DESIGN STRUCTURE AND ITERATIVE RELEASE ANALYSIS OF SCIENTIFIC SOFTWARE

AHMED TAHSIN ZULKARNINE
Bachelor of Science in Computer Science and Information Technology,
Islamic University of Technology (Bangladesh), 2006

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

DESIGN STRUCTURE AND ITERATIVE RELEASE OF SCIENTIFIC SOFTWARE

AHMED TAHSIN ZULKARNINE

Approved:

Signature	*Date*
______________________________	__________
Supervisor:	
______________________________	__________
Committee Member:	
______________________________	__________
Committee Member:	
______________________________	__________
External Examiner:	
______________________________	__________
Chair, Thesis Examination Committee:	

I dedicate this thesis to my wife and parents.

Abstract

One of the main objectives of software development in scientific computing is efficiency. Being focused on highly specialized application domain, important software quality metrics, e.g., usability, extensibility ,etc may not be amongst the list of primary objectives. In this research, we have studied the design structures and iterative releases of scientific research software using Design Structure Matrix(DSM). We implemented a DSM partitioning algorithm using sparse matrix data structure Compressed Row Storage(CRS), and its timing was better than those obtained from the most widely used C++ library boost. Secondly, we computed several architectural complexity metrics, compared releases and total release costs of a number of open source scientific research software. One of the important finding is the absence of circular dependencies in studied software which attributes to the strong emphasis on computational performance of the code. Iterative release analysis indicates that there might be a correspondence between "clustering co-efficient" and "release rework cost" of the software.

Acknowledgments

I take much pleasure to express my profound graditude to my super Dr. Shahadat Hossain, Associate Professor, University of Lethbridge for his inspiring guidance, close supervision and helpful suggestions throughout the period of my Masters Degree.

I would also like to thank my M.Sc supervisory committee members Dr. Robert Benkoczi and Dr. Saurya Das for their valuable suggestion and guidance.

It give me immense pleasure to cordially thank NSERC and University of Lethbridge for the financial and travel support. I am also thankful to my fellow graduate students Shah Mostafa Khaled, Tariqul Alam Sabbir, Mahmudul Hasan, Sangita Battacharjee and Ben Burnett for their spontaneous cooperation and encouragement.

Finally I would like to thank my wife, Shabnam Sultana for her support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

“Whenever science makes a discovery, the devil grabs it while the angels are debating the best way to use it!”

These are the verses of Alan Valentine, a player and coach of American 1924 Olympic champion Rugby team. Scientific research software are the devils of such kind of scientific discoveries in modern era. Though a number of scientific research software have been developed as proof-of-concept tool, powerful hardware resources facilitated scientific software to solve and simulate large problem. Sometimes these problems are as large as current technology allows. Starting from scratch often is not a wise option as the scientific simulation software are highly complex and large. Thus, scientific software development involves a substantial amount of time and other essential resources. The concept of iterative release helped them to have lifecycle measured in decades. Scientific research software are generally developed by people working in highly technical, knowledge rich professions who have no formal background in modern scientific engineering principles [56]. Typically this includes financial mathematicians, scientists and engineers who develop their own software to achieve professional goals [56]. Production of new scientific knowledge is one of the main objectives. Efficiency is core speciality of such software. At the origin of this thesis stood a series of unanswered questions regarding the design structure of scientific software such as modularity, sensitivity, etc. In order to answer those questions, we study the design structure and iterative releases of scientific research software.

The design structure matrix (DSM) has been used in this thesis as a tool for analyzing and comparing design decisions and quantifying structural metrics [35] [32] [60] [59]. DSM is a square matrix with identical number of rows and columns where each off-diagonal mark represents dependency between two design elements. DSM partitioning is considered as one of the first DSM computational problem [70] [39]. Tarjan [64] developed the first asymptotically optimal algorithm for partitioning the DSM. Many linear time algorithms now exist for solving DSM partitioning problem that originated from three core algorithms: Tarjan's algorithm, Cheriyan-Mehlhorn-Gabow algorithm [14] and Kosaraju-Sharir algorithm [57]. The DSM computed for majority of application areas contain large proportion of zero entries. As a result, the computation is expensive in terms of time and memory using traditional graph data structure. Hossain [29] proposed an efficient way for partitioning the DSM using sparse matrix data structure with the help of Tarjan's algorithm. This thesis discusses an efficient implementation of DSM partitioning algorithm proposed by Hossain. Hence, DSM partitioning using sparse matrix data structure leads to savings in computational work and intermediate storage.

Managing architectural evolution of complex software systems requires the identification of dominant subsystem and their dependency analysis [35] [32] [60] [59]. However, as the design goal and development nature of scientific software are somewhat different than general purpose commercial software, study of design structure for scientific software has become an important domain of research. MacCormack *et al.*[35] applied the DSM technique to study dependencies between system elements of two large scale software applications: *Mozilla* and *Linux*. Open source Linux was identified having more modular architecture than Proprietary Mozilla reflecting geographically distributed nature of Linux development team.

The **main contribution** of this thesis is an analysis of the design structure of open source scientific computing software. We used a number of architectural complexity metrics and DSM technique to analyze the design structure. We have chosen automatic differentiation (AD), linear programming (LP) and mixed integer programming (MIP). Automatic differentiation software are primarily responsible for automatic computation of first and higher order derivatives for mathematical functions written using different computer programming languages such as C/C++. Numerous algorithms for solving scientific and engineering problems require the computation of derivatives of mathematical models. Hence, software tools implementing AD represents a branch in major scientific computing applications. On the other hand, for a given mathematical model with a list of requirements or constraints, linear programming (LP) software tries to find out the best possible outcome. LP has attracted entrepreneurs from different parts of business. It is now considered as a key tool for making business decision [63]. This fact influenced us to include LP into our analysis. When choosing the best possible outcome, in many situations fractional solutions are not realistic. MIP deals with mathematical models where some variables are required to be integer. Therefore, MIP software are taken into consideration. We choose four software packages: ADOL-C [67] and CppAD [7] as representation of AD software, DyLP [26] and BCP [36] as representation of LP and MIP software respectively.

One of the main goal for iterative release development in general purpose commercial software is to meet customer requirement. Special emphasis is also placed on early delivery of the product to reduce development cost. Brown [12] introduced a technique for making decision between early product delivery and fulfilling customer requirements. Whereas, the prime motivation of iterative release development in scientific software involves feature enhancement, computational performance improvement,etc. Hence, iterative release analysis of scientific software is an interesting case study. In this thesis we perform an iterative release analysis of the software from three above scientific computing domains. We used a variation of Brown's [12] technique to quantify total implementation cost of a new release due to rework. Thirty seven releases from four above scientific software were investigated. The outline of the remaining chapters proceed as follows:

In Chapter 2, we present background materials relevant to this thesis. We review preliminary graph theory concepts, fundamental data structures and two graph algorithms for finding shortest path and strongly connected components.

In Chapter 3, We first describe different kind of source code dependencies and ways to extract them for scientific software. We then introduce DSM, different types of DSM and partitioning of DSM. Later we describe DSM partitioning algorithm using sparse matrix data structure Compressed Row Storage(CRS) proposed by Hossain [29]. Finally, we give the computational results and timing along with implementation details.

In Chapter 4, we introduce architectural complexity metrics that we used to analyze the design structure of scientific software. This includes characteristic path length, clustering coefficient, propagation cost, nodal degree, partitioned DSM analysis, degree distribution analysis and centrality measure.

In Chapter 5, we feature a variation of Brown's [12] technique that we used to quantify total implementation cost of iterative releases. We also described how we compute the implementation cost of new element, release rework cost accommodating this new architectural elements and finally the total implementation cost.

In Chapter 6, we present experimental results from thirty seven releases of four scientific software. We provide comparison of structural properties, structural metrics and total implementation cost for these iterative releases.

Lastly, in Chapter 7, we provide concluding remarks and some proposed directions for future research in this sector.

Chapter 2

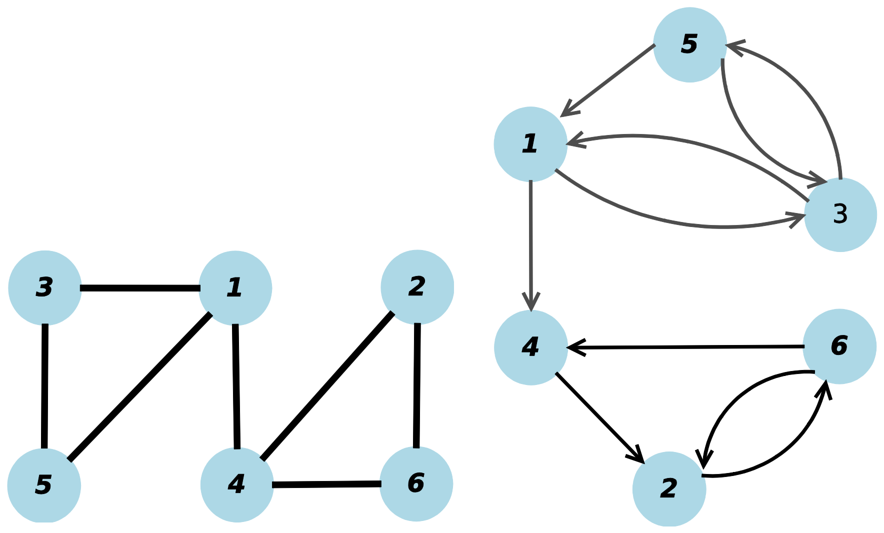
Background

In this chapter we briefly discuss regarding background concepts for analyzing the design structure of scientific software. We begin this chapter reviewing preliminarily graph theory concepts. Section 2.2 and section 2.3 review running time performance analysis of a computational algorithm and some fundamental data structure respectively. In section 2.4 and 2.5, we discuss two classical graph problems: finding shortest path and strongly connected components.

2.1 Basic Graph terminologies

The classical Konigsberg Bridge Problem solved by Leonhard Euler in 1736 was the first recorded evidence of the use of graph [28]. Seven bridges were used to interconnect four land areas. The problem was to determine a path so that one could cross each of the seven bridges only once and return to starting land area [3]. Graphs have been used in a wide range of applications since then. In graph theory, a graph $G = (V, E)$ consists of two sets V and E. The set V is a finite set of vertices. The set E represents the pairwise relationship between the vertices in V. This pairwise relationship is called edge. For each edge $e = (u, v)$ where the two endpoints $u, v \in V$ are said to be *neighbours* or *adjacents* to each other. A *weighted graph* is a graph where a number is associated with an edge. These numbers are called *weight* or *cost* [33].

Graphs can be directed or undirected. In an *undirected graph*, the vertices $u, v \in e$ are unordered. Therefore, the pair (u, v) and (v, u) represents the same edge. Whereas an edge $e = (u, v)$ in *directed graph*, u is the *tail* and v is the *head* of the edge. So, the pairs (u, v) and (v, u) represent two different edges in the graph. If the tail and head are the same vertices for an edge, then its called a *loop*. Directed graphs are also called *digraphs*. Figure 3.1 displays an undirected graph and directed graph.



(a) An undirected graph (b) Digraph

Figure 2.1: Vertices are shown in circles and edges are the lines connecting the vertices. A graph $G = (V,E)$ in (a), where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,3),(1,4),(1,5),(2,4),(2,6),(3,5),(4,6)\}$. In (b) a graph $G = (V,E)$, G has the same set of vertices V as (a) but $E = \{(1,3),(1,4),(2,6),(3,5),(3,5),(4,2),(5,1),(5,3),(6,2),(6,4)\}$

Subgraph G' of G is a graph such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$ where $E(G') = \{(u,v) \in E(G) | u,v \in V(G')\}$. A *path* from vertex u and v of a graph G is defined as the sequence of adjacent vertices $u,i_1,i_2,...,i_k,v$ such that $(u,i_1),(i_1,i_2),...(i_k,v)$ are edges in $E(G)$. If G is directed, the path consist of directed edges $(u,i_1),(i_1,i_2),...(i_k,v)$ in $E(G)$. The *length* of a path is the number of edges in the path. A path is *simple*, if all the vertices in the path are distinct. For example, consider a path from vertex 1 to 6 in Figure 2.1(a), it contain vertices $1,4$ and 6 and edges $(1,4),(4,6)$. Hence, the path is simple and length of the path is two. Similarly a directed path from vetex 1 to 6 in Figure 2.1(b), contain vertices $1,4,2,6$ and edges $(1,4),(4,2),(2,6)$. This path is also simple and path length is three. An *induced subgraph* is the graph defined on a subset of the vertices $V(G') \subseteq V(G)$ of a graph $G = (V, E)$ together with edge set $E(G') \subseteq E(G)$ where $E(G') = \{(u, v) \in E(G) | u \in V(G'), v \in V(G')\}$ whose endpoints are both in this subset [68].

A *cycle* is path in which first and last vertices are same. For example, the path with vertex sequence $4, 2, 6, 4$ is a cycle in Figure 2.1b. A graph without any circles is called *acyclic* graph. A *tree* is a connected acyclic graph. The vertex at level 0 of a tree is called *root*. If an edge (u, v) is in a path from some root vertex to vertex v, then u is said to be the *parent* of v and v is called the *child*. A *binary tree* is a tree in which every vertex has no more than two children and each child is designated as *left child* or *right child*. *Directed Acyclic Graph (DAG)* is a directed graph which does not contain any cycles.

The *degree* of a vertex v is the number of edges incident to that vertex v in a graph $G = (V, E)$. For a directed graph $G = (V, E)$, the *in-degree* of vertex v is the number of edges in which v is the head. Similarly, the *out-degree* of vertex v is the number of edges where v is the tail. Vertex 1 in Figure 2.1a has degree 3 and in Figure 2.1b, vertex 1 has in-degree 2 and out-degree 2.

2.2 Performance Analysis

To measure the efficiency of an algorithm, computer scientists use three notations: O called big oh, Ω called big omega and θ called big theta [33]. The running time of the algorithm is denoted with a function $t(n)$ for input size n and another function $g(n)$ is used to compare with it. A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if there exists some positive constant c and some non-negative integer n_0 such that

$$t(n) \leq cg(n), for\ all\ n \geq n_0 \tag{2.1}$$

$O(n)$ actually refers to an upper bound of the running time. A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$ there exists some positive constant c and some non-negative integer n_0 such that

$$t(n) \geq cg(n), for\ all\ n \geq n_0 \tag{2.2}$$

$\Omega(n)$ tells us about a lower bound of the running time. For example, so $n^3 \in \Omega(n^2)$ where $c = 1$ and $n_0 = 0$. A function $t(n)$ is said to be in $\theta(g(n))$, denoted $t(n) \in \theta(g(n))$ there exists some positive constant c_1 and c_2 and some non-negative integer n_0 such that

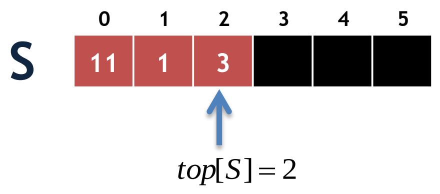
$$c_1g(n) \leq t(n) \leq c_2g(n), for\ all\ n \geq n_0 \tag{2.3}$$

$\theta(n)$ tells us about a tight bound of the running time.

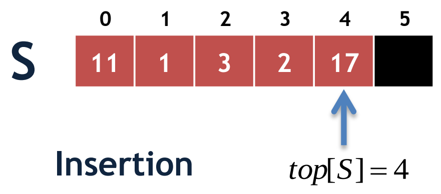
2.3 Stacks and Queues

Levitin [33] defined *data structure* as a 'particular scheme of organizing related data item'. One way to improve algorithms is structuring the data in such a manner that the resulting operation can be carried out efficiently [28]. In this section we will be familiar with stacks and queues.

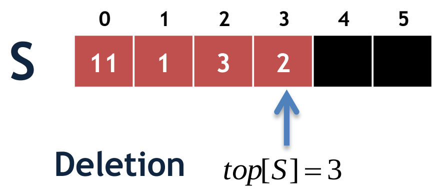
A linear list of elements is a sequence of n items of the same data type that are store continuously in computer memory. Often written as $a = (a_0, a_1, a_2, ...a_{n-1})$ where a_i are elements of the linear list for $i = 0, 1, ...n - 1$ and i is the index of the element in the linear list. This is also called an one dimensional *array*. A *stack* is a linear list where all insertions and deletions are made at one end called the *top*. An example of this can be a stack of dishes where the top dish is taken out from the stack for serving and also if any new dish comes in, it is placed in the top of the stack. For this reason, stack is called **L**ast **I**n **F**irst **O**ut **(LIFO)** list.



(a) Original Stack



(b) Insertion



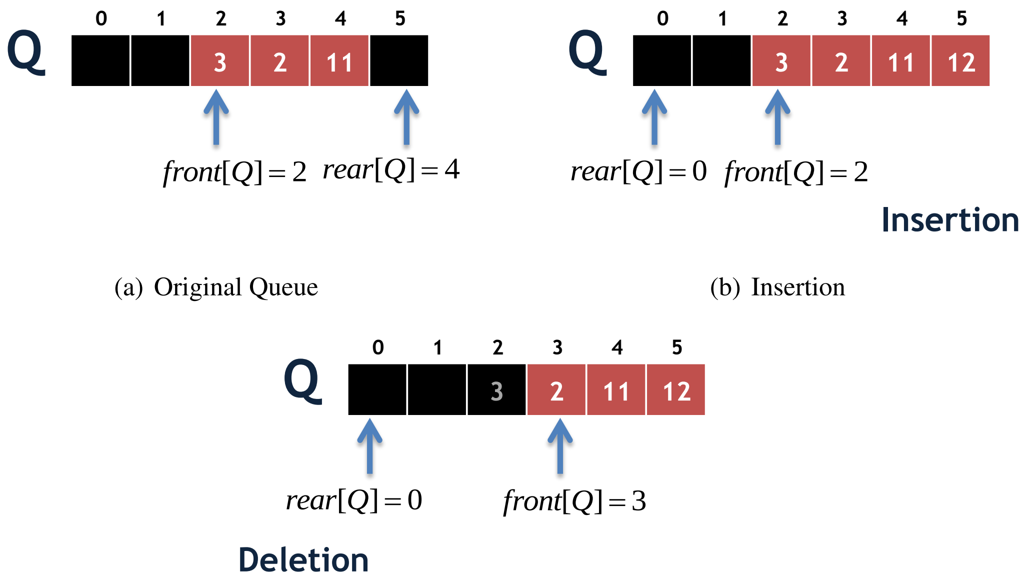
(c) Deletion

Figure 2.2: An array implementation of stack S [17]

One of simplest way to represent stack is using one-dimensional array. Figure 2.2 describes such an implementation. Stack S is an array $S[0,1,..n-1]$ where n is the capacity of stack. An attribute $top[S]$ indexes to the top element of the stack. Whenever any element is inserted, $top[S]$ is incremented. Similarly when any element is deleted, $top[S]$ is decremented. When $top[S] = -1$, the stack contains no element and is empty. In Figure 2.2, initially the stack contains 3 element. After inserting of two new element, $top[S]$ points to the 5th element. Analogous to insertion, upon deletion of an element $top[S]$ is decremented and now points to 4th element of the stack.

A *queue* is also an linear list where all insertions take place at one end called the *rear* but all deletions take place on the other end called *front*. Single queue in-front of a cash register at coffee shop can be an example. The customer at the front at the queue will be served first . New customers will be at the back of the queue and will be served last. Hence, queues are know as **F**irst **I**n **F**irst **O**ut **(FIFO)** lists.

