
(2) $|V_1|=n_1$ and $|V_2|=n_2$;and

(3) Each of V1 and V2 induces a connected sub graph of G.

The following figure shows an instance of the problem and a solution. In this algorithm for the general result for the k-partition problem, in which one wishes to partition a given graph into k disjoint connected sub graphs each of which contains a specified vertex and has a specified number of vertices.
We present a simple linear algorithm in this paper which solves the bipartition problem for bi connected graphs. To recover Software architecture, this solution will be useful. This algorithm is based on characteristics of a depth first search tree in a bi connected graph.

FIGURE:

Fig:2(c)



An instance of bipartition problem and a solution(thick lines depict the subgraphs induced by v1 and v2)

Preliminaries

Let G=(V,E) be an undirected connected graph with vertex set V and edge set E. The vertex set of a graph G is often denoted by V(G), G-X is the graph obtained from G by removing all the vertices in X and all the edges incident with vertices in X .
Let T be a depth first search tree of G. For each vertex v € V the set of descendants of v including v itself is denoted by DES (v). In this paper, ancestors and descendants of v € V include v itself. It is clear from the following lemma.

Lemma

Let G =(V,E) be an undirected graph, and let T be a depth first search tree of G. Then G is biconnected if and only if the root r of T has exactly one child c and, for each vertex v € V –{r,c} there is an edge which joins an ancestor of v' s grandparent and a descendent of v.

Algorithm

We present a linear algorithm in this section for solving the bipartition problem for a biconnected graph G which is similar to the problem involved in Software Architecture recovery. G does not always contain edge (s_1,s_2) joining the two specified vertices s_1 and s_2.In what ever way a solution for the graph obtained by adding edge (s_1,s_2) to G is always a solution for G. In this way in the algorithm below, we may suppose that G has edge (s_1,s_2) . Let T be a depth first search tree with s_1 as the root and s_2 as the s1's child. The algorithm will be noted below.

```
Function PART2 (G, T, s1, s2, n1, n2);
Begin
(1) If n1=1 then return ({$s_1$},V(G)-{$s_1$})
        else  if $n_2$=1 then return (V(G)-  {$s_2$},{$s_2$});

(2) let a be an arbitrary child of $s_2$;
  If $s_2$ has two or more children then {see fig 2.note that lemma1 implies that, for every child v of $s_2$, s1 is adjacent to a vertex in DES (v)}
(2.1) if DES (a) U {$s_2$} $\leq$ then
 Begin {include DES (a) in$V_2$}
 $V_2$:= DES (a);
$G_{21}$:=G-$V_2$ ;{ G21 is biconnected, and is obtained from G by identifying all descendants of a with $s_2$}
$T_{21}$: T-$V_2$ ;{ T21 is a depth first search tree of $G_{21}$}
($V_1^1$, V2):=PART2 ($G_{21}$, T21, s1, s2, n1,|V($G_{21}$)-$n_1$);
Return ($V_1$, V2UV21)
End
(2.2) else {|DES (a) U {$s_2$}|>$n_2$, that is,
| (DES ($s_2$)-DES (a)-{$s_2$}) U {$s_1$}|<$n_1$}
Begin {include DES ($s_2$)-DES (a)-{$s_2$} in $V_1$}
$V_1$:=Des ($s_2$)-DES (a) {$s_2$};
$G_{22}$:=G-$V_1$ ;{ G22 is biconnected}
$T_{22}$:=T-$V_1$;
($V_1^1$, V2):=PART2 ($G_{22}$, T22, s1, s2, |V ($G_{22}$)|-$n_2$, n2);
Return ($V_1$U$V_1^{1,}$ $V_2$)
End

(3) else {a is the only child of $s_2$}
    Begin
    Let b an arbitrary child of a;
(3.1) If $s_1$ is adjacent to a vertex in DES (b) then {see fig.3}
(3.1.1) if |DES (b) U {$s_1$} $\leq n_1$ then
      Begin {include DES (b) in $V_1$}
V1:=DES (b);
Let $G_{311}$ be the graph obtained from G by identifying all vertices in $V_1$ with $s_1$;
$T_{311}$:=T-$V_1$;
($V_1^1$,$V_2$):=PART2($G_{311}$,$T_{311}$,$s_1$,$s_2$,|V($G_{311}$)|-$n_2$,$n_2$);
```

```
Return ($V_1UV_1^1$, $V_2$)
End
(3.1.2) else {|DES (b) U {$s_1$}|>$n_1$, that is, | (DES (a)-DES (b)) U {$s_2$}|<$n_2$}
begin
{include DES(a)-DES(b) in $V_2$}
$V_2$:=DES (a)-DES (b);
Let $G_{312}$ be the graph obtained from G by identifying all vertices in $V_2$ with $s_2$;
Let $T_{312}$ be the tree obtained from T by identifying all vertices in $V_2$ with $s_2$;
($V_1$, V21):=PART2 ($G_{312}$, T312, s1, s2, n1,|V($G_{312}$)|-$n_1$);
Return ($V_1$, V2UV21)
End
(3.2.) else {$s_1$ is not adjacent to any vertex in DES (b), and hence $s_2$ is adjacent to a vertex in DES (b). see in fig4.}
(3.2.1) if |Des (b) U {$s_2$}≤$n_2$ then
Begin   {include DES (b) in $V_2$}
$V_2$:= DES (b);
$G_{321}$: G-$V_2$;
$T_{321}$:=T-$V_2$;
($V_1$, V21):=PART2 ($G_{321}$,$T_{321}$,$s_1$, s2,n1,|V($G_{321}$)|-$n_1$);
Return ($V_1$, V2UV21)
End
(3.2.2)    else {|DES (b) U {$s_2$}|>$n_2$, that is, |(DES(a)-DES(b))U{$s_1$}|<$n_1$}
begin
{include DES(a)-DES(b) in $V_1$}
$V_1$:=DES (a)-DES (b);
Let $G_{322}$ be the graph obtained from G by identifying all vertices in $V_1$ with $s_1$;
Let $T_{322}$ be the tree obtained from
T-($V_1$-{a}) by identifying s1 with
 a ;{select $s_2$ as the root of $_{T322}$}
($V_2$,$V_1^1$):=PART2($G_{322}$,$T_{322}$,$s_2$,$s_1$,$n_2$,|V($G_{322}$)|-$n_2$);
Return ($V_1UV_{11}$,$V_2$)
    End
    End
End
```