Analogous to stack implementation, the simplest implementation of queue can be done



(a) Original Queue (b) Insertion (c) Deletion

Figure 2.3: An array implementation of queue Q [17]

using an one dimensional array. Queue Q is an array $Q[0,1,..n-1]$ where n is the capacity of queue. The Queue Q is treated as if it is circular [28]. Two attribute $front[Q]$ and $rear[Q]$ indexes to the front element of the queue and next free position in the queue respectively. Whenever any element is inserted, $rear[Q]$ is incremented. When $rear[Q] = n-1$, upon insertion of the new element, $rear[Q]$ is set to 0. Similarly when any element is deleted, $front[Q]$ is incremented. When $front[Q] = n-1$, with the deletion of any element, $front[Q]$ is set to 0. The queue is empty if and only if $rear[Q] = front[Q]$. Figure 2.3 displays the insertion and deletion operation of a queue.

2.4 Single Source Shortest Path Problem

Single source shortest path problem is one of the most interesting and widely used graph problem. Highway structure of a country can be represented as a graph where vertices are cities and edges represent the highways that connect the cities. [28]. Weight assigned to an edge can be the distance between two cities. A motorist wishes to travel from Edmonton to Vancouver and he has a road map of Canada. His prime quest will be to determine a shortest route from the starting vertex, *source* to last vertex, the *destination*.

Given a weighted directed graph $G = (V,E)$ with a weight function $w : E \to R$ mapping edges to real-valued weight, the weight of path $p = (v_0, v_2, ...v_k)$ is $\sum_{i=1}^{k} w(v_{i-1}, v_i)$ [28]. The single source shortest path problem is to determine the minimum weighted path from a source vertex, v_0 to all the remaining vertices in G.

Algorithm 1 Dijkstra's algorithm [28]

```
Input: Directed graph $G = (V,E)$ with non-negative weights and a source vertex, $s$. G is
  represented by an adjacency matrix $cost[1,..n,1,..n]$
Output: distance from source array, $dist[1,2,...n]$
  Begin
  initialize boolean array $S$ of size $n$;
  initialize array $dist$ of size $n$;
  for each vertex $v$ in $V$ do
      $S[v] \leftarrow$ false;
      if $cost[s,v] = 0$ then
          $dist[v] \leftarrow +\infty$;
      else
          $dist[v] \leftarrow cost[s,v]$;
      end if
  end for
  $S[s] \leftarrow$true;
  $dist[s] \leftarrow 0$;
  for $i = 2 \to n-1$ do
      Choose $u$ from among the vertices not in S such that $dist[u]$ is minimum;
      $S[u] \leftarrow$ true;
      for each vertex $w$ adjacent to $u$ with $S[w] = false$ do
          if $dist[w] dist[u] + cost[u,w]$ then
              $dist[w] \leftarrow dist[u] + cost[u,w]$;
          end if
      end for
  end for
  End
```

Various algorithm exists for finding the shortest path problem. However, the optimal running time algorithms originated from two major algorithms: *Bellman-Fold algorithm* and *Dijkstra algorithm*. *Bellman-Ford algorithm* solves the problem where edge weight may be negative but for *Dijkstra algorithm* solution, all edges weight are required to be non-negative. Nevertheless, with a good implementation Dijkstra algorithm runs faster than Bellman-Ford algorithm [17]. Call graph have non-negative weight assigned to its edges. Hence, in our research we used Dijkstra algorithm for finding the shortest paths.

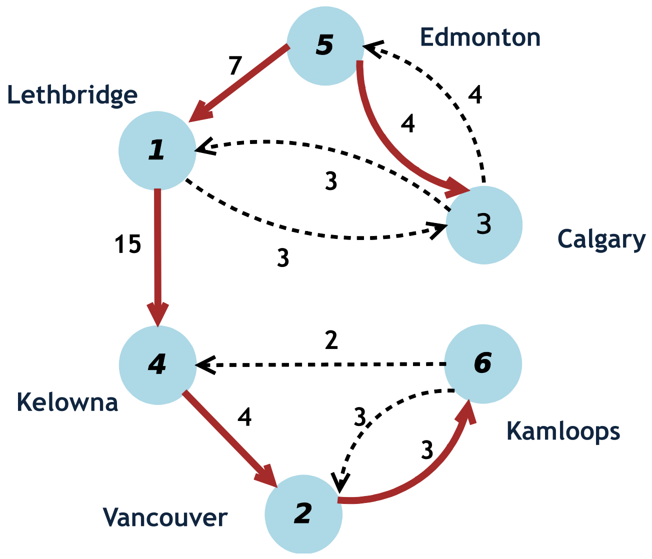


Figure 2.4: A weighted digraph with source as vertex 5. The bold edges form a shortest path tree from the source.

Algorithm 2.1 contains the pseudocode of Dijksta's algorithm. The graph $G = (V, E)$ is represented using an adjacency matrix *cost*. The array *dist* captures the distance from source for each vertex. Boolean array S detects whether a vertex resides in the current shortest path. The algorithm proceeds by traversing from the source vertex to the closest vertex u which is not in S. The vertex having the minimum value in array *dist* is the closest vertex. On each iteration, the algorithm finds a closest vertex. Later, for each adjacent vertices w of u which are not in S, the algorithm computes the path cost from source to w

Iteration	S	Vertex selected	Distance					
			Let	Van	Cal	Kel	Edm	Kam
			1	2	3	4	5	6
Initial	--	--	7	+∞	5	+∞	0	+∞
1	5	3	7	+∞	5	+∞	0	+∞
2	5,3	1	7	+∞	5	22	0	+∞
3	5,3,1	4	7	24	5	22	0	+∞
4	5,3,1,4	2	7	24	5	22	0	27
5	5,3,1,4,2	6	7	24	5	22	0	27
6	5,3,1,4,2,6							

Figure 2.5: Execution of the Dijksta's Algorithm on digraph in Figure 2.5

through u using $dist[u] + cost[u,w]$. If the path cost from source to w through u is less than $dist[w]$, $dist[w]$ is updated with $dist[u] + cost[u,w]$. These process is named as "relaxation" by Cormen *et al.*[17]. The algorithm terminates when all of the vertices that are reachable from source are put in the shortest path. Figure 2.6 provides a complete execution of Dijksta's algorithm on Figure 2.5. With the help of a good data structure such as priority queue, the time efficiency of this algorithm can be achieved in $O(|E|log|V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertices in a graph $G = (V,E)$. A priority queue [28] is like a queue with an additional feature where each element is associated with a priority. With an advanced data structure such as fibonacci heap, the running time can be lowered to $O(|V|log|V| + |E|)$. A fibonacci heap is a collection of trees where the value associated with the child node is greater than or equal to the value associated with parent node.

2.5 Strongly Connected Components

A connected component of a graph $G = (V,E)$ is a maximal set of vertices $C \subseteq V$ such that every pair of vertices u and v, there is a path from u to v and also from v to u [17]. Analogous to connected component, for directed graph we have strongly connected components. A strongly connected component of a directed graph $G = (V,E)$ is a maximal set of vertices $C \subseteq V$ such that every pair of vertices u and v, there is a directed path from u to v and also from v to u [17]. In other words, all vertices in strongly connected sub-graph are reachable from each other. The term *maximal set* defines that a strongly connected component can not be enlarged by introducing some other extra nodes. For a directed graph, each vertex belong to a single strongly connected component.

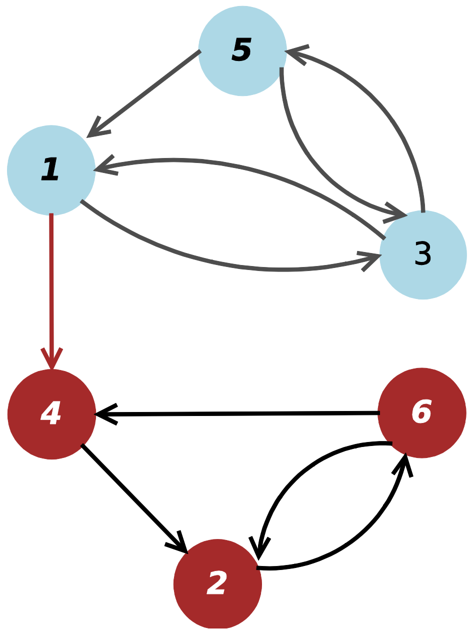


Figure 2.6: A digraph with strongly connected components

Figure 2.7 has two strongly connected components labeled with light and dark colors. The first component consist of vertices:$\{1,3,5\}$ while the second one contains vertices:

$\{2,4,6\}$. All pair of vertices within each component is mutually reachable. None of the strongly connected component can be extended by adding any more vertices. In addition, each vertex resides to a single strongly connected component.

Many linear-time algorithm exist for finding strongly connected components due to the help of three major algorithms: Tarjan's algorithm, the Cheriyan-Mehlhorn-Gabow algorithm , and the Kosaraju-Sharir algorithm [38]. All of them exploit a technique name **D**epth **F**irst **S**earch **(DFS)**. DFS is one of the core algorithms for traversing vertices and edges of a graph in a systematic manner [33]. As the name suggest, the DFS goes deeper in the graph whenever possible for searching unexplored edges [17] . DFS initiates traversing vertices of a graph at an arbitrary vertex and marking it as visited. The algorithm proceeds to an unvisited vertex on each iteration with is adjacent with the currently visited vertex. The algorithm backtracks when any dead end found . It backs up one edge to the vertex it came from and proceed to continue to another unvisited vertex if possible. The algorithm finally terminates when all of the edges are explored. In our research we used Tarjan's algorithm with a sparse matrix data structure. Chapter 3 contains a detail discussion regarding strongly connected component and DFS.

Chapter 3

Modelling Dependencies

In this chapter we illustrate different kind of source code dependencies. In section 3.2 we discuss how we extract the dependency from the source code. Section 3.3 and 3.4 review Design Structure Matrix (DSM) and sparse matrix data structure. Finally we conclude this chapter by describing the performance of our DSM partitioning implementation.

3.1 Source Code Dependencies

A large and complex set of dependencies exist between the modules of software systems [65]. Software systems are composed of one of more independently developed modules where each module is a segment of the software. Manytimes, developers require the understanding of a software system they are unfamiliar with [27]. Parnas [48] defined the primary concept of software dependency as a “uses” relationship. A software module A uses another software module B if there exist situations in which the correct functioning of A depends upon availability of a correct implementation of B.

Software dependency can be static or dynamic. Static dependencies, generally known as “compile time” dependency use the concept that one module is required to compile another module [32]. In other words, a static software dependency is a dependency intended to represent every possible run of the software. Software module A depends on B if the source code of A makes an explicit reference to B.

On the other hand, dynamic dependencies knowns as “run-time” dependency is a record of an execution of the program [32]. In other words, it is generated based on actual call pattern of the software during operation. Though it provides the run-time scenario, there is a potential disadvantages attached to it when analyzing the design structure. Not all subroutines are executed at run-time so complete code dependency in the software will be missing to some extent. Suppose, subroutine S_1 depends on subroutine S_2, but at run-time S_1 is not executed. Consequently the dependency of S_1 and S_2 will be missing in dynamic call graph.

In this research, we are interested in static graphs. This is because we need to extract call relations corresponding to occurrence of subroutine calls in the source code instead of call relations extracted from actually running the program. Also, we need to encapsulate every dependency that exists in the software system in order to get actual scenario. For this reason, we focused on analyzing static software dependencies.

A software system can be modeled as a *call graph* [9]. Bisselling [9] defined the call graph as a directed graph, where vertices may be program, classes or functions and an edge (u, v) means program v calls program u. Call graphs are well-known instruments for understanding complex software systems and are of great help in maintaining a system [27]. For example, consider the following small C/C++ program for calculating the area of a circle:

```
void print(double area)
        {
        cout << "The area of circle is :"<< area << endl;
        }

double get_area()
        {
        double k=get_radius();
        double area= 3.142 * k * k;
        }
```

```
double get_radius()
        {
        double radius;
        cout << "Please provide the radius of the circle: " << endl;
        cin >> radius;
        return radius;
        }

void main()
        {
        double area=get_area();
        print(area);
        }
```

Figure 2.8 displays a static call graph of the example C/C++ program. Nodes are the user-defined functions and edges are the function call.

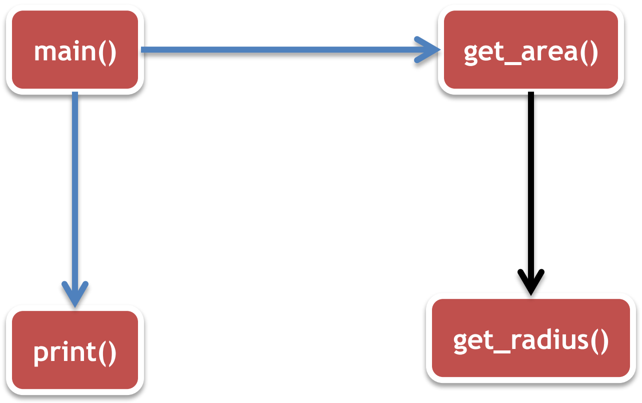


Figure 3.1: A static call graph associated with C/C++ program to calculate area of a circle

3.2 Dependency Extraction

Examining program dependencies is challenging for large system[65]. In this thesis, the focus is on C/C++ code base. In terms of complexity, C++ comes ahead of many other programming languages as its standard has evolved over the years. Extraction and visualization of call graphs for C/C++ programs is a well-known research area which focus on two domain: Source code dependency extraction and extracted data visualization [65]. Numerous tools exist for extracting and visualizing call graphs for facilitating software engineers to understand the program [41].

Telea *et al.*[65] identified the two main classes of call graph extractor: *Lightweight* and *Heavyweight* extractors. Lightweight extraction provides a fraction of entire static information as it does partial parsing and type checking. XML based extractor *SRCML* [16], *GCCXML*, etc are some examples of lightweight extractor. On the other hand, heavyweight extractors provide nearly a complete call graph by performing full parsing and type checking. Extractors like *CPPX* [34] and *OINK* [69] are heavyweight extractors. Moreover, the heavyweight extraction can of two types: strict and tolerant . Strict heavyweight extractors are based on compiler which stops when there is a lexical or syntax error [65]. However, tolerant ones are based on Fuzzy or Generalized Left Reduce (GLR) parsing. GLR parsing is one of the most efficient parsing for context free grammar [43]. The tool *OINK* is a tolerant one while *CPPX* is strict. Hence, tolerant heavyweight extractor are more probable of providing complete static call graphs. OINK is one of the most complete, robust and scalable open-source static analyzer for C/C++ programs [69].

Numerous tools exist for visualization of call graph but very few optimize layout and graph data management for very large graphs [65]. One such system is Tulip graph visualization framework [4]. Tulip is an information visualization framework developed by LaBRI, University of Bordeaux I, France and is dedicated to the analysis and visualization of relational data [4]. This framework written in C++ has a development of over 10 years and one of the most sophisticated graph visualization framework available in market[65].

3.2.1 Unit of Analysis

The dependencies between source code were captured by examining the ”function calls”. A function call can be defined as an instruction that requests a specific task to be executed by a program [9]. McCormack *et al* [35] used the source code file as the basic unit of analysis using function calls. Though our research work is roughly, not exactly equivalent to McCormack, the function calls were given preference as this type of dependency is at the kernel of many analysis tool and also have been used in prior work that examines system structure [35]. Hence in our research, functions are the basic unit of analysis. For example, if Function i calls Function j , then this dependency is marked by an directed edge $i \rightarrow j$ in the associated directed call graph.

3.2.2 Tools used

Telea [65] described an entire tooling pipeline that covers static code analysis, extraction of calls, hierarchy data and their attributes. His research team implemented a standalone call graph extractor based on OINK framework. In order to extract the static call graph we required an open-source tolerate heavyweight static analyzer for C/C++. Hence, we used the OINK-based call-and-structure extractor developed by his research team from University of Groningen, the Netherlands [65]. One of important enhancement the call graph extractor did over OINK framework includes linking declaration to definition across multiple translation unit. Translation units are the basic units of compilation in C++. They consist of content of source files and header files but excludes code which are ignored in conditional preprocessing statements. For each translation unit, the call graph extractor scans each function declaration and links it to function definition in same translation unit or another translation unit. Another feature enhancement is the detection of potential set of called candidate for virtual function and function pointers.

The call graph extractor package is a complete call graph construction and exploration tool chain, consisting of an information extractor (CCIE) and a call graph constructor(CCC) [27]. The information extractor locates all function definition, function declaration and program hierarchy (folders, files, classes, namespaces,etc) [65]. The call graph extractor scans these elements and find out the links between these items. The linking provides us with a call graph where nodes are functions, folders, namespaces and files. The edges are function calls and relations in program hierarchy. The call graphs can be exported into a number of format including tulip file format, *.tlp. Hoogendorp's [27] article contains a detailed discussion regarding the call graph extractor package.

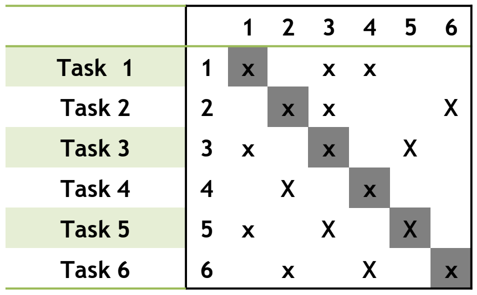
In this research, we developed an Statical Toolkit for Software Code Base (STSCB). Our developed toolkit take call graphs $G = (V, E)$ as input in *.tlp format and is also open to another graph format *.vcg. "Visualization of Compiler Graphs" (VCG) is a graph format developed for the visualization of graphs from the area of compiler design [11]. The output by the call graph extractor provides vertex list containing functions, files, directory, classes and namespace. Moreover, functions can be of two types: *system functions* and *user-defined functions*. Functions that are explicitly implemented in the software are called *user defined* where as the functions that are part of the software libraries are *system functions*. The first task of our toolkit STSCB is to segregate user-defined function from the vertex list and build a induced subgraph $G' = (V, E)$ where $V(G')$ is the user-defined functions and $E(G')$ contains the edges within the user-defined functions. The functions that are defined in a file that resides in the source code directory are considered as user-defined functions while the rest functions defined elsewhere are system functions. The induced sub-graph is constructed using efficient data structure stated in Chapter 3.4. The output of toolkit is a number of different structural metrics and version comparison of the softwares to be analyzed. Chapter 4 and 5 contains detailed discussion regarding these topics.

3.3 DSM

Complexity of product design has been a critical topic for both researchers and managers for many years. However, with the help of a reasonable model, it has become possible to explore approaches to understand the complexity of the product. This model known as "Design Structure Matrix (DSM)" was originated by Don Steward in 1981 [32] for understanding interactions between product design elements. Eppinger *et al.*[22] extended this model to visualize more elaborately the relationship between design structure and task structure in product development. Task structure is the method of breaking down larger task(s) into small logically interrelated operations. Later on, DSM has become one of the popular representation and analysis tool for system modeling, specifically in areas of decomposition and integration [13]. Some of the prime domains where DSM has been successfully applied are building construction, semiconductor design, automotive, aerospace, telecommunication, small-scale manufacturing, factory equipment and electronics [13].

A DSM displays the relationship between components of a system in a compact , visual and analytically advantageous format [13]. It is a square matrix having identical number of rows and columns. The dependency between one element with other is indicated with an off-diagonal mark. Figure 3.2 displays a simple DSM with 6 tasks represented with 6×6 square matrix. The diagonal elements are shaded. Scanning through a particular row reveals the element associated with that row depends on what other elements. Similarly, scanning the column indicates what other elements depend on the element associated with that column. Thus, from Figure 3.2, task 1 depends on 3 and 4 with first row scan. Similarly

3 and 5 depends on 1 with first column scan.



		1	2	3	4	5	6
Task 1	1	x		x	x		
Task 2	2		x	x			X
Task 3	3	x		x		X	
Task 4	4		X		x		
Task 5	5	x		X		X	
Task 6	6		x		X		x

Figure 3.2: A sample DSM

Chapter 2 provided a detailed discussion regarding graph theory. Associated with the DSM in Figure 3.2 is a directed graph $G = (V, E)$ where V is a set of 6 vertices and and a non-zero entry $a_{ij} \neq 0$, $i \neq j$ represents a directed edge from vertex v_i to vertex v_j denoted $(v_i, v_j \in E)$ in graph G. Figure 3.3 displays the graph associated with the DSM in Figure 3.2.

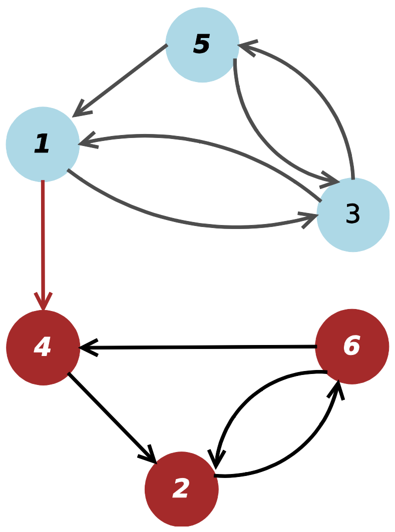


Figure 3.3: The directed graph associated with the DSM in Figure 3.2

Tasks listed in a DSM are placed in a chronological order, i.e early tasks in the upper rows. Hence, the entries below the diagonal represents feed-forward information while entries above diagonal provides feedback information. Rogers *et al.*[53] defined feed-forward information as the data computed before it is used and feedback information as the data required as input before it is computed. The edges (v_3, v_1) and (v_1, v_3) in Figure 3.2 illustrates the feed-forward and feedback marks for the associated DSM. Feedback marks are expensive for product design because they imply that early tasks require information from later or upcoming tasks [72]. Section 3.3.2 describes how we can minimize the number of feedback marks.

3.3.1 Types of DSM

Browning [13] identified two categories of DSM: static and time based.

- Static DSM: It represents a system where all the elements exist simultaneously. Modeled using a $N \times N$ matrix where N is the number of design elements, it enables system engineers to represent architectural components and interfaces, organization designers to document communication network, economist to visualize the effect of a change in one products on others and so on. This kind of DSMs are generally analyzed using clustering algorithms. Clustering is a process of finding subsets of DSM elements that are interconnected among themselves to an important extent and also minimally connected to the rest of the system [70]. The static DSM can be of two types: Architectural and Organizational.

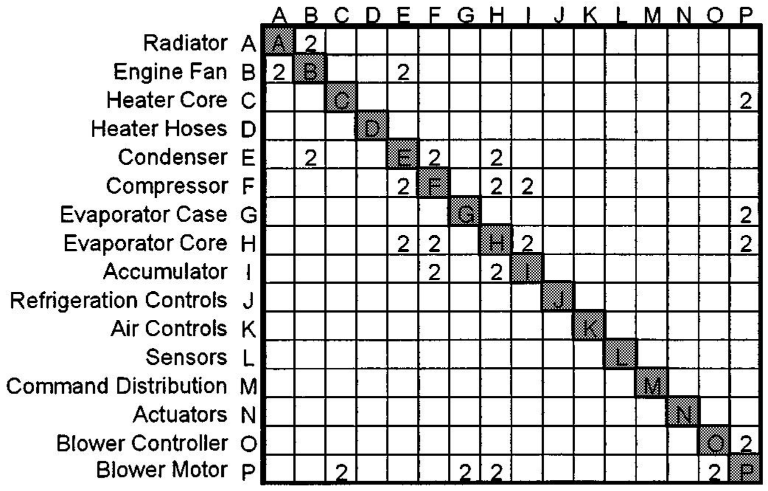
    1. Architectural DSM: A DSM that documents interactions among elements in a system architecture. Modular system architecture have advantages in simplicity and reusability for a product family or platform [5]. Thus system engineers decompose the system into subsystems using the following steps [13]:

(a) Decomposition of the system into basic elements.

(b) Documentation of the interaction between these basic elements

(c) Analysis of potential reintegration of the elements through clustering.

Figure 3.4 shows an automotive climate control system modeled by Pimmler and Eppinger [50] for Ford motor company. The weighting of the interactions with each other were facilitated by a quantification scheme (see Table 3.1). The numerical entry of +2 in the DSM implies every element do not interact with every other but all of the material that have interactions are essential for desired functionality [13]. The interaction between radiator and engine fan can be quantified with entry +2. The engine fan provides airflow to surroundings of the radiation. Both of them, the radiator and engine fan are closely located for design efficiency [50]. This is an example of architectural DSM.



		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Radiator	A	A	2														
Engine Fan	B	2	B			2											
Heater Core	C			C													2
Heater Hoses	D				D												
Condenser	E		2			E	2		2								
Compressor	F					2	F		2	2							
Evaporator Case	G							G									2
Evaporator Core	H					2	2		H	2							2
Accumulator	I						2		2	I							
Refrigeration Controls	J										J						
Air Controls	K											K					
Sensors	L												L				
Command Distribution	M													M			
Actuators	N														N		
Blower Controller	O															O	2
Blower Motor	P			2				2	2							2	P

Figure 3.4: Architectural DSM - Automobile Climate Control System [50]

2. Organizational DSM: Organizational DSM helps to model organization design. Better understanding of organization enhances innovation and improvement in

Required	+2	Physical adjacency is neccessary for functionality
Desired	+1	Physical adjacency is beneficial, but not neccessary for functionality
Indifferent	0	Physical adjacency does not affect functionality
Undesired	-1	Physical adjacency causes negative effect but does not prevent functionality
Detrimental	-2	Physical adjacency must be prevented to achieve functionality

Table 3.1: Interaction Quantification scheme [13]

organization design [13]. Similar to architectural DSM, the system can be decomposed into subsystem using three steps:

(a) Decomposition of the organization into teams with specific function, roles or assignments.

(b) Documentation of the interaction between these teams.

(c) Analysis of clustering of the teams.

McCord and Eppinger analysed an automobile engine development organization using team-based DSM shown in Figure 3.5 [37]. In order to accomplish the redesign of small block V-8 automotive engine at General Motors, 22 product development team were established. The frequency of interaction between the teams defined the level of dependence in the DSM. Several meeting in a week defined high dependency while weekly meeting for average and infrequent meeting for low dependency.

- Time-based DSM: This DSM represents a system which involves a time flow. Time based DSM is modeled using a precedence diagram which has been used for many years to manage projects. In order to achieve accelerated successful completion of product development process, concurrent design is used [71]. In addition, concurrent design facilitates improvement of product quality and development effort(time and cost) reduction [31]. However, when performing concurrent (parallel) tasks, de-

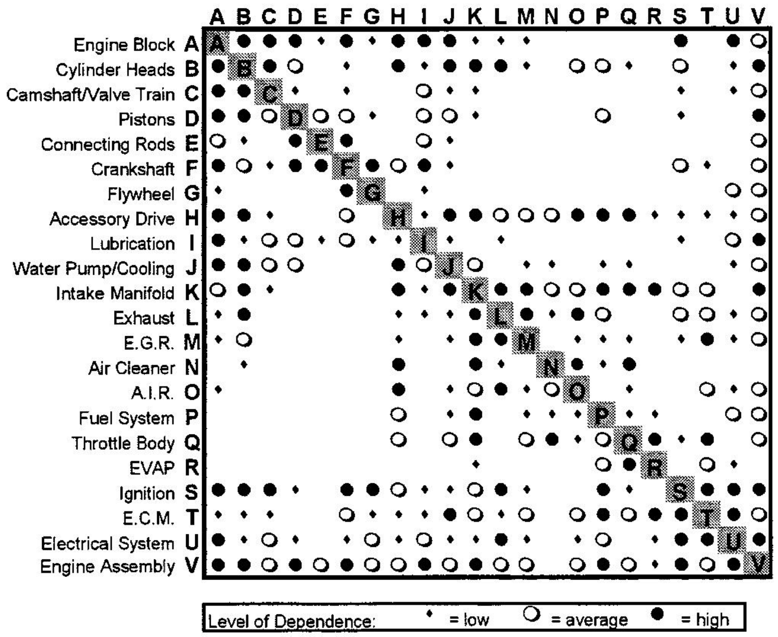


Figure 3.5: Organizational DSM - Automobile Engine Redesign [37]

signers/engineers may find they are missing their requisite input information [21]. Whenever a design activity proceeds with missing input information, feedback occurs. Meier *et al.*defined iteration as repeated design refinement due to information feedback [39]. Ballard stated iteration as type of waste in product design which needs to be avoided [6]. One way to manage iteration is to sequence the design activities to streamline information flow [39]. Sequencing algorithms are concerned with sequencing the iterative design elements. Generally sequencing algorithms are used to analyze time-based DSM. Similar to Static DSM, Time-based DSM can be of two types: Schedule and Parameter-based.

1. Schedule DSM: It provides a concise, visual format for understanding and analysing process design. The parallel activities which can be accomplished without causing additional iterations can be determined using these kind of

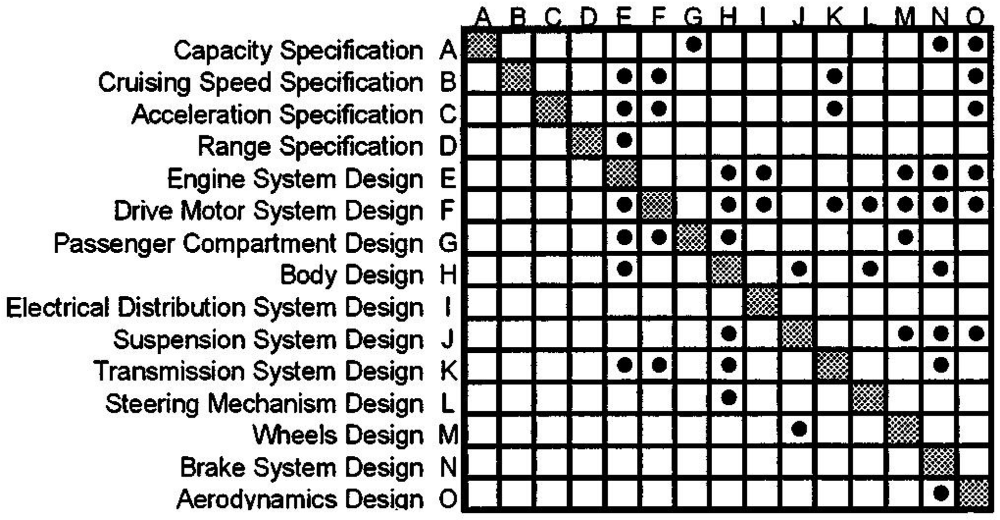
DSM. For example, putting on shoes consist of 5 activities: get socks, get shoes, inspect shoes, put on socks and put on shoes. The activities “get socks” and “get shoes” can be done in parallel. Like other DSM, the modeling of schedule DSM consists of three steps:

(a) Decomposition of the processes into activities.

(b) Documentation of the information flow among the activities.

(c) Analysis of the sequencing of activities.

A high-level description of a simplified automobile design process shown in Figure 3.6 was demonstrated by Kusiak and Wang [31]. Fifteen major subsystem of the automobile were identified where each subsystem corresponds to a set of activities.

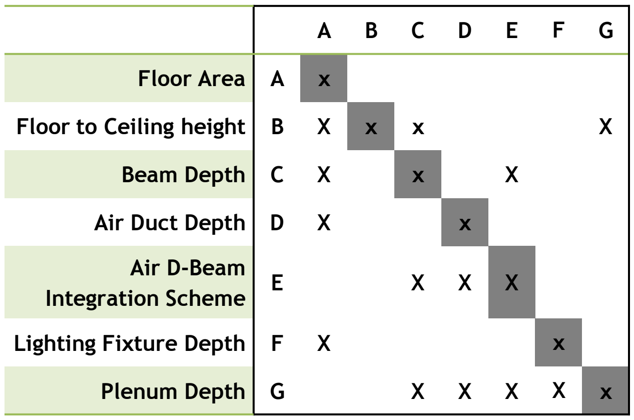


		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Capacity Specification	A	▨						●							●	●
Cruising Speed Specification	B		▨			●	●					●				●
Acceleration Specification	C			▨		●	●					●				●
Range Specification	D				▨	●										
Engine System Design	E					▨			●	●				●	●	●
Drive Motor System Design	F					●	▨		●	●		●	●	●	●	●
Passenger Compartment Design	G					●	●	▨	●					●		
Body Design	H					●			▨		●		●		●	
Electrical Distribution System Design	I									▨						
Suspension System Design	J								●		▨			●	●	●
Transmission System Design	K					●	●		●			▨			●	
Steering Mechanism Design	L								●				▨			
Wheels Design	M										●			▨		
Brake System Design	N														▨	
Aerodynamics Design	O														●	▨

Figure 3.6: Schedule DSM - Simplified automobile design process [31]

2. Parameter-based DSM: In terms of system design, Pektas [49] defined ‘parameter’ as ‘a physical property whose value determines a characteristic of behavior of a system component’. During a collaborative design, designers take decision on parameter values. For example, automobile designers might agree

on maximum speed of the car or specific dimension of driver's seat. Rouibah and Castay identified that these parameter decisions are the basis of a process [54]. A parameter-based DSM is a square matrix which captures the dependency between parameters[49]. One such example of parameter-based DSM is "suspended ceiling design" described by Pektas [49] shown in Figure 3.7. Six design parameter were identified for designing a suspended ceiling. In order to make decision of beam depth parameter, it requires information from parameter floor area and Air D-beam Integration scheme.



		A	B	C	D	E	F	G
Floor Area	A	x						
Floor to Ceiling height	B	X	x	x				X
Beam Depth	C	X		x		X		
Air Duct Depth	D	X			x			
Air D-Beam Integration Scheme	E			X	X	X		
Lighting Fixture Depth	F	X					x	
Plenum Depth	G			X	X	X	X	x

Figure 3.7: A Parameter-based DSM [49]

3.3.2 *Partitioning the DSM*

One of the first DSM computational problem is partitioning the DSM. Yassine defined partitioning as the sequencing (e.g reordering) of the DSM rows and columns such that the new DSM arrangement contains minimum number of feedback marks [70]. This enables a faster development process as fewer system elements will be involved in the interdependency cycle. Partitioning and Sequencing are two closely related DSM terms which were distinguished by Meier *et al.*[39]. Partitioning algorithm finds out a topological ordering if no cycle present. Otherwise, when cycle is present, it finds out the design elements which are present in the cycle but do not provide a sequence for them. On the other hand, sequencing algorithm finds an ordering of the design elements within the cycle. A matrix can be decomposed into a number of blocks where each block is a sub-matrix. A block lower triangular matrix is a special kind of square matrix where all blocks above the main diagonal are zero. Block lower triangular matrix indicates no feedback marks which provides the ideal arrangement of jobs in a project design. Hossain [29] stated a matrix A with non-zero diagonal entries is irreducible if and only if the graph associated with A, G(A) is strongly connected. Hence, main goal of partitioning is to transform the DSM in a block lower triangular form (bltf).

This DSM partitioning can be represented as a graph problem. With the background knowledge from previous chapter, we can say a graph can be represented using an adjacency matrix. *Subgraph* G' of G is a graph such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$ where $E(G') = \{(u,v) \in E(G) | u,v \in V(G')\}$ [28]. Recall from Chapter 2, a strong component of a directed graph $G = (V,E)$ is a maximal set of vertices $C \subseteq V$ such that every pair of vertices u and v, there is a directed path from u to v and also from v to u [17]. In other words, vertices u and v are reachable from each other. Thus, a sub-graph is strongly connected component if it is strongly connected and can not be extended to another strongly connected sub-graph by introducing extra nodes along with its edges. Hence, it can be concluded that each node can belong to only one strong component. Therefore these strong components partitions the graph. Let the strong components in a graph be $C_1, C_2, C_3, ...C_k$. If the strong components are identified and the nodes of C_1 labeled before those of C_2 and so on, then the associated matrix is block lower triangular where the block represents the strong components [20]. Hence it can be concluded that the DSM partitioning can be seen as a graph problem.

Many algorithm exists for finding the strong components. Graph theory text books dis-

Algorithm 2 Tarjan's algorithm [29]

```
Input: Directed graph $G = (V, E)$
Output: list of strongly connected components, $cmpt[1, 2, ...n]$
  Begin
  initialize array $root$ of size $n$;
  initialize array $dfsn$ of size $n$;
  create an empty stack $st$ of size $n$;
  initialize list cmpt;
  $compNum \leftarrow 1$;
  $dfsNum \leftarrow 1$;
  for each vertex $v$ in $V$ do
      if $v$ is not processed then
          $dfsn[v] \leftarrow dfsNum$; $dfsNum = dfsNum + 1$;
          $DFS(v)$ ;
      end if
  end for
  End
```

Algorithm 3 DFS [29]

```
Input: Vertex $v$
  Begin
  $root[v] \leftarrow v$;
  push $v$ onto st;
  for each edge $(v, w)$ in $E$ do
      if $w$ is not processed then
          $dfsn[v] \leftarrow dfsNum$; $dfsNum = dfsNum + 1$;
          $DFS(w)$ ;
      end if
      if $w$ is not assigned to any strongly connected component then
          if $dsfn[root[w]] < dfsn[root[v]]$ then
              $root[v] \leftarrow root[w]$;
          end if
      end if
  end for
  if $root[v]$ is $v$ then
      pop vertices $z$ from $st$ until $v$ is popped;
      for each $z$ popped do
          $cmpt[z] \leftarrow compNum$;
      end for
      $compNum \leftarrow compNum + 1$ ;
  end if
  End
```

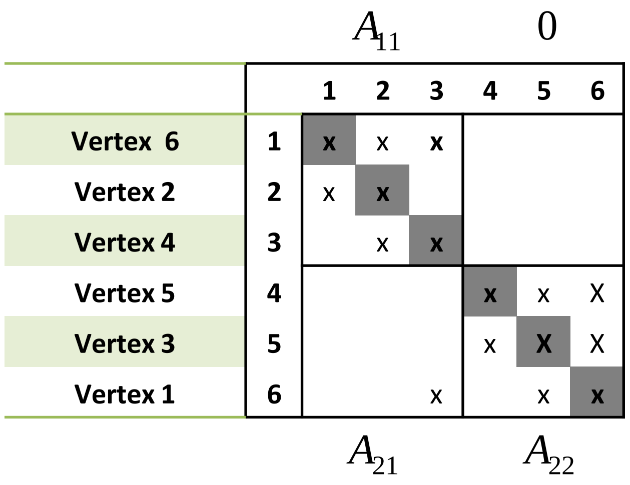
cuss ‘power method’ for finding the strong components. The advantage of ‘power method’ is simplicity but its not efficient [31]. DSM can be considered a binary matrix where dependency between design element can be represented by 1 and other wise 0. This $O(n^3)$ algorithm was introduced by Harary which relies on the principle of repeatedly multiplying the binary matrix with itself. Sargent and Westerberg developed an $O(n^2)$ algorithm for finding the strong components. The algorithm utilizes the fact that all of the nodes in any cycle of a graph lies in the same strong component. Once a cycle is found, the graph is modified where all the nodes in the cycle are combined together to form a composite node [20]. Any node or composite node with no outgoing edge is considered as a strongly connected component by Sargent and Westerberg. Tarjan [64] developed the first asymptotically optimal algorithm for identifying strong components in a way such that each arc is visited exactly once and each vertex is visited but no more than a constant number of times. In other words, it relies on depth-first-search(DFS) for finding the strong components. This algorithm with a good data structure runs is $O(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges. Various linear time algorithms originated from three major algorithms, all based on depth-first search: Tarjan’s algorithm, the Cheriyan-Mehlhorn-Gabow algorithm, and the Kosaraju-Sharir algorithm [38]. Cheriyan-Mehlhorn-Gabow algorithm uses two stacks to keep track of unassignned vertices to any component, and upon exploration of the final vertex of a new component, it moves unassigned vertices to that component [38]. On the other hand, Kosaraju-Sharir algorithm performs two passes of DFS, one on original graph and one on the modified graph.