Figures:

Fig:3(a)

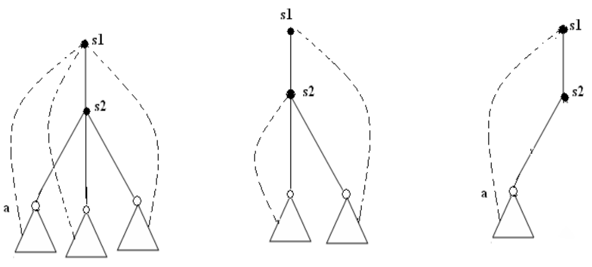


Fig (a)G (b)G_{21} (c)G_{22}

Lemma:

All graphs G_{21},G_{22},G_{311},G_{312},$G_{321}$and G_{322} in PART2 are biconnected , T_{21},T_{22},T_{311},T_{312},and T_{321} are depth first search trees with s1 as the root in G_{21},G_{22},G_{311},G_{312} and G_{321}, [5]respectively, and T_{322} is a depth first search tree [4] with s_2 as the root in G_{322}.
One can prove the correctness of the algorithm by using Lemma.
In order to make the algorithm run in O(m) time, we compute and maintain low(v) and id(v); for each vertex v € V, low(v) is defined to be the vertex which is adjacent to a vertex in DES(v) and whose depth first number is minimum; and
Id (v) = {0, if v € V1UV2U {s_1, s2},

1, if v€ V_1U {s_1},
2, if v€ V_2U {s_2}}

In this way, one can decide whether S1is adjacent to a vertex in DES (b) ((in 3.1)) by checking whether id(low(b))=1. We initially set id(s_1)=1, id(s_2)=2 and id(v)=0 for each v€V-{s_1,s_2}, and update id according to proceeding of the algorithm . But yet it is not necessary to update low (v).
After an execution of (3.2.2), id(low(v)) may become incorrect for some vertices v in DES(a)-DES(b), but such vertices have been included in V1 and hence will not be selected as b. Thus, it suffices to compute low (v) for all v€V only once at the beginning of the algorithm.