Algorithm 2 provides the pseudo-code of Tarjan’s algorithm. The arrays `root` and `dfsn` of size n identifies the root and the ordering of dfs traversal for each vertex. The two variables `dfsNum` and `CompNum` is used to identify the current ordering of dfs traversal and component number. They are both initially assigned to 1. The stack `st` is used to identify and incrementally build strongly connected components during the depth-first search of the graph. The output of the algorithm is an array cmpt of size n which states in which component each vertex resides. The algorithm starts with an arbitrary vertex v, i.e. vertex v_1 of the sample graph in Figure 3.3. Assigns its ordering dfsn[v] with dfsNum, dfsNum is incremented and the recursive function *DFS* (presented at Algorithm 2) is invoked. The *DFS* algorithm initially assigns vertex v's root as itself and pushes v to the stack st. For each of its outgoing edge (v,w) , if w is not processed it assigns dfsn[w] with dfsNum, increments dfsNum and once again the DFS function is invoked with vertex w. Later if w has been not assigned to any strong component, the dfs traversal ordering of root[v] and root[w] is compared. If root[w]'s dfs traversal ordering is found less, the root of v is assigned with root of w. Finally whenever a root vertex is discovered (i.e. $root[v]$ is v) from the recursive call of DFS, all of the vertices including the root are popped out of the stack st. The array cmpt is updated with CompNum for each vertex popped out from the stack and compNum is increment to identify the next component.

For our sample graph after the execution of Tarjan's algorithm, dfsn $= \{1,5,2,4,3,6\}$, root $= \{1,4,1,4,1,4\}$ and cmpt $= \{2,1,2,1,2,1\}$. So vertex 1 and 4 are root vertices corresponding to the two strong components found. The first strong component found consist of vertices: $\{v_6, v_4, v_2\}$ with v_4 as root vertex while the second one consisting of vertices $\{v_5, v_3, v_1\}$ with v_1 as root. The ordering of the dfs traversal defines the permutation matrix that can be efficiently represented by a permutation vector. Using the array $dfsn$, we can say vertex v_6 has been popped up first from the stack (or traversed last in dfs) and vertex v_1 popped us last. Thus, the permutation vector P can be defined as $P = \{6,2,4,5,3,1\}$ with the help of dfsn.

In Figure 3.8 displays the original DSM permutation into a block lower triangular form using the help of permutation vector P from Tarjan's algorithm. The top-left diagonal block represents the first strongly connected component while the bottom-right represents the second one. The Tarjan's algorithm was first implemented by Duff and Raid in 1978

[20]. Their Fortran implementation discussion includes the complexity analysis and timing.


A_{11}
0
A_{21}
A_{22}

		1	2	3	4	5	6
Vertex 6	1	x	x	x			
Vertex 2	2	x	x				
Vertex 4	3		x	x			
Vertex 5	4				x	x	X
Vertex 3	5				x	X	X
Vertex 1	6			x		x	x

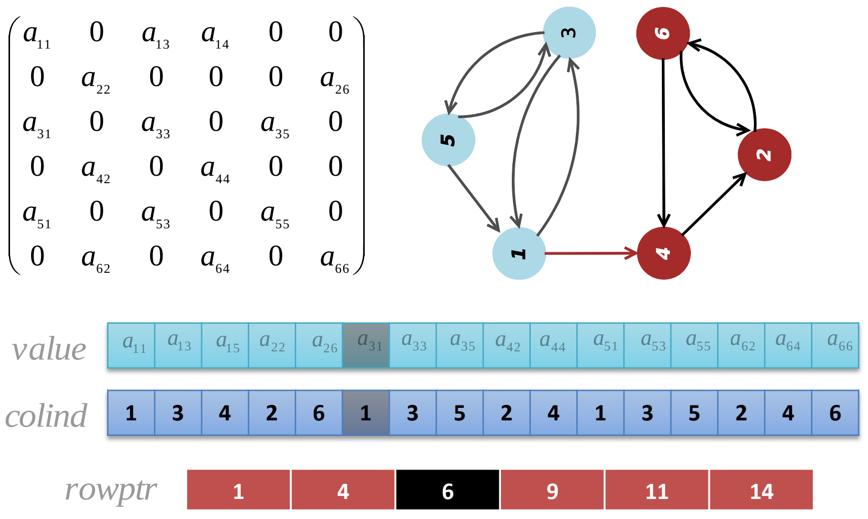
Figure 3.8: Permuted DSM associated with the sample graph

3.4 Sparse Matrix Data Structure

James Wilkinson provided an interesting definition of sparse matrix, he defined a matrix as sparse “if it contains enough zero that it pays to take advantage of them” [18]. The definition highlights that with proper use of data structure and algorithm, sparse matrix enables a significant saving of memory and computational time. Sparse matrix data structure requires less memory by avoiding to store zero elements. On the other hand, sparse matrix algorithm requires less computation time by not performing arithmetic operations on zero elements. In this section, we will discuss an important sparse matrix data structure *Compressed Row Storage* which appeared to be a suitable fit for our research work.

Compressed Row Storage (CRS)

One of the most widely used storage scheme Compressed Row Storage (CRS) stores the matrix in a sequence of compressed rows [51]. Three arrays are used to implement this data structure. The non-zero entries are in row-wise sequence into array $value$. Array $colind$ stores the column index of each non-zero element of the matrix. For example, $colind[6]$ indicates the column index of 6th non-zero element, $value[6]$. Finally, index of the first non-zero entry of each row is stored in array $rowptr$.



$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix}$$

value	a_{11}	a_{13}	a_{15}	a_{22}	a_{26}	a_{31}	a_{33}	a_{35}	a_{42}	a_{44}	a_{51}	a_{53}	a_{55}	a_{62}	a_{64}	a_{66}
colind	1	3	4	2	6	1	3	5	2	4	1	3	5	2	4	6
rowptr	1	4	6	9	11	14										

Figure 3.9: Compressed Row Storage Data Structure

Figure 3.9 displays the data structure to store our example matrix with the help of CRS scheme. The array $value$ store zero elements in a row-wise manner. For each element of $value$, array $colind$ stores its column index. Array $rowptr$ stores the first non-zero element of each row. For example, the first non-zero element of row 3 resides at 6th index of $value$ and column index of $value[4]$ is 1. In order to traverse all the non-zero entries of row i, one need to access $value[rowptr[i]]$ to $value[rowptr[i]-1]$ [29]. Let $nnz(A)$ be the number of non-zero entries in Matrix A. Hence, CSR storage scheme requires $2 \times nnz(A) + n + 1$ unit of computer memory for storing the matrix A.

3.5 The DSM Partitioning Performance

Hossain [29] proposed an efficient way for partitioning the DSM using sparse matrix data structure with the help of Tarjan's algorithm. The DSM computed for majority of application areas contain large proportion of zero entries. Therefore block lower triangularization of a sparse matrix leads to savings in computational work and intermediate storage [52]. For a directed graph $G = (V, E)$, the outgoing edge (v_i, v_j) of each vertex v_i are non-zero entries a_{ij} in matrix A. Consequently, it can be expressed in a CSR representation. Hossain identified the dominating computation step is to access the out-going edge at each vertex [29]. To access the adjacent vertices j of vertex v_i, $j = colind[k]$ where $k = rowptr[i]...rowptr[i+1]-1$. Hence, the number of non-zero entries of matrix A, $nnz(A) = |E|$, and therefore CRS storage provides and efficient implementation of Tarjan's algorithm.

Matrix Name	No. of Vertices (N)	No. of Strong Components	Boost Timing (s)	Our Timing (s)
NotreDame	325729	231666	1.6812	0.318
amazon0601	403394	1588	11.08	2.418
StanfordBerkeley	683446	109238	22.568	3.80

Table 3.2: Timing of boost and our implementation of Tarjan's Algorithm

In our research we used the Tarjan's algorithm using Compresses Row Storage (CRS) data structure for finding strongly connected components. Hence it enables us to partition the dsm for analyzing the design structure of scientific software. The implementation is done in C++ programming language. Later on, the results has been compared with boost implementation of Tarjan's algorithm. Boost is an open-source and most widely used portable C++ library [19]. A set of random large sparse matrices for input were collected from University of Florida Sparse Matrix Collection [18]. In all cases, the timing were better than those obtained from boost implementation, approximately 5 times faster than that of boost one. Table 3.2 displays the timing comparison. The experimental results were generated using a computer with Intel(R) Pentium(R) 4 CPU 3.00GHz processor and 1 GB of RAM.

Chapter 4

Analyzing Design Structure

The interpretation of a real-world complex biological, technological or social phenomena requires the understanding of structure and function of that complex network [10]. Statistical techniques revealed surprising structural properties when applied to the analysis of these networks[10]. These statistical structural properties have major effect on the functionality of the respective network. Newman [44] described a way to measure the statistical properties that seem to be common to these networks. Meyer [42] identified software call graphs as complex network and also found that these software graph reveal properties that are identical to those found in other biological, technological, and social networks. The first objective of this thesis is to quantify these statistical structural properties for scientific software to analyze their design structure. In this chapter we review some basic structural properties of real-world complex network and define metrics which will be later used to compare the designs of a number of open source scientific software. These structural metrics indicates some software quality attributes such as modularity, sensitivity, efficiency, etc of the respective software.

4.1 Characteristic Path Length

In most networks despite of their large size, every pair of vertices seem to be connected by a relatively short path [44]. This fact has been referred as "small world effect" [1]. Social network is a classical example where small world effect is applicable. Whenever someone far from our home surprisingly turns out to be a mutual acquaintance with us, often we state "It's a small world!" [40]. Recent study of information technology networks suggest that these networks, for example internet [2] show small world effect. Small-world networks tend to contain sub-networks which have connections between almost any two nodes within them. In a small world, the average distance between two vertices are called the *characteristic path length*. The characteristic path length between two vertices i and j defined by Braha *et al.*[10] is the number of edges present along the shortest path connecting them. For an undirected graph with N vertices, the characteristic path length l is calculated using,

$$l = \frac{\sum_{i \neq j} d_{ij}}{N(N-1)} \tag{4.1}$$

where d_{ij} is the shortest path length (minimum number of edges) connecting the vertices i and j if there exists a path between the two vertices.

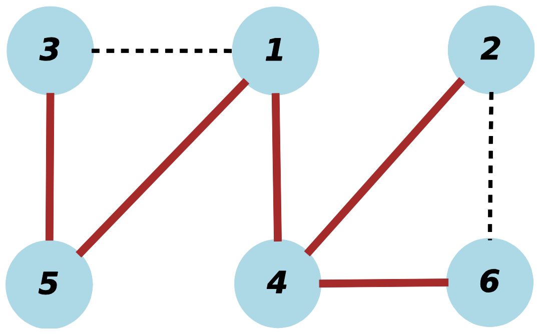


Figure 4.1: A shortest path tree from vertex 5.

Scientific software is also an information carrying network. Characteristic path length indicates the call sequence required from reaching one function to another. The smaller the call sequence, faster information transfer throughout the network. Thus the processing time will be smaller. Small characteristic path length also attributes to efficiency in terms of memory as stack used in registers to store call sequence will be shorter.

The quantity l computed with the help of Dijkstra's algorithm for a graph $G = (V, E)$ with N vertices and $|E|$ edges. Dijkstra's algorithm has been run N times to find the all pair shortest paths. This provides a better running time performance when compared to all pair shortest path algorithm using dynamic programming. The complexity for finding the characteristic path length is therefore $O(|V||E||log|V||)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph G. Whereas, the complexity of all pair shortest path algorithm using dynamic programming is $O(|V|^3)$. The distance from each vertex to itself is zero. When there exists no path between a pair of vertices, the path between such two vertices are excluded from l calculation. Figure 4.1 displays a shortest path tree from source as vertex 5 to all other vertices of the original graph in Figure 2.1(b). The sum of the shortest paths from vertex 5 is 10 $(1+3+1+2+0+3)$. Similarly the sum of the shortest paths from each vertex $v \in V$ to other vertices is calculated. Finally after computing the summation of all shortest path lengths, the sum divided by $N(N-1)$ to get the characteristic path length, l. For the graph in figure 2.1(b), l comes out $54/30 = 1.8$.

4.2 Clustering Coefficient

Another common property of complex network is the inherent tendency of vertices to cluster in modules [10]. In terms of social network it can be represented as a circle of friends where every member know each other [1]. There is a likeliness that a friend of your friend might be your friend [44]. *Clustering coefficient* is a measure of degree to which vertices in a graph tend to cluster together in an undirected graph. Let vertex $i \in V$ in a graph $G = (V,E)$ is connected to k_i neighbours. The maximum number of edges between these k_i vertices can be $k_i(k_i - 1)/2$. If n_i is the number of actual edges between these k_i vertices, then the clustering coefficient of vertex *is* is calculated by:

$$C_i = \frac{2 \times n_i}{k_i(k_i - 1)} \tag{4.2}$$

The network's potential modularity is measured by the clustering coefficient of the graph [10]. User defined functions in same cluster have higher communication and the interaction between separate cluster is minimized [44]. Higher the clustering coefficient, more modular the product is. The modular the product is, it is easy to maintain and modify. In this research, we are interested to find out the potential modularity of the scientific software. The clustering coefficient of the graph G, C is the average of all the vertices:

$$C = \frac{1}{N}\sum_{i=1}^{N} C_i \tag{4.3}$$

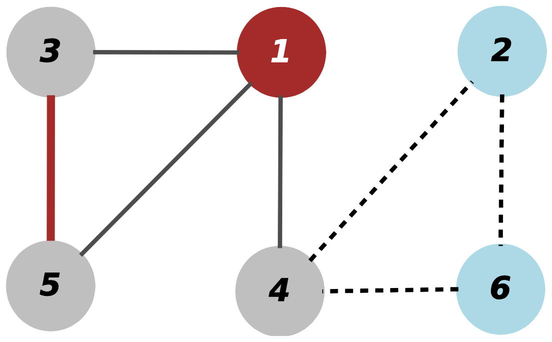


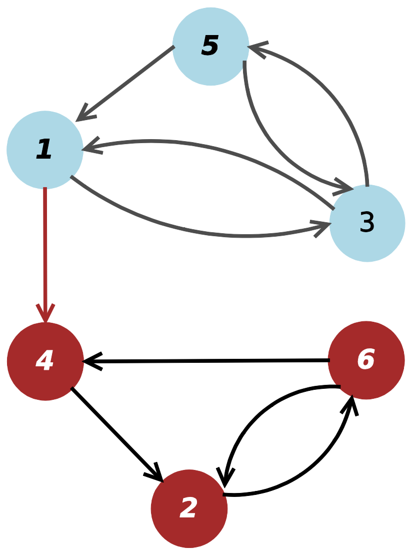
Figure 4.2: Illustration of the clustering coefficient C_1 for vertex 1.

In order to find out C_i for a vertex $i \in V$ of an undirected graph $G = (V,E)$, we first need to identify the vertex list $V' \subset V$ where every vertex $j \in V'$ is adjacent to vertex i. Secondly, an induced subgraph is built which contains the vertex set $\{V' - i\} \in V$ and their associated edges. Figure 4.2 displays an illustration of finding the clustering coefficient for vertex 1. Vertex 1 has 3 neighbouring vertices and there is only one edge between that 3 vertices. Thus the induced graph contains $\{3,4,5\}$ vertices and edge $(3,5)$. Hence, the clustering coefficient of vertex 1, $C_1 = 2 \times 1/6 = \frac{1}{3}$. Similarly the clustering coefficient C_i for all vertices i can the calculated to find the clustering coefficient of the graph, C. Therefore, C comes to be 7 / 9 $\{(\frac{1}{3} + 1 + 1 + \frac{1}{3} + 1 + 1)/6\}$.

4.3 Propagation Cost

A design architect might have been asked ”What would be the consequences to a network connectivity if a vertex was removed?” This type of question has real meaning in complex network. Suppose in software development, programmers tend to group functions of a related nature into classes or namespaces. A change in a single function effects a number of other functions directly or indirectly. MacCormack *et al.*[35] proposed a method to answer such question. They defined a metric *“propagation cost”* which is a measure of proportion of element that will be affected when an specific element is changed in the network. If p_i is the number of vertices reachable from vertex $i \in V$ in a directed graph $G = (V, E)$ using a directed shortest path, then the propagation cost of the directed graph is calculated using

$$P = \frac{\sum_{i=1}^{N} p_i}{N^2} \qquad (4.4)$$



$$\begin{array}{c|c}
 & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}
\end{array}$$

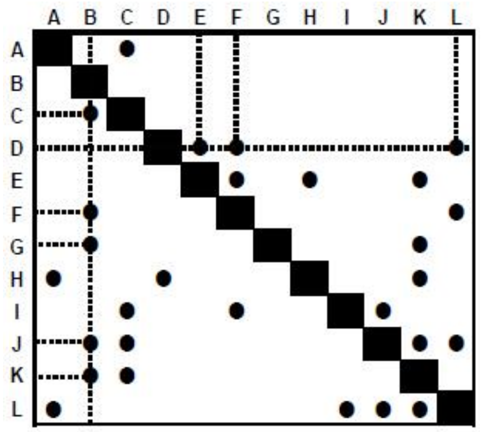
Figure 4.3: Reachability matrix of associated digraph

Figure 4.3 displays a reachability matrix of associated digraph. Each entry a_{ij} of the reachability matrix A is 1 if there exists a directed path from vertex i to j and otherwise 0. Hence the summation of the entries of reachability matrix enables us to compute the propagation cost of the associated digraph. The propagation cost of the digraph in Figure 4.3 is **75%** $([6+3+6+3+6+3]/6^2)$. In other words, this enables us to state that a change in single element will affect 75% of the other element in the network. The reachability matrix can be generated for a graph $G = (V,E)$ in time $O(|V||E|)$ using a simple depth first search where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph G.

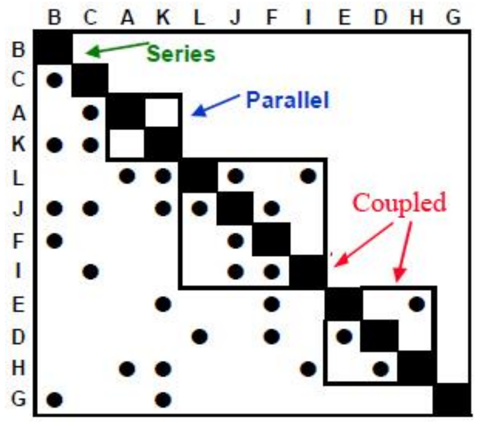
4.4 Partitioned DSM Analysis

DSM partitioning manipulates the DSM in such a order that it reduces or eliminates the feedback marks. In addition, a transparent structure of the network begins to emerge which enables a better planning of project and also efficiently study the architectural elements [70]. Yassine *et al.*[70] identified partitioned DSM displays “an improvement in the flow of design decisions and a faster, less iterative, process”. Partitioned DSM facilitates us to view the sequential task, the tasks which can be done in parallel and most importantly the ones which are involved in iteration or having circular dependencies.

Figure 4.4 displays an partitioned DSM which has been generated by applying partitioning algorithm on the original dsm. Task B and C are sequential because C requires the output of B. On the other hand, A and K can be done in parallel as none of them are mutually dependent. However, the tasks E,D and H are coupled as they are involved in a circular dependency. Sequencing the iterative tasks is difficult to solve and classified as NP-complete problem. NP-complete problems are such problem whose solution can be verified in polynomial time. However, no efficient algorithm to compute the optimal solution in polynomial time in the problem size has been discoverted yet.



(a) Original DSM



(b) Partitioned DSM

Figure 4.4: Partitioned DSM displaying sequential, parallel and iterative task. [70]

4.5 Nodal Degree

One of the important properties to characterize the real world complex network is the distribution of degrees of vertices [10]. In software, nodal degree states how many functions are connected a function in average. Hence, its an indication of style of dependencies between the functions. Lower the nodal degree, lower the dependencies. Lower nodal degree is a good sign of design structure as an impact on a single function affects lower number of functions. Recall from Chapter 2, degree of vertex i denoted by k_i is the number of vertices adjacent to vertex i in an undirected graph with N vertices. *Nodal Degree* is the average degree of the vertices in the undirected graph,

$$<k>=\frac{\sum_{i=1}^{N} k_i}{N} \tag{4.5}$$

For directed graphs , in-degree of vertex i denoted by $k_{in}(i)$ is the number of directed edges pointing to node i and out-degree $k_{out}(i)$ is the number of directed edges pointing away from node i to other nodes. Hence, degree of node i denoted by k_i is the sum of its in-degree $k_{in}(i)$ and out-degree $k_{out}(i)$. The nodal degree of the undirected graph in Figure

2.1(a) is **2.33** $[(3+2+2+4+2+2)/6]$ whereas the nodal degree of the directed graph in Figure 2.1(b) is **3.33** $[\{(2+2)+(2+1)+(2+2)+(2+1)+(1+2)+(1+2)\}/6]$.

4.6 Centrality Measure

The degree distribution analysis provides a local information only. To obtain the global information on how function exert their influence on other functions we need to use an index that measures the centrality of a node. This has been a well-known research area and researchers have introduced a number of *centrality indices*. In social network these indices plays an important role in understanding the role of actors. A measure named *"betweenness"* introduced by Freeman [23] has been the most widely used and simplest centrality index. Newman [45] defined betweenness as a centrality index that is calculated based on the fraction of shortest paths between vertex pairs that pass through the node of interest. Hence in the network on n elements, the betweenness of vertex i is the fraction of shortest paths between two vertices in the network that passes through i. In software engineering, the most central function helps us to identify certain attributes of the software, such as how much object-oriented the system is, what are kernel functions of the software. In our research, we identified the user defined function as most central function which resides in maximum number shortest path in the associated call graph.

With the help of Dijkstra's algorithm we can find out the shortest path $\{v_s, v_1, v_2, ..., v_j, v_t\}$ from vertex v_s to vertex v_t. Let an array *count* tracks the number of shortest path in which each vertex $v \in V$ resides in graph $G = (V,E)$. Whenever a shortest path is found between two vertices v_s and v_t, $count[i]$ for all the intermediate vertices $i \in \{v_1, v_2, ..., v_j\}$ in the shortest path are incremented. Finally when we have explored all the shortest paths, the vertex having the highest count is the most central vertex of our graph G.

Figure 4.5 displays an iteration for the process of finding most central vertex. Tak-

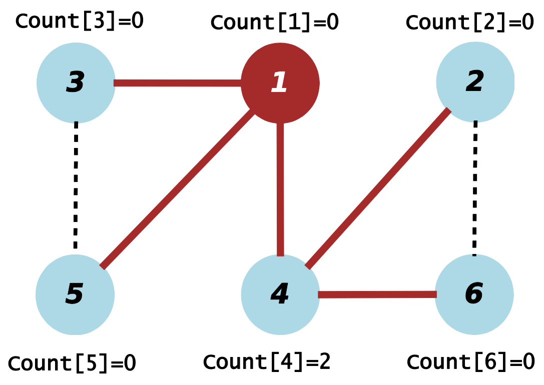


Figure 4.5: Shortest path tree a digraph in Figure 2.1(a) with source as vertex 1 where *count* array tracks how many times each vertex resides in the shortest paths. The vertex with the maximum count is the most central vertex

ing vertex 1 as source and all other vertices as destination, shortest paths from source to destination are computed using Dijkstra's algorithm. Vertex 4 lies in two shortest paths and therefore $count[4]$ is updated with value 2. Similarly, all other shortest paths are computed taking each vertex $i \in V$ as source in graph $G = (V, E)$ and consequently the *count* container is updated. Finally , the vertex having the highest count is our most central vertex. For our sample graph in Figure 4.8 after the computation of all shortest paths, $count = \{6, 0, 0, 12, 0, 0\}$. Hence, vertex 4 is the most frequent vertex and lies in 12 shortest paths.

4.7 Degree Distribution Analysis

Power Law

A quantity x is said to obey power law if

$$p(x) \propto x^{-\alpha} \tag{4.6}$$

where $p(x)$ is the probability distribution of x, α is a constant parameter of the distribution called *exponent* or *scaling*. Although there are some exceptions, the scaling parameter lies between $2 < \alpha < 3$ [15]. Not all values of x obey power law. In practice, x_{min} is the lower bound of power law distribution, where power law applies to x for values greater than x_{min}. The degree of vertices are integer valued and greater than 1. So quantity x is integer and value of x is greater than 1.

Scale Free Networks

The degree distribution of numerous real world complex network have been documented to obey power law distribution [10]. *Scale-free Networks* are networks with power-law degree distribution [44]. Many real word complex network has been identified as scale free networks which includes World Wide Web, biological networks, social networks, etc. Scale free networks are not like random networks and have certain characteristics. One of the important characteristics of scale free network is having some vertices with a degree that greatly exceeds the average. These high connectivity vertices are called hubs. In software, user defined functions with high degree are the hubs. Scale free networks have a fault tolerance behavior. If a function with low degree fails, it will have negligible impact on the system. If a hub fails, there are some other hubs connected to it which will provide service to the other low degree nodes. Finally scale free networks are small world network with smaller characteristic path length. A recent study by Sosa *et al.*[61] indicates the presence of hubs minimizes the number of defects in the system.

Power Law Analysis for Degree Distribution

In this research, we are interested to identify whether power law is obeyed by the degree distribution of the user defined functions. This enables us to find whether the network formed by user defined functions are scale free networks. Various methods exist for analyzing data, but majority of them can produce inaccurate estimate of parameters in power-law distribution [15]. Some of the reliable techniques are based on maximum likelihood. Maximum likehood estimator provides an estimate of model's paramter when a data set is applied on a given model. Clauset *et al.*[15] presented such a method that combined maximum likelihood and goodness-of-fit. Goodness of fit measures the discrepancy between observed values and the values expected under the model [66]. Clauset *et al.*also developed a software for estimating the power law in empirical data. In this thesis, we used their software for identifying the power law behavior for degree distribution of user-defined function.

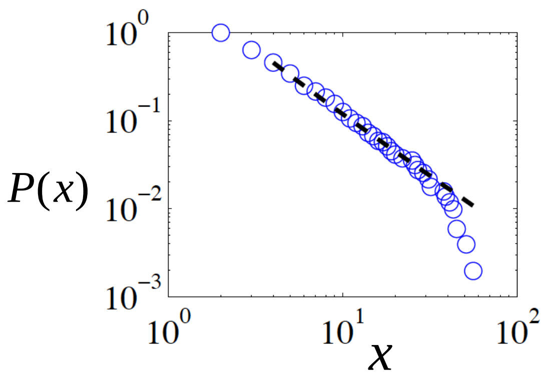


Figure 4.6: Power law distribution of species of mammal. Solid line represent the best fit of the data

Data Set	n	$<x>$	σ	x_{max}	x_{min}	α	p
Species per genus	509	5.59	6.94	56	4 ± 2	2.4	0.1

Table 4.1: Basic parameters along with their power-law fits & corresponding p-value [15]

Figure 4.6 displays the power law distribution for a data set containing the number of species per genus of mammals. The data set is generated by Smith *et al.*[58] which consists of species alive today and also includes some extinct species. Table 4.1 show results from the fitting of the power law form to this data set evaluated using the method described by Clauset *et al.*[15]. The first three columns provide generic statistics of the data which includes mean, standard deviation and maximum value. The last column provides a measure $p-value$ which quantifies whether power law is a plausible fit to the data. If $p \leq 0.05$, power law is ruled out by many researchers [15].

Chapter 5

Iterative Release Analysis

An increasing tendency to develop software in an iterative manner is to achieve better flexibility and higher customer satisfaction [24]. Iterative approaches encompass various ways for production of system components, testing it and finally aggregating user experience feedback for the production of next release [8]. Feature enhancements force large-scale system to change constantly and this results in developing iterative releases of such systems. The design goals of scientific research software systems and the organization in which they are developed are somewhat different from that of commercial or general-purpose software systems [35]. In the chapter, we will review a technique which is useful to analyze the iterative releases of scientific software.

Brown *et al.* [12] described an approach for calculating the total release cost with the help of propagation cost. Each incremental release is characterized by two attributes: 1) new architectural elements implemented and 2) number of dependencies between the new architectural elements and previously implemented architectural elements [12]. They calculated the total cost of release n, Tc_n as the combination of the cost to implement the architectural element selected to be added in this release, Ic_n plus the cost of rework of pre-existing element, Rc_n.

$$Tc_n = Ic_n + Rc_n \tag{5.1}$$

In this thesis, we assumed implementation cost of each architectural element to be 1. In our research, user defined functions are the architectural elements. Ic_n is the summation of all new elements of release n. Let m be the number of new architectural elements to be implemented in the new release. Whenever any new elements are added to the system, there might be modification in one or more pre-existing elements in order to accommodate the new ones [12]. Suppose that for a new release element j, $I[j]$ old release elements have to be modified. P_{n-1} is the propagation cost of previous release (release $n-1$). As the implementation cost for each new element is considered 1, the rework cost of release n is defined by

$$Rc_n = \sum_{j=1}^{m} I[j] * P_{n-1} \tag{5.2}$$

where m is the number of new release element. Our developed toolkit computes the total release cost of the incremental releases. Given two releases as input in tulip [4] format or vcg(visualization of compiler graph) [55] format, it computes the structural metrics of both releases. Later it segregates the new user-defined functions and extracts old release dependencies. Finally, the toolkit computes the total release cost for the later version.

We introduce a metric in this thesis that quantify the percentage change (PC) in structural properties between release $n-1$ and release n. For release n, suppose $NUDF$ is the number of new user-defined function introduced and $DUDF$ is the number of user-defined function dropped from release $n-1$. Then PC_n is calculated as

$$PC_n = \frac{(NUDF + DUDF) \times 100}{N_{n-1}} \tag{5.3}$$

where N_{i-1} is the total number of user defined functions in release $n-1$. Section 3.1 displays the source code and call graph of small C/C++ program which computes area of a circle. Suppose the next version V_2 of the software computes the area of circle and as well as square. Provided below is the modified source code for the new version:

```
void print(double area)

        {

        cout << "The area is :"<< area << endl;
        }

double compute_area(int type)
        {
        double k,area;
        if(type == 1)
            {
            k=get_radius();
            // calculating area for circle
            area= 3.142 * k * k;
            }
        else if(type == 2)
            {
            k=get_length();
            // calculating area for square
            area= k * k;
            }
        return area;
        }

double get_radius()
        {
        double radius;
        cout << "Please provide the radius of the circle : " << endl;
        cin >> radius;
        return radius;
        }
```

```
double get_length()
        {
        double length;
        cout << "Please provide the length of the square : " << endl;
        cin >> length;
        return length;
        }
void main()
        {
        int type;
        cout << "Input 1 for circle , 2 for square: ";
        cin >> type;
        double area=compute_area(type);
        print(area);
        }
```

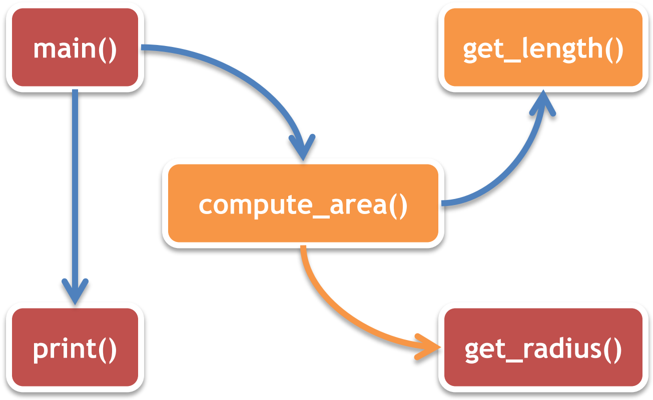


Figure 5.1: Static call graph associated with new version V_2 of C/C++ program to calculate area of a circle and square

Figure 5.1 displays a static call graph of the new version. Two new functions get_length() and compute_area() has been introduced while get_area() from previous version has been dropped. Release V_1 contains 4 user-defined functions. Hence, *PC* of new release V_2 in terms of structural elements is **75%** $\{[(2+1)*100]/4\}$. Table 5.2 displays the percentage change (PC) between the two releases in terms of structural elements and links.

Release	Files		User Defined Functions		
	Nodes	Edges	Nodes	Edges	*PC*
V_1	1	1	4	3	_
V_2	1	1	5	4	75

Table 5.1: Structural Properties

The propagation cost of old release V_1 in Figure 2.7 is **50%** $([4+2+1+1]/4^2)$ (see Section 4.3). Function compute_area() of new version V_2 depends on function get_radius() of previous V_1. So the release rework cost is $0.5(1 \times 0.5)$. Hence, the total new release cost is 2.5. Table 5.2 displays the total release cost for the two versions.

Old Version	New Version	New elements	P_{n-1}	Ic_n	Rc_n	Tc_n
V_1	V_2	2	0.5	2	0.5	2.5

Table 5.2: Release Cost Comparison

Chapter 6
Experiments

6.1 Initial Database of Scientific Software

Computational Infrastructure for Operations Research (COIN-OR) is one of the largest and most widely studied open source communities for scientific research software. We studied open source software projects from COIN-OR. Initially, we identified 10 scientific computing C/C++ projects. They provided our initial database. After an effective examination, we selected those for which we could obtain data for successive major releases. This filter left us with a set of eight coin-or projects for our experiments. Out of these eight, four projects align with our scientific research software category discussed above.

First, we downloaded the original source codes for each releases using their version control tool SVN repositories. Later compiled using the static call graph extractor to a get a tulip file (*.tlp) for each release. Finally, we used the tulip file as input to our developed toolkit STSCB.

6.2 Testing Enviroment

The details of the experimentation environment are as follows: Machine: HP P6510F, Processor: AMD Athlon II X4 630 quad core precessors, Operating System: Ubuntu Release 10.04 and Physical Memory: 8GB.

6.3 Results

We measured the structural properties, structural metrics and the cost of alternative release using dependency analysis with the help of propagation cost. As stated earlier, we tested four scientific research software and present the preliminary results below. DSM analysis have been conducted on several releases but we present results of a single version. The results of the other versions were found in this section similar in nature.

6.3.1 ADOL-C

ADOL-C is an open-source package for the automatic differentiation of C and C++ programs [25]. The first and higher derivatives of vector functions are evaluated using operator overloading method by ADOL-C [67]. It is developed by a team of researchers from Argonne National Lab, Dresden University of Technology, and Humboldt University over a period of 20+ years.

Release	Files		User Defined Functions		
	Nodes	Edges	Nodes	Edges	Percentage Change, *PC* (see Chapter 5)
V 1.9	60	78	315	1033	_
V 1.10.0	60	77	320	1037	2.2222
V 1.10.1	60	77	320	1037	0
V 1.10.2	60	77	220	1037	0
V 2.1.0	60	66	271	703	87.813
V 2.1.1	60	66	271	703	0
V 2.1.2	60	66	271	703	0
V 2.1.4	60	66	272	699	0.369
V 2.1.12	61	67	263	692	4.0441
V 2.2.1	62	68	265	691	4.5627

Table 6.1: Structural Properties of ADOL-C Versions

ADOL-C Versions	Characteristic Path length, l	Clustering coefficient, C	Nodal Degree	Components	Propagation Cost (%)
V 1.9	3.36142	0.107767	6.55873	315	3.53238
V 1.10.0	3.25725	0.106083	6.48125	320	3.43262
V 1.10.1	3.25725	0.106083	6.48125	320	3.43262
V 1.10.2	3.25725	0.106083	6.48125	320	3.43262
V 2.1.0	2.05005	0.080382	5.18819	271	3.41635
V 2.1.1	2.04977	0.0803177	5.19557	271	3.42316
V 2.1.2	2.04977	0.0803177	5.19557	271	3.42316
V 2.1.4	2.0611	0.0777237	5.13971	272	3.38452
V 2.1.12	2.20408	0.0803834	5.26236	263	3.58831
V 2.1.12	2.16312	0.0799071	5.21509	295	3.50303

Table 6.2: Structural Metrics of ADOL-C Versions

Table 6.1 presents the structural properties and Table 6.2 provides us the structural matrices of 10 different releases of ADOL-C. Out of these 10 release, 4 releases are identical with their previous releases in terms of structural properties and metrics. A major change is observed between release V1.10.2 and V2.1.0. V2.1.0 contains 155 functions from V1.10.2 and 116 new functions has been introduced. Table 6.4 represents the summary of the cost of each release. Table 6.3 provides the information of most central function (most frequently called function) of the version. The major change has also changed the most central function of the new release. Function “putof” is the central function (resides in 32523 shortest path) in V 1.10.2 whereas “fail” is the central function (resides in 16398 shortest paths) in V 2.1.0.

No significant change in propagation cost of release indicates the stability of the software. The decreasing trend of nodal degree is a good sign of design as it decreases the dependencies of each function. Decreasing trend of characteristic path length indicates shorter call sequence. So later releases are faster and takes less memory. Decreasing trend of clustering indicates less modular design of later release.