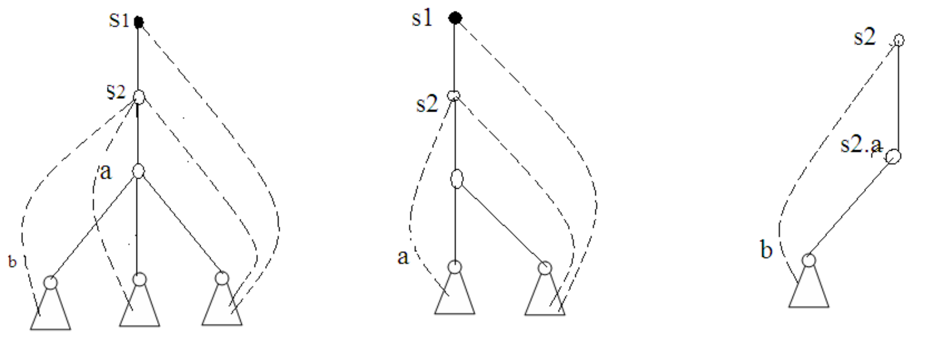


Fig 3(c) (a) G (b)G321 (c)G322

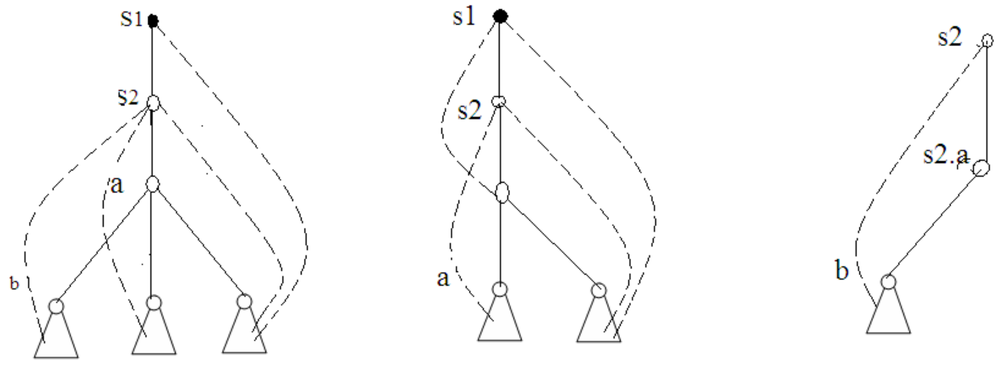


Fig 3(d) (a) G (b)G311 (c)G312

A depth first search tree [4] of a graph and low (v) for all vertices v€V can be found in O (m) time. All the other tasks can be done in O (n) time. Thus the bipartition problem for biconnected graphs can be solved in O (m) time.
In this algorithm the source graph should not be divided into multiple phases. So there is no need for backtracking. So, the performance increases

Conclusion

This algorithm runs in linear time which improves the performance in terms of iterations when compared to A* algorithm.

References

[1] M.E.Dyer and A.M.Frieze , on the complexity of partitioning graph into connected sub graphs, Discrete applmath.10(1985)139-153.
[2] E.Gyori, On division of connected subgraphs, in Combinatorics(1978)
[3] M.Imase and Y.Manabe , Fault tolerant routings in a connected network,Tech. Rept. Inst. Elect. Commun. Eng.Japan. COMP86-70,1987,95-105.
[4] Lovasz. A homology theory for spanning trees of a graph,Acta math.Acad.Sci. unger.30(1977)241
[5] H.Suzuki, N.Takahashi, T.Nishizeki,H.Miyano and S.Ueno, An algorithm for tripartitioning 3-connected graphs,Tech.Rept. Inf. Proc. Soc. Japan,AL5-11,1989.
[6] Garlan D., Perry D.E., Introduction to the Special Issue on Software Architecture *IEEE Transactions on Software Engineering*, Vol. 21,No. 4, April 1995
[7] Gall H., Jazayeri M., Klösch R., LugmayrW., Trausmuth G., Architecture Recovery in ARES, *Proceedings of the Second International Software* Architecture *Workshop (ISAW-2)* , November 1996.
[8] Harris D.R., Reubenstein H.B., Yeh A.S.,Reverse Engineering to the Architectural Level,in *Proceeding of ICSE-17*, IEEE ComputerSociety Press, pp. 186-195,April 1995.

Shaheda Akthar received bachelor of Computer Science from Acharya Nagarjuna University, M.Tech Computer Science from BITS Pilani and persuing Ph.D from Acharya Nagarjuna University. Presently working as Assoc.Professor in Dept of Computer Science in Mittapalli College of Engineering affiliated to JNTU Kakinada.