Version	User-defined functions		Version	User-defined functions	
	Function Name	Number of shortest path		Function Name	Number shortest path
V 1.9	putof	31958	V 2.1.1	fail	16396
V 1.10.0	putof	32523	V 2.1.2	fail	16396
V 1.10.1	putof	32523	V 2.1.4	fail	16561
V 1.10.2	putof	32523	V 2.1.12	fail	16561
V 2.10	fail	16398	V 2.2.1	fail	16561

Table 6.3: Centrality Measure of ADOL-C Version

Old Version	New Version	New elements	P_{n-1}	Ic_n	Rc_n	Tc_n
V1.9	V1.10.0	6	0.0353238	6	0.52986	6.52986
V1.10.0	V1.10.1	0	0.0343262	0	0	0
V1.10.1	V1.10.2	0	0.0343262	0	0	0
V1.10.2	V2.1.0	116	0.0343262	116	13.696	129.696
V2.1.0	V2.1.1	0	0.0341635	0	0	0
V2.1.1	V2.1.2	0	0.0342316	6	0	0
V2.1.2	V2.1.4	1	0.0342316	1	0.0342316	1.0342316
V2.1.4	V2.1.12	1	0.0338452	1	0.0338452	1.0338452
V2.1.12	V2.2.1	7	0.0353238	7	0.28707	7.28707

Table 6.4: Release Cost Comparison of ADOL-C Versions

Data Set	n	$<x>$	σ	x_{max}	x_{min}	α	p
ADOL-C In degree	271	1	8.718	131	1	1.6	0.267
ADOL-C Out degree	271	1	7.277	91	2	1.56	0.286

Table 6.5: Basic parameters along with their power-law fits - ADOL-C

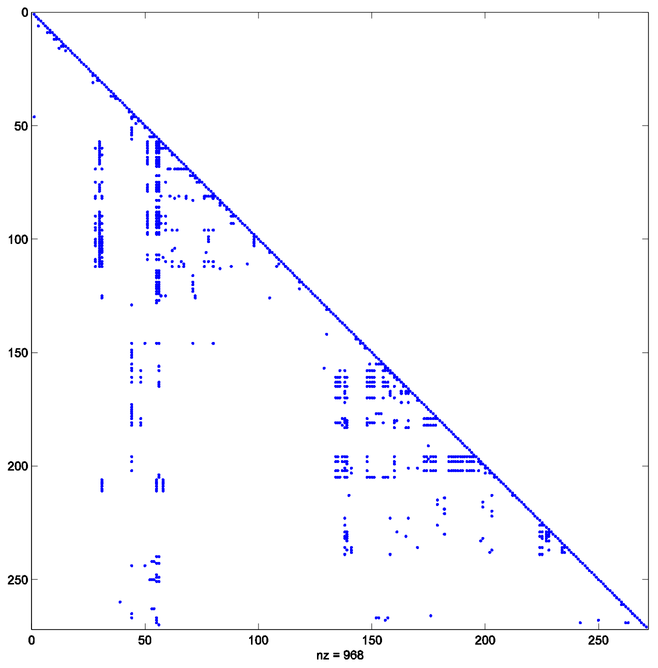


Figure 6.1: Partitioned ADOL-C

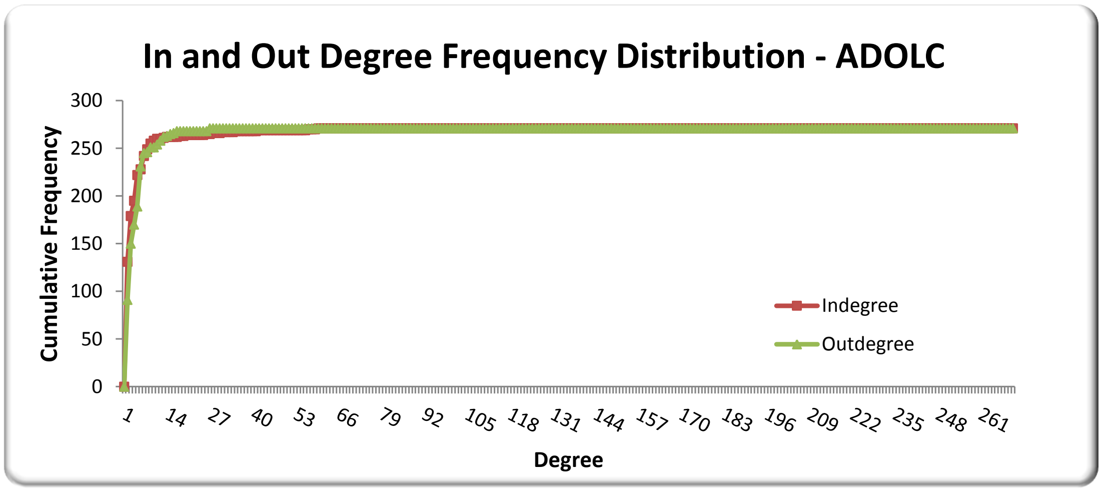


Figure 6.2: In-degree and Out-degree distribution of ADOL-C

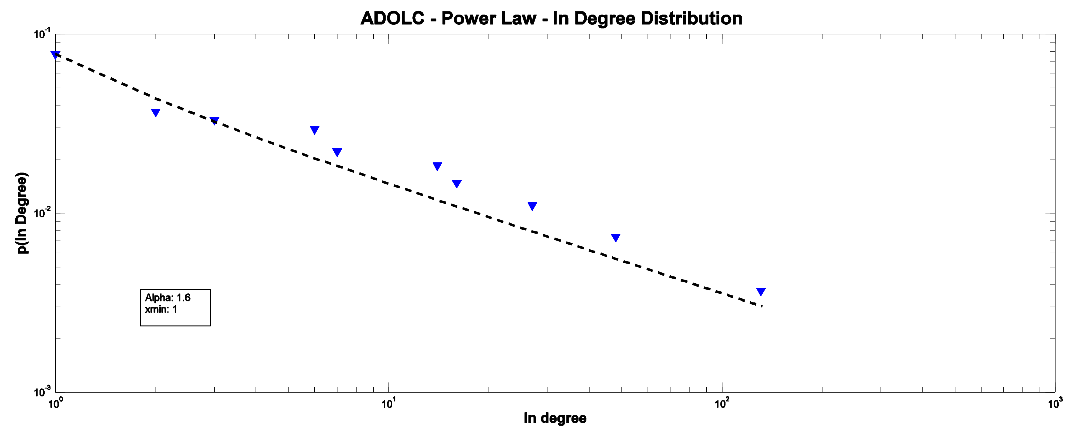


Figure 6.3: Power Law Distribution for In Degree of ADOL-C

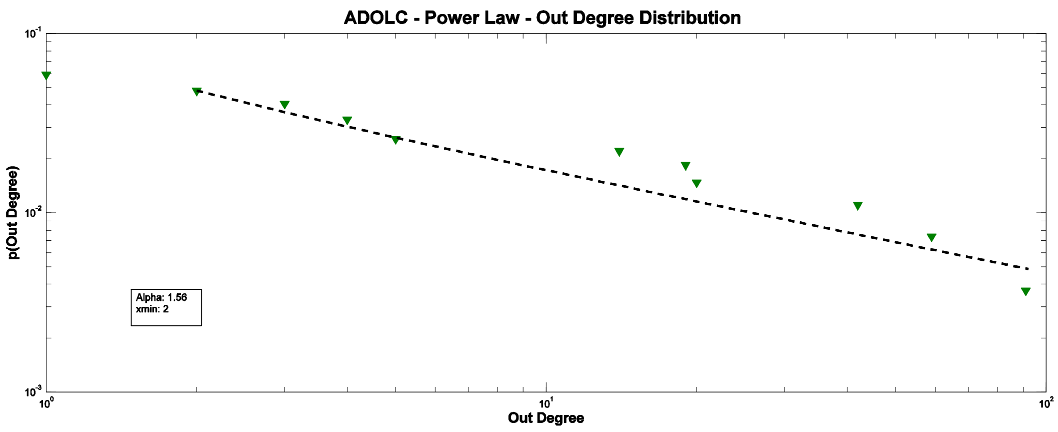


Figure 6.4: Power Law Distribution for Out Degree of ADOL-C

The DSM analysis in Figure 6.1 indicates the absence of any feedback mark or no cycle present in dependency for user-defined functions. Figures 6.2 present cumulative (in, out) degree distributions of call graph nodes. We note that the total nodal degree varies from 1 to 62 for ADOL-C with approximately 80% of the nodes having degree less than or equal to 8. In other words, only a small fraction of the functions are most relevant with regard to the functioning of the software. In table 6.5 we show results from the fitting of a power-law form to each of these data. The degree distribution found from toolkit ADOL-C has been conjectured to follow power law. As we see, the results indicate that all of the datasets are indeed consistent with a power-law hypothesis as the p value is greater than 0.05. The variable p is the probability whether a distribution obeys power law or not. The figures 6.3 and 6.4 show these data graphically, along with the estimated power-law distribution. The solid line represents the best fit to the data. The power law confirmity indicates the network formed by the user defined functions is a scale free network.

6.3.2 Branch-Cut-Price (BCP)

A parallel framework , BCP is used for implementing branch, cut, and price algorithms which are required to solve mixed integer programs (MIPs)[30]. BCP was developed by IBM, and later contributed to the COIN-OR library [46]. It has been used successfully in a number of IBM consulting engagements.

Table 6.6 presents the structural properties and Table 6.7 provides us the structural matrices of 7 different releases of BCP. Table 6.9 represents the summary of the cost of each release. Interestingly, all of the versions are identical to the first version in terms of structural properties and metrics. Hence there seems to be virtually no effort for the incremental releases. Table 6.8 provides the information of most central function (most frequent called

Release	Files		User Defined Functions		
	Nodes	Edges	Nodes	Edges	Percentage Change, *PC* (see Chapter 5)
V 1.2.0	45	29	60	118	_
V 1.2.1	45	29	60	118	0
V 1.2.2	45	29	60	118	0
V 1.2.3	45	29	60	118	0
V 1.3.0	45	29	60	118	0
V 1.3.1	45	29	60	118	0
V 1.3.2	45	29	60	118	0

Table 6.6: Structural Properties of BCP Versions

BCP Versions	Characteristic Path length, *l*	Clustering coefficient, *C*	Nodal Degree	Components	Propagation Cost (%)
V 1.2.0	0.264972	0	3.93333	60	4.94444
V 1.2.1	0.264972	0	3.93333	60	4.94444
V 1.2.2	0.264972	0	3.93333	60	4.94444
V 1.2.3	0.264972	0	3.93333	60	4.94444
V 1.2.0	0.264972	0	3.93333	60	4.94444
V 1.3.1	0.264972	0	3.93333	60	4.94444
V 1.3.2	0.264972	0	3.93333	60	4.94444

Table 6.7: Structural Metrics of BCP Versions

function) of the version. Consequently, there is no change in the central function. Clustering coefficient of zero implies that BCP has tree like design structure where vertices are the user-defined functions.

Version	User-defined functions		Version	User-defined functions	
	Function Name	Number of shortest path		Function Name	Number of shortest path
V 1.2.0	”size”	25	v 1.3.0	size	25
V 1.2.1	”size”	25	v 1.3.1	size	25
V 1.2.2	”size”	25	v 1.3.2	size	25
V 1.2.3	”size”	25			

Table 6.8: Centrality Measure of BCP Version

Old Version	New Version	New elements	P_{n-1}	Ic_n	Rc_n	Tc_n
V1.2.0	V1.2.1	0	00.0494444	0	0	0
V1.2.1	V1.2.2	0	00.0494444	0	0	0
V1.2.2	V1.2.3	0	00.0494444	0	0	0
V1.2.3	V1.3.0	0	00.0494444	0	0	0
V1.3.0	V1.3.1	0	00.0494444	0	0	0
V1.3.1	V1.3.2	0	00.0494444	0	0	0

Table 6.9: Release Cost Comparison of BCP Versions

In Figures 6.5, the DSM for BCP is displayed after the partitioning algorithm is applied to the user function call graph. Absence of feedback mark is also visible indicating no cycle present.

Data Set	n	$<x>$	σ	x_{max}	x_{min}	α	p
BCP In degree	60	1	4.988	37	3	1.9	0.54
BCP Out degree	60	1	4.679	26	10	2.5	0.26

Table 6.10: Basic parameters along with their power-law fits - BCP

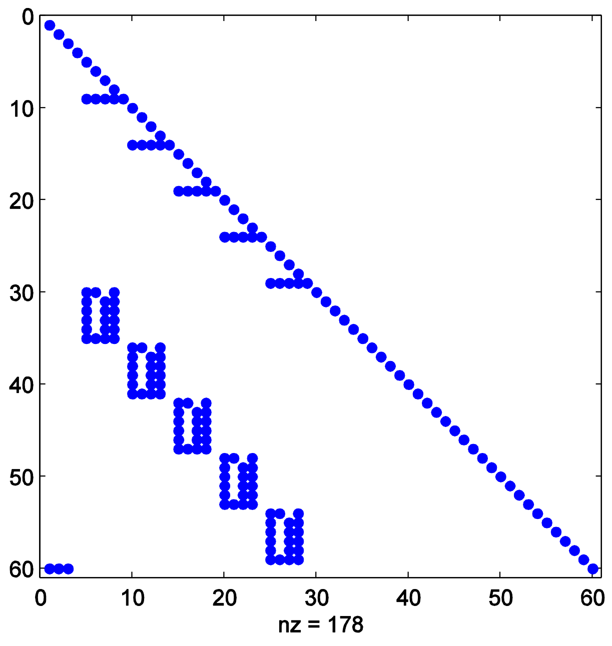


Figure 6.5: Partitioned DSM - Bcp

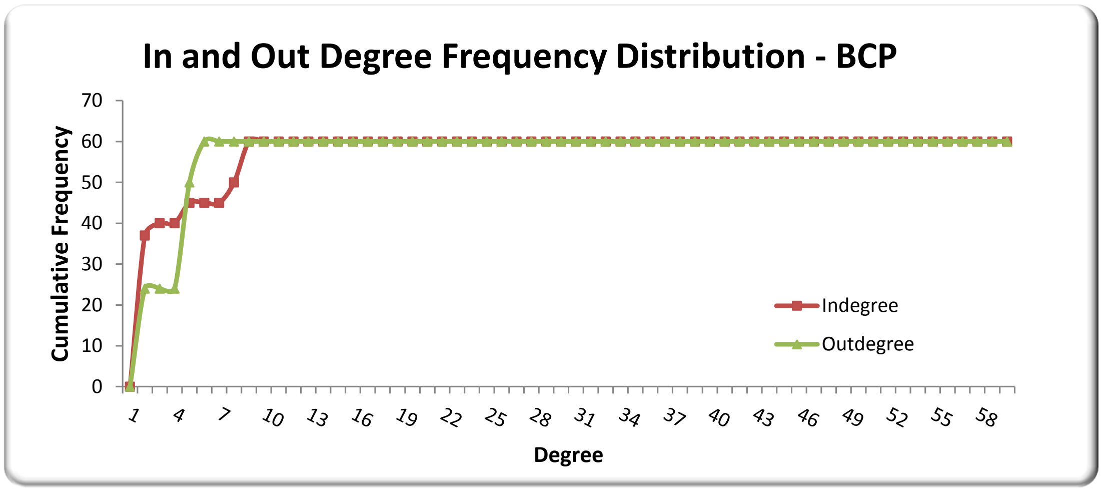


Figure 6.6: In-degree and Out-degree distribution of BCP

Figures 6.6 present cumulative (in, out) degree distributions of call graph nodes. We note that the total nodal degree varies from 1 to 8 for BCP with approximately 80% of the nodes having degree less than or equal to 4. In table 6.10 we show results from the fitting of a power-law form to each of these data sets along with a variety of generic statistics. Figure 6.7 and 6.8 contains the power law graphs for in-degree and out-degree of BCP. The results indicate that all of the datasets are indeed consistent with a power-law hypothesis as the *p* value is greater than 0.05. The power law confirmity indicates the network formed by the user defined functions is a scale free network.

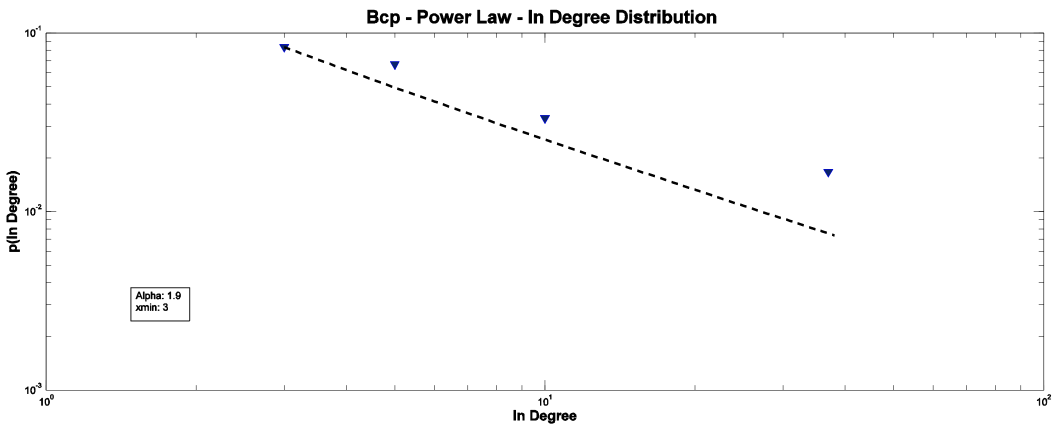


Figure 6.7: Power Law Distribution for In Degree of BCP

6.3.3 CppAD

CppAD is another Automatic differentiation software that generates an algorithm that computes corresponding derivative values [7]. Similar to other AD software, it computes first and higher derivatives using either forward or reverse mode. It is developed as a one-person effort at the University of Washington, Seattle.

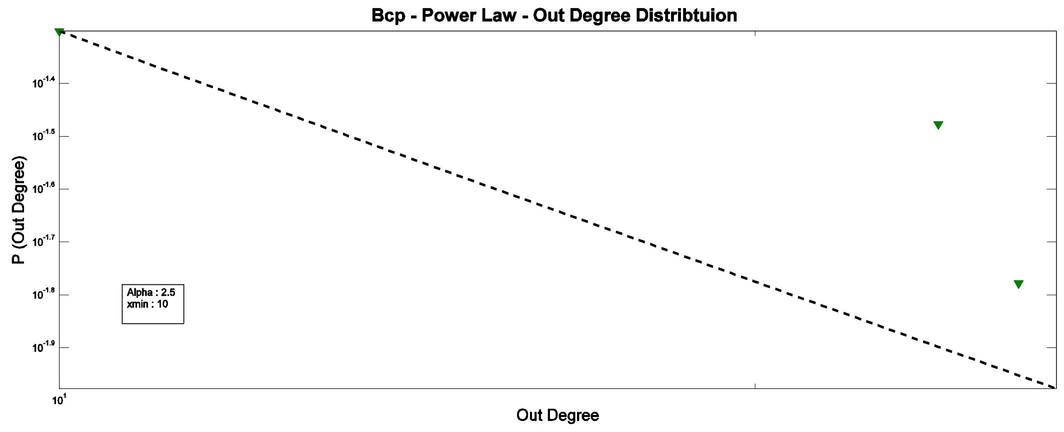


Figure 6.8: Power Law Distribution for Out Degree of BCP

Release	Files		User Defined Functions		
	Nodes	Edges	Nodes	Edges	Percentage Change, *PC* (see Chapter 5)
V 110101.0	65	65	76	164	_
V 110101.1	65	65	76	164	0
V 110101.2	65	65	76	164	0
V 110101.3	65	65	76	164	0
V 110101.4	65	65	76	164	0
V 110101.5	65	65	76	164	0
V 110308	66	67	80	175	5.2632
V 111103	68	67	103	196	53.75
V 111104	68	67	103	196	0
V 111105	68	67	103	196	0

Table 6.11: Structural Properties of CppAD Versions

Table 6.11 presents the structural properties and Table 6.12 provides us the structural matrices of 10 different releases of CppAD. Out of these 10 releases, 7 releases are identical to previous release in terms of structural properties and metrics. In a major change between release V110308 and V111103, V111103 contains 70 functions from V110308 and 33 new functions has been introduced. Table 6.14 represents the summary of the cost of each release. Table 6.13 provides the information of most central function (most frequent called function) of the version. This major change increased maximum shortest path count of the most central function, however "constructor-special" remained as the central function.

CppAD Versions	Characteristic Path length, l	Clustering coefficient, C	Nodal Degree	Number of Components	Propagation Cost (%)
V 110101.0	2.35228	0.0363174	4.31579	76	6.95983
V 110101.1	2.35228	0.0363174	4.31579	76	6.95983
V 110101.2	2.35228	0.0363174	4.31579	76	6.95983
V 110101.3	2.35228	0.0363174	4.31579	76	6.95983
V 110101.4	2.35228	0.0363174	4.31579	76	6.95983
V 110101.5	2.35228	0.0363174	4.31579	76	6.95983
V 110308	2.37373	0.0342364	4.375	80	6.64062
V 111103	2.44108	0.0265913	3.80583	103	9.6239
V 111104	2.44108	0.0265913	3.80583	103	9.6239
V 111105	2.44108	0.0265913	3.80583	103	9.6239

Table 6.12: Structural Metrics of CppAD Versions

A significant change in propagation cost observed between V110308 and V111103 which indicates the increase of sensitivity in the new release. The decreasing trend of nodal degree is a good sign of design as it decreases the dependencies of each function. The increasing trend of characteristic path length indicates longer call sequence. So earlier releases are faster and takes less memory. Decreasing trend of clustering indicates less modular design of later release.

Version	User-defined functions		Version	User-defined functions	
	Function Name	Number of shortest path		Function Name	Number of shortest path
V110101.0	”constructor-special”	1884	V110101.5	”constructor-special”	1884
V110101.1	”constructor-special”	1884	V110308	”constructor-special”	2296
V110101.2	”constructor-special”	1884	V111103	”constructor-special”	2788
V110101.3	”constructor-special”	1884	V111104	”constructor-special”	2788
V110101.4	”constructor-special”	1884	V111105	”constructor-special”	2788

Table 6.13: Centrality Measure of CppAD Version

Old Version	New Version	New elements	P_{n-1}	Ic_n	Rc_n	Tc_n
V110101.0	V110101.1	0	0.0695983	0	0	0
V110101.1	V110101.2	0	0.0695983	0	0	0
V110101.2	V110101.3	0	0.0695983	0	0	0
V110101.3	V110101.4	0	0.0695983	0	0	0
V110101.4	V110101.5	0	0.0695983	0	0	0
V110101.5	V110308	4	0.0695983	4	0.34799	4.34799
V110308	V111103	33	0.0664062	33	3.5859	36.58599
V111103	V111104	0	0.096239	0	0	0
V111104	V111105	0	0.096239	0	0	0

Table 6.14: Release Cost Comparison of CppAD Versions

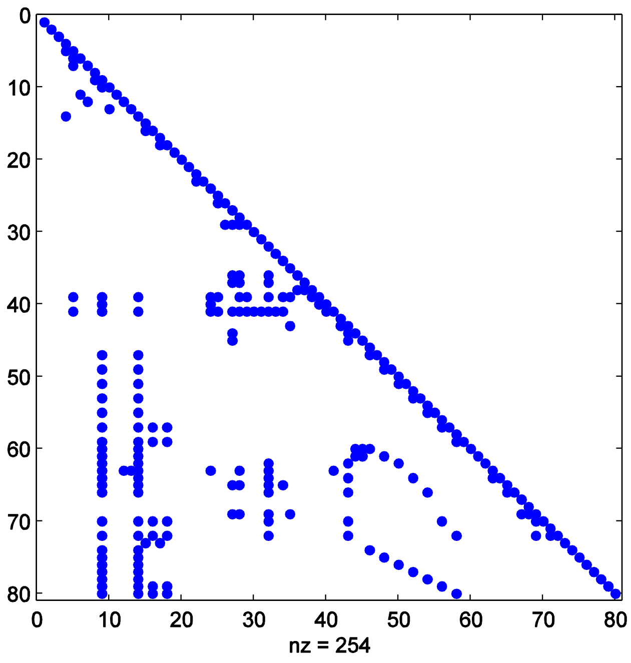


Figure 6.9: Partitioned DSM - CppAD

In Figures 6.9, the DSM for CppAD is displayed after the partitioning algorithm is applied to the user function call graph. Absence of feedback mark is also visible indicating no cycle present.

Data Set	n	$<x>$	σ	x_{max}	x_{min}	α	p
CppAD In degree	81	1	4.3341	30	1	1.6	0.388
CppAD Out degree	81	1	4.2097	31	1	1.57	0.493

Table 6.15: Basic parameters along with their power-law fits - CppAD

Figures 6.10 present cumulative (in, out) degree distributions of call graph nodes. We note that the total nodal degree varies from 1 to 29 for CppAD with approximately 80% of the nodes having degree less than or equal to 6. In table 6.15 we show results from the fitting of a power-law form to each of these data sets along with a variety of generic statistics. Figure 6.11 and 6.12 contains the power law graphs for in-degree and out-degree of CppAD. The

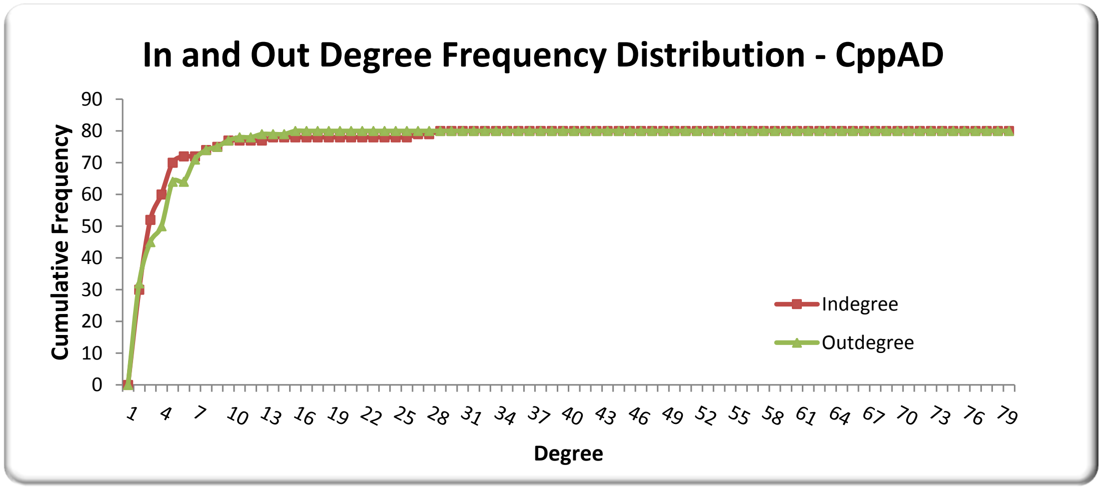


Figure 6.10: In-degree and Out-degree distribution of CppAD

results indicate that all of the datasets are indeed consistent with a power-law hypothesis as the p value is greater than 0.05. The power law confirmity indicates the network formed by the user defined functions is a scale free network.

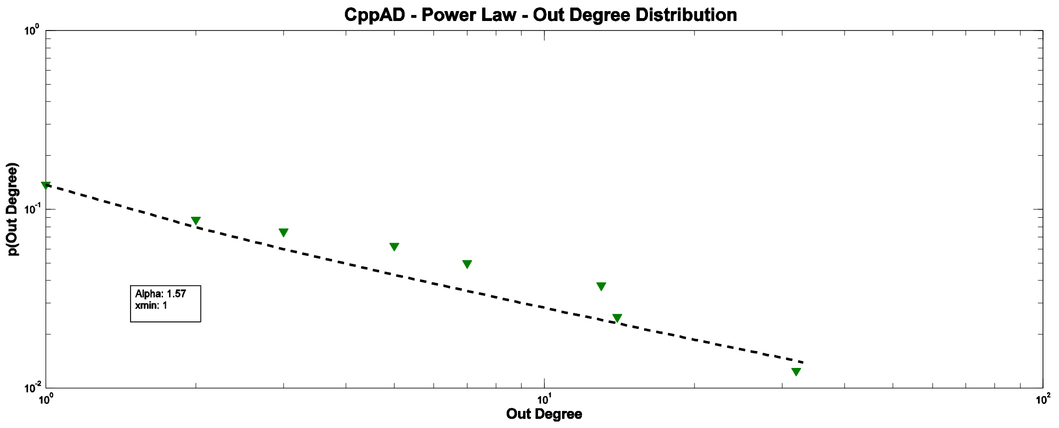


Figure 6.11: Power Law Distribution for In Degree of CppAD

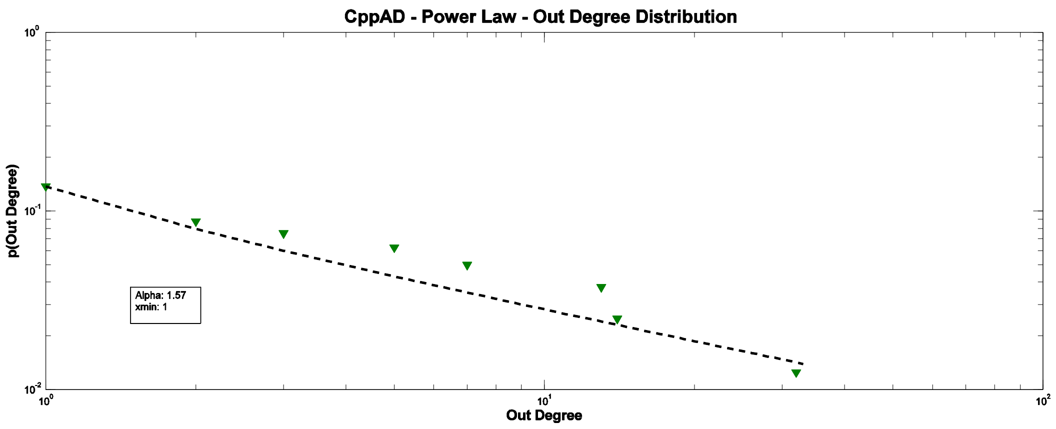


Figure 6.12: Power Law Distribution for Out Degree of CppAD

6.3.4 *A DYnamic Linear Programming code (DyLP)*

Padberg [47] described a dynamic simplex algorithm for solving LP problems. Dynamic simplex attempts to maintain a reduced active constraint system. DyLP implements the dynamic simplex algorithm. It alternates between primal and dual simplex phases [26]. It is developed by a research team from Computing Science department of Simon Fraser University, BC, Canada.

Release	Files		User Defined Functions		
	Nodes	Edges	Nodes	Edges	Percentage Change, *PC* (see Chapter 5)
V 1.3.0	48	304	299	1224	_
V 1.4.0	48	304	299	1229	43.478
V 1.4.4	48	304	299	1229	0
V 1.5.0	51	333	315	1321	8.6957
V 1.6.0	51	333	315	1321	0
V 1.7.0	51	334	320	1327	2.2222
V 1.7.2	51	334	320	1327	0
V 1.8.0	51	334	320	1327	0
V 1.8.1	51	334	320	1327	0
V 1.8.2	51	334	320	1327	0

Table 6.16: Structural Properties of DyLP Versions

DyLP Versions	Characteristic Path length, l	Clustering coefficient, C	Nodal Degree	Number of Components	Propagation Cost (%)
V 1.3.0	2.67341	0.261488	8.18729	299	5.51112
V 1.4.0	2.6719	0.258967	8.22074	299	5.51784
V 1.4.4	2.6719	0.258967	8.22074	299	5.51784
V 1.5.0	2.67967	0.245807	8.3873	315	5.17208
V 1.6.0	2.67967	0.245807	8.3873	315	5.17208
V 1.7.0	2.63341	0.24186	8.29375	320	5.04883
V 1.7.2	2.63341	0.24186	8.29375	320	5.04883
V 1.8.0	2.63341	0.24186	8.29375	320	5.04883
V 1.8.1	2.63341	0.24186	8.29375	320	5.04883
V 1.8.2	2.63341	0.24186	8.29375	320	5.04883

Table 6.17: Structural Metrics of DyLP Versions

Table 6.16 presents the structural properties and Table 6.17 provides us the structural matrices of 10 different releases of DyLP. Out of these 10, 6 releases were identical to previous release in terms of structural properties and metrics. A major change is observed between first two releases though the number of user-defined functions are same. 65 functions has been introduced and dropped in the later release. Table 6.19 represents the summary of the cost of each release. Table 6.18 provides the information of most central function (most frequent called function) of the version. However, these major changes increased the shortest path count of the most central function but no change in observed for most central function.

Version	User-defined functions		Version	User-defined functions	
	Function Name	Number of shortest path		Function Name	Number of shortest path
V1.3.0	errmsg	32301	V1.7.0	errmsg	36817
V1.4.0	errmsg	32918	V1.7.2	errmsg	36817
V1.4.4	errmsg	32918	V1.8.0	errmsg	36817
V1.5.0	errmsg	36372	V1.8.2	errmsg	36817
V1.6.0	errmsg	36372	V1.8.2	errmsg	36817

Table 6.18: Centrality Measure of DyLP Version

No significant change in propagation cost of release indicates the stability of the software. The increasing trend of nodal degree is not a good sign of design as it increases the dependencies of each function. Decreasing trend of characteristic path length indicates shorter call sequence. So later releases are faster and takes less memory. Decreasing trend of clustering indicates less modular design of later release.

Old Version	New Version	New elements	P_{n-1}	Ic_n	Rc_n	Tc_n
V1.3.0	V1.4.0	65	0.0551112	65	12.896	77.896
V1.4.0	V1.4.4	0	0.0551784	0	0	0
V1.4.4	V1.5.0	21	0.0551784	21	4.2487	25.2487
V1.5.0	V1.6.0	0	0.0517208	0	0	0
V1.6.0	V1.7.0	6	0.0517208	6	1.08614	7.08614
V1.7.0	V1.7.2	0	0.0504883	0	0	0
V1.7.2	V1.8.1	0	0.0504883	0	0	0
V1.8.0	V1.8.2	0	0.0504883	0	0	0
V1.8.2	V1.8.3	0	0.0504883	0	0	0

Table 6.19: Release Cost Comparison of DyLP Versions

In Figures 6.13, the DSM for DyLP is displayed after the partitioning algorithm is applied to the user function call graph. Absence of feedback mark is also visible indicating no cycle present.

Data Set	n	$<x>$	σ	x_{max}	x_{min}	α	p
DyLP In degree	299	1	8.893	142	1	1.62	0.1
DyLP Out degree	299	1	5.9412	67	17	2.9	0.12

Table 6.20: Basic parameters along with their power-law fits - DyLP

Figures 6.14 present cumulative (in, out) degree distributions of call graph nodes. We note that the total nodal degree varies from 1 to 153 for DyLP with approximately 80% of the nodes having degree less than or equal to 8. In table 6.20 we show results from the fitting of a power-law form to each of these data sets along with a variety of generic statistics. Figure

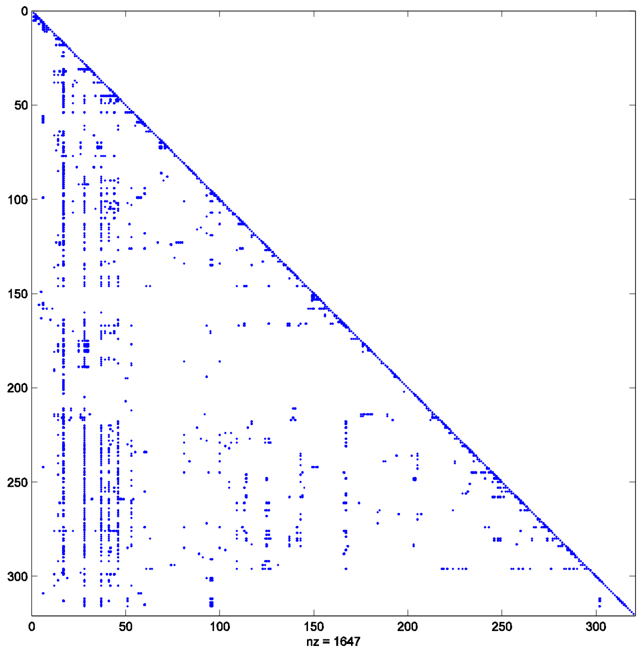


Figure 6.13: Partitioned DSM - DyLP

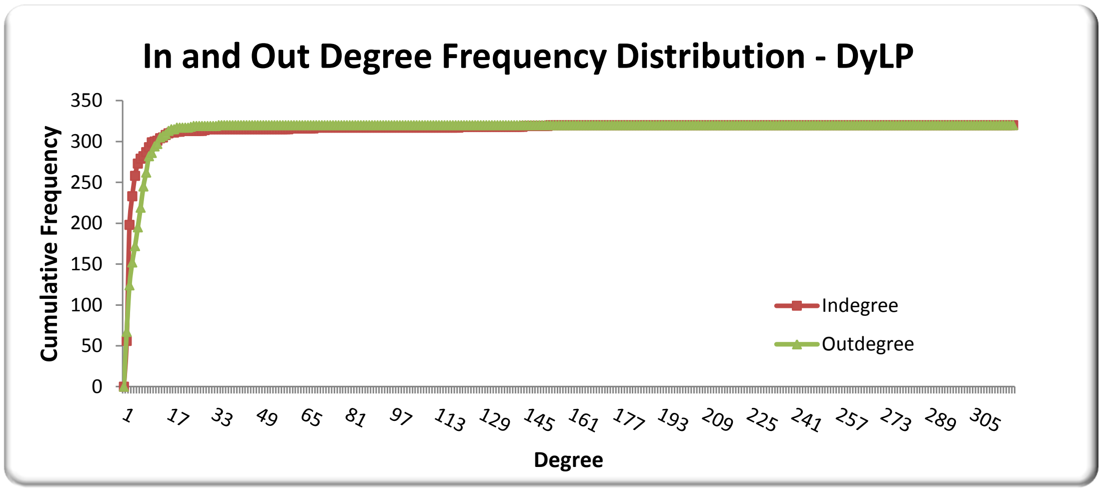


Figure 6.14: In-degree and Out-degree distribution of DyLP

6.15 and 6.16 contains the power law graphs for in-degree and out-degree of DyLP. The results indicate that all of the datasets are indeed consistent with a power-law hypothesis as the *p* value is greater than 0.05. The power law confirmity indicates the network formed by the user defined functions is a scale free network.

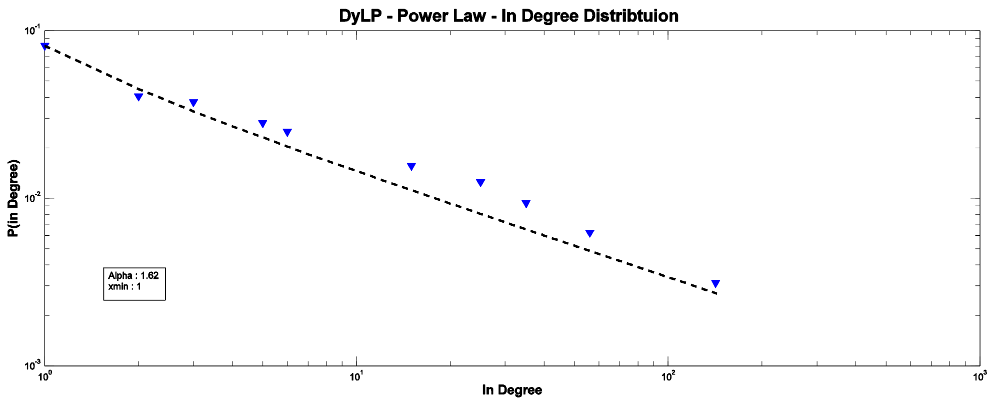


Figure 6.15: Power Law Distribution for In Degree of DyLP

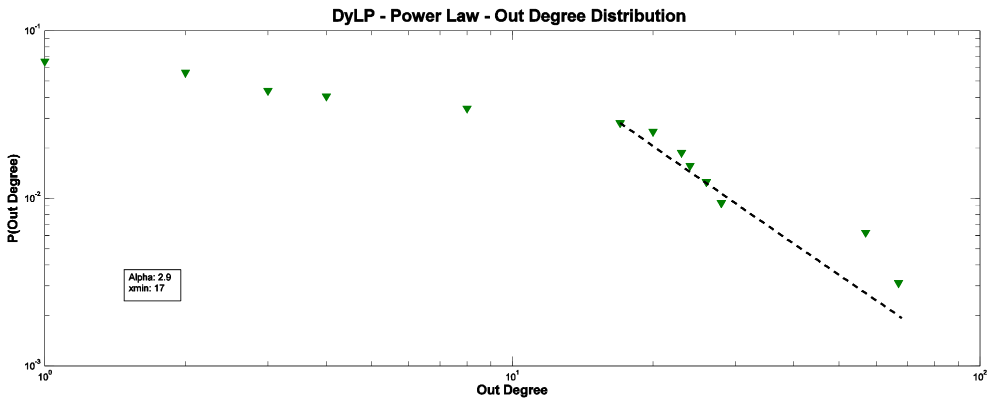


Figure 6.16: Power Law Distribution for Out Degree of DyLP

6.4 Discussion

Braha *et al.*[10] measured the clustering coefficient of four systems: vehicle design, operating system software, pharmaceutical facility design and hospital facility design to be 0.205, 0.327, 0.449 and 0.274 respectively. Our studied scientific software: ADOL-C, BCP, CppAD and DyLP displayed clustering co-efficent of 0.08, 0.03, 0 and 0.2489 respectively. Hence, AD tools have one magnitude lower clustering coefficient than general purpose software where is DyLP has similar clustering coefficient. This indicates three of the four software (except DyLP) has less modular structure than general purpose commercial software.

Structural Properties	General Purpose Commercial Software	Scientific Software
Characteristic Path Length	2.8-3.7	2.2-2.7
Clustering coefficient	0.2-0.45	0-0.25
Average Nodal Degree	7-20	3-8
Propagation Cost	5-17	3-7
Feedback Marks	Present	Absent

Table 6.21: Structural metrics - General purpose software Vs scientific software

The characteristic path length found by Braha *et al.*[10] for vehicle design network is 2.878, operating system software is 3.7, pharmaceutical facility is 2.628 and hospital facility is 3.118. Our studied scientific software displayed a characteristic path length from 2.2 to 2.7. Hence, the call graphs of studied software tools displayed shorter characteristic path lengths when compared with general-purpose software. This indicates the computation performance of scientific software is better than general purpose commercial software.

The average nodal degree for the four systems: vehicle design, operating system software, pharmaceutical facility design and hospital facility design measured by Braha *et al.*are approximately 7, 10, 15 and 20 respectively. Whereas our studied scientific software displayed nodal degree between 3 and 8. Hence, the call graphs of studied software tools displayed small average nodal degree than general-purpose software. This indicates scientific software has less number of dependencies between system elements.

MacCormack *et al.*[35] studied Mozilla Firefox and Linux operating system. They found the propagation cost of 5.82% and 17.85% for Mozilla and Linux respectively. Our studied software displayed propagation cost within range 3-7. This indicates the scientific research software is less sensitive to structural changes than the general purpose commercial software.

MacCormack *et al.*also found the presence of feedback mark in the associated partitioned DSM of the two softwares. Our studied partitioned DSM of the studied software displays the absence of feedback marks or circular dependencies. Hence, this also indicates the optimized computation performance of scientific software compared to the general purpose commercial software. A recent study by Sosa *et al.*[62] indicates the presence of circular dependencies exhibit higher level of defects in the system. This provides us with an intuition that scientific software exhibit lower level of defects.

Braha *et al.*[10] found that the four systems: vehicle design, operating system software, pharmaceutical facility design and hospital facility design are scale free networks. Our results indicate that four studied software: ADOL-C, BCP, CppAD and DyLP exhibit scale free properties. So scientific software has similar fault tolerance behavior as general purpose commercial softwares.

Software	Old Release	New Release	m	C	Rc_n
ADOLC	V1.10.0	V2.1.0	116	0.106083	13.696
DyLP	V1.3.0	V1.4.0	65	0.258967	12.896
CppAD	V110308	V111103	33	0.0363174	3.5859
DyLP	V 1.4.0	V 1.5.0	21	0.258967	4.2487

Table 6.22: Clustering Coefficient Vs Release Rework Cost

Iterative release analysis provided an important observation. Table 6.22 displays a relationship established between clustering coefficient and release rework cost from our test results. For one major change, ADOL-C introduced double the new architectural elements compared to DyLP but the release rework cost is similar. Another major change displays higher release rework cost for DyLP, although it introduced 1.5 times less new elements than CppAD. Thus, higher clustering coefficient implies that there might be higher rework cost for the later release though the propagation cost may be similar.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we perform design structure and iterative release analysis for scientific software from multiple application domain. As a first step, we implemented an efficient DSM partitioning algorithm. Later on, we compute a number of architectural complexity metrics of four open source scientific research software: ADOL-C, BCP, CppAD and DyLP from three application domain. Finally, we compare and compute total release cost for thirty seven releases of above four software tools. To the best of our knowledge this is the first attempt to analyze the design structure and iterative release of scientific research software.

Efficient DSM partitioning has been achieved by implementing Tarjan's algorithm using sparse matrix data structure CRS. In order to analyze the design structure, we extracted the source code dependency from four scientific software. Using the source code dependency, we constructed DSM and computed characteristic path length, clustering coefficient, nodal degree, propagation cost and centrality measure. Moreover, we analyzed the partitioned DSM and power law behavior of degree distribution from the software tools. Iterative release analysis involved segregation of new user-defined function from previous releases. We then extracted the dependency between newly implemented functions and functions implemented on previous releases to calculate the release rework cost. Lastly, the total implementation cost is computed for all releases and compared.

Our implemented DSM partitioning provided better timing than those obtained from C/C++ Boost library implementation of Tarjan's algorithm, approximately 5 times faster than that of boost one. This results indicates DSM partitioning using sparse matrix data structure leads to savings in computational work and intermediate storage. The call graphs for the ADOL-C and CppAD software tools displayed shorter characteristic path lengths, small nodal degrees, and small propagation costs, similar to general-purpose software such as operating systems [35] [10]. However, variation is observed in clustering coefficient and characteristic path length for DyLP and BCP. DyLP has clustering coefficient similar to operating systems while on the other hand, BCP has clustering coefficient of zero. Clustering coefficient of zero implies that BCP has tree like design structure where vertices are the user-defined functions. In addition, Bcp has one magnitude shorter characteristic path length compared with others. A relatively small clustering coefficient in ADOL-C and CppAD points to a less modular design structure than DyLP. Absence of circular dependencies in the studied tools can be attributed to the strong emphasis placed on the computational performance of the code. This also indicates the design structure of all studied software is typically a directed acyclic graph(DAG). While studying the centrality measure, we note CppAD heavily uses object-oriented features compared with others as a constructor function is one of the most frequently called member function.

Iterative release analysis provided a good insight regarding the development cost of the software releases. Across all the version of any particular software except BCP, we have observed the clustering coefficient decreased across versions. This provides us with an intuition that upgraded releases enhance the functionality but reduces the modularity of scientific software. Our test results also indicates that release rework cost might have some correspondence with the clustering coefficient of the scientific software. The release rework cost might be higher for later releases if modularity of previous release is higher. In addition, except for a major change in ADOL-C, the most central function remained the same in all the releases. This indicates the kernel components of scientific software undergo very minor change in iterative releases to ensure the stability of the product.

7.2 Future Work

There can be a number of extensions for future research in addition to more detailed analysis of structural metrics and iterative releases. First, a development of domain-specific centrality metrics will enable us to have better understanding of scientific software from multiple domains. These scientific software tools are often integrated into larger system to achieve better efficiency and performance. Secondly, it will be interesting to know how much integration effort is required when these scientific software are integrated into larger systems. We envision to perform these research in future.

Bibliography

[1] ALBERT, R., AND BARABÁSI, A. Statistical mechanics of complex networks. *Reviews of modern physics 74*, 1 (2002), 47.

[2] ALBERT, R., JEONG, H., AND BARABÁSI, A. The diameter of the world wide web. *Arxiv preprint cond-mat/9907038* (1999).

[3] ALEXANDERSON, G. About the cover: Euler and konigsberg's bridges: A historical view. *Bulletin of the American Mathematical Society 43*, 4 (2006), 567.

[4] AUBER, D. Tulip-a huge graph visualization framework. *Graph Drawing Software* (2003), 105–126.

[5] BALDWIN, C., AND CLARK, K. *Design rules: The power of modularity*, vol. 1. The MIT Press, 2000.

[6] BALLARD, G. Positive vs negative iteration in design. In *Proceedings Eighth Annual Conference of the International Group for Lean Construction, IGLC-6, Brighton, UK* (2000).

[7] BELL, B. Cppad: a package for differentiation of c++ algorithms. 2008 06-01]. http://www. coin-or. org/CppAD, 2011.

[8] BENEDIKTSSON, O., DALCHER, D., REED, K., AND WOODMAN, M. Cocomo-based effort estimation for iterative and incremental software development. *Software Quality Journal 11*, 4 (2003), 265–281.

[9] BISSELING, R., BYRKA, J., CERAV-ERBAS, S., GVOZDENOVIC, N., LORENZ, M., PENDAVINGH, R., REEVES, C., RÖGER, M., AND VERHOEVEN, A. Partitioning a call graph. In *Proceedings of the 52nd European Study Group with Industry* (2005), pp. 95–108.

[10] BRAHA, D., AND BAR-YAM, Y. The statistical mechanics of complex product development: Empirical and analytical results. *Management Science 53*, 7 (2007), 1127–1145.

[11] BRAUNE, B., DIEHL, S., KERREN, A., AND WILHELM, R. Animation of the generation and computation of finite automata for learning software. *Automata Implementation* (2001), 39–47.

[12] BROWN, N., NORD, R., OZKAYA, I., AND PAIS, M. Analysis and management of architectural dependencies in iterative release planning. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture* (2011), IEEE, pp. 103–112.

[13] Browning, T. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *Engineering Management, IEEE Transactions on 48*, 3 (2001), 292–306.

[14] Cheriyan, J., and Mehlhorn, K. Algorithms for dense graphs and networks on the random access computer. *Algorithmica 15*, 6 (1996), 521–549.

[15] Clauset, A., Shalizi, C., and Newman, M. Power-law distributions in empirical data. *Arxiv preprint arxiv:0706.1062* (2007).

[16] Collard, M., Kagdi, H., and Maletic, J. An xml-based lightweight c++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on* (2003), IEEE, pp. 134–143.

[17] Cormen, T. *Introduction to algorithms*. The MIT press, 2001.

[18] Davis, T., et al. University of florida sparse matrix collection. *NA digest 97*, 23 (1997), 7.

[19] Dawes, B., Abrahams, D., and Rivera, R. Boost c++ libraries. *URL http://www. boost. org* (2009).

[20] Duff, I., and Reid, J. An implementation of tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software (TOMS) 4*, 2 (1978), 137–147.

[21] Eppinger, S. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design 2*, 4 (1991), 283–290.

[22] Eppinger, S., Whitney, D., Smith, R., and Gebala, D. A model-based method for organizing tasks in product development. *Research in Engineering Design 6*, 1 (1994), 1–13.

[23] Freeman, L. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[24] Greer, D., and Ruhe, G. Software release planning: an evolutionary and iterative approach. *Information and Software Technology 46*, 4 (2004), 243–253.

[25] Griewank, A., Juedes, D., and Utke, J. Algorithm 755: Adol-c: a package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software (TOMS) 22*, 2 (1996), 131–167.

[26] Hafer, L. Dylp: a dynamic lp code. Tech. rep., Technical Report SFU-CMPT TR 1998-23, School of Computing Science, Simon Fraser University, Burnaby, BC, V5A 1S6, 1998.

[27] HOOGENDORP, H. Extraction and visual exploration of call graphs for large software systems. *Order 501*, 3063.

[28] HOROWITZ, E., AND SAHNI, S. *Fundamentals of data structures*. Computer science press, 1983.

[29] HOSSAIN, S. Efficiently computing with design structure matrices. In *Proceedings of the 12th International DSM Conference–Managing Complexity by Modelling Dependencies* (2010), pp. 345–358.

[30] HUNSAKER, B. Coin-or bcp(accessed nov 2011). `http://www.coin-or.org/projects/Bcp.xml`.

[31] KUSIAK, A., AND WANG, J. Efficient organizing of design activities. *The International Journal Of Production Research 31*, 4 (1993), 753–769.

[32] LAMANTIA, M. *Dependency Models as a Basis for Analyzing Software Product Platform Modularity: A Case Study in Strategic Software Design Rationalization*. PhD thesis, Massachusetts Institute of Technology, System Design and Management Program, 2006.

[33] LEVITIN, A. *Introduction to the design & analysis of algorithms*. Addison-Wesley Reading, MA, 2003.

[34] LIN, Y., HOLT, R., AND MALTON, A. Completeness of a fact extractor. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE03)* (2003), vol. 1095, pp. 17–00.

[35] MACCORMACK, A., RUSNAK, J., BALDWIN, C., AND OF RESEARCH, H. B. S. D. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science 52*, 7 (2006), 1015.

[36] MARGOT, F. Bac: A bcp based branch-and-cut example. *Tepper School of Business. Paper 263* (2010).

[37] MCCORD, K., EPPINGER, S., AND OF MANAGEMENT, S. S. Managing the integration problem in concurrent engineering. Master's thesis, Massachusetts Institute of Technology, Dept. of Mechanical Engineering, 1993.

[38] MEHLHORN, K., NÄHER, S., AND SANDERS, P. Engineering dfs-based graph algorithms, 2007.

[39] MEIER, C., YASSINE, A., AND BROWNING, T. Design process sequencing with competent genetic algorithms. *Journal of Mechanical Design 129* (2007), 566.

[40] MILGRAM, S. The small world problem. *Psychology today 2*, 1 (1967), 60–67.

[41] MURPHY, G., NOTKIN, D., GRISWOLD, W., AND LAN, E. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM) 7*, 2 (1998), 158–191.

[42] MYERS, C. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E 68*, 4 (2003), 046116.

[43] NEDERHOF, M. Generalized left-corner parsing. In *Proceedings of the sixth conference on European chapter of the Association for Computational Linguistics* (1993), Association for Computational Linguistics, pp. 305–314.

[44] NEWMAN, M. The structure and function of complex networks. *SIAM review* (2003), 167–256.

[45] NEWMAN, M. A measure of betweenness centrality based on random walks. *Social networks 27*, 1 (2005), 39–54.

[46] NIELSEN, S. S. *Programming languages and systems in computational economics and finance*, vol. 18. Springer, 2002.

[47] PADBERG, M. *Linear optimization and extensions*, vol. 12. Springer Verlag, 1999.

[48] PARNAS, D. Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on*, 2 (1979), 128–138.

[49] PEKTAS, S., AND PULTAR, M. Modelling detailed information flows in building design with the parameter-based design structure matrix. *Design Studies 27*, 1 (2006), 99–122.

[50] PIMMLER, T., AND EPPINGER, S. *Integration analysis of product decompositions*. Alfred P. Sloan School of Management, Massachusetts Institute of Technology, 1994.

[51] PINAR, A., AND HEATH, M. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)* (1999), ACM, p. 30.

[52] POTHEN, A., AND FAN, C. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software (TOMS) 16*, 4 (1990), 303–324.

[53] ROGERS, J., MCCULLEY, C., AND BLOEBAUM, C. Optimizing the process flow for complex design projects. *Optimization 1* (1999), 3.

[54] ROUIBAH, K., AND CASKEY, K. Change management in concurrent engineering from a parameter perspective. *Computers in Industry 50*, 1 (2003), 15–34.

[55] SANDER, G. Vcg visualization of compiler graphs, 1995.

[56] SEGAL, J. Models of scientific software development. In *First International Workshop on Software Engineering in Computational Science and Engineering, SECSE 08* (2008).

[57] SHARIR, M. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications 7*, 1 (1981), 67–72.

[58] SMITH, F., LYONS, S., ERNEST, S., JONES, K., KAUFMAN, D., DAYAN, T., MARQUET, P., BROWN, J., AND HASKELL, J. Body mass of late quaternary mammals. *Ecology 84*, 12 (2003), 3403–3403.

[59] SOSA, M. A structured approach to predicting and managing technical interactions in software development. *Research in Engineering Design 19*, 1 (2008), 47–70.

[60] SOSA, M., BROWNING, T., AND MIHM, J. Studying the dynamics of the architecture of software products. In *Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007)* (2007), pp. 4–7.

[61] SOSA, M., MIHM, J., AND BROWNING, T. Degree distribution and quality in complex engineered systems. 2011.

[62] SOSA, M., MIHM, J., AND BROWNING, T. Product architecture and quality: A study of open-source software development. 2011.

[63] STAPEL, E. Linear programming: Introduction (accessed apr 2012). `http://www.purplemath.com/modules/linprog.htm`.

[64] TARJAN, R. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on* (1971), IEEE, pp. 114–121.

[65] TELEA, A., HOOGENDORP, H., ERSOY, O., AND RENIERS, D. Extraction and visualization of call dependencies for large c/c++ code bases: A comparative study. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on* (2009), IEEE, pp. 81–88.

[66] VOSE, D. Fitting distributions to data. `http://www.vosesoftware.com/`, 2010.

[67] WALTHER, A., GRIEWANK, A., AND VOGEL, O. Adol-c: Automatic differentiation using operator overloading in c++. *PAMM 2*, 1 (2003), 41–44.

[68] WEISSTEIN, E. W. Vertex-induced subgraph. *MathWorld–A Wolfram Web Resource http://mathworld.wolfram.com/Vertex-InducedSubgraph.html* (2012).

[69] WILKERSON, D., CHEN, K., AND MCPEAK, S. Oink: a collaboration of c++ static analysis tools (accessed 2007). `http://www.cubewano.org/oink/`, 2007.

[70] YASSINE, A. An introduction to modeling and analyzing complex product development processes using the design structure matrix (dsm) method. *Urbana 51*, 9 (2004), 1–17.

[71] YASSINE, A., AND BRAHA, D. Complex concurrent engineering and the design structure matrix method. *Concurrent Engineering 11*, 3 (2003), 165–176.

[72] YASSINE, A., AND FALKENBURG, D. A framework for design process specifications management. *Journal of Engineering Design 10*, 3 (1999), 223–234.