Analysis of Strongly Connected Components (SCC) Using Dynamic Graph Representation

Saleh Alshomrani [1], Gulraiz Iqbal [1]

[1] Department of Information Systems
Faculty of Computing and Information Technology
King Abdulaziz University, Jeddah, Saudi Arabia

Abstract

Graphs are the basis of many real life applications. In our research we compare and analyse strongly connected components algorithm by using general techniques for efficient implementation. This experimental procedure exemplify in two contexts. 1. Comparison of strongly connected components algorithms. 2. Analysis of particular algorithm.
Such a practice will enable java programmers, especially for those who work on such algorithms to use them efficiently. In this paper we described algorithms implementation, test and benchmark to experiment the performance of algorithms. During experimenting we found some interesting results as Cheriyan-Mehlhorn-Gabow algorithm outperform then Tarjan's algorithm

Keywords: *Graph, Directed Graph, SCC (Strongly Connected Components), Java, Benchmark.*

1. Introduction

Graphs are fundamental to many problems in computer science as well in chemistry, biology and physics. Pairwise relation between objects e.g. Computer networks (Switches, routers and other devices are vertices and edges are wire / wireless connection between them), electrical circuits (vertices are diodes, transistors, capacitors, switched etc. and edges are wire connection between them), World Wide Web (webpages are vertices and hyperlink are edges) and Molecules (vertices are atoms and edges are bond between them) all benefits from the pairwise model. Problem that arises on graph is the performance because of complex graph algorithms and size [5, 6].

Modern processors are complex computing engines; the running time of a computer is determined by three factors.

1. Number of instruction executed
2. Efficient use of processor pipeline
3. Efficient use of cache memory

Utilizing cache, processor and exceeding the number of instruction cause high penalty [3]. Considering these factors we decided to implement our own graph structure instead using existing libraries like JUNG and JGraphT. Our own structure will help us to made several choices instead of re-engineering and making understanding of the libraries. We make a start on generic graph library by choosing dynamic data structure (a link list as an adjancy list) instead of using static data structure (adjancy matrix). In this paper we consider the above mentioned general techniques to implement efficient graph algorithms, evaluate and exemplify them on two strongly connected components of directed graph.

2. Notation & basic definition of directed graph

A directed graph G is a finite set of vertices V and set of directed edges E that forms the pair (V, E) and $E \subseteq V \times V$ is a set of directed graph. If $(v, u) \in E$, then u is called immediate successor of v, and v is called immediate predecessor of u.
Undirected graphs may be observed as a special kind of directed graphs, where directions of edges are unimportant $(v, u) \in E \leftrightarrow (u, v) \in E$ [2, 6]. A directed graph G = (V, E) is called strongly connected if there is a path between v to u and u to v. Graphs are used in many applications as listed below[6].

2.1 Social Applications

Graphs are popular to manage build relation and maintain their information. Vertices are different people and edges are the relations among those people. This can be used in management hierarchical, family tree, social media's such as Facebook or LinkedIn [5, 9].

2.1.1 Road Maps

Here vertices represent crossings and edges represent streets, vertices could be cities and edges could be the weighted link to represent the distance between two cities. In this case, each edge may also have an associated distance [5, 9].

2.1.2 Airline Route Maps

Graph can be used for airline route maps. Here vertices represent airports and edges are the direct or indirect link from one airport to another. An edge may have a weight to represent the cost of the flight [5, 9].

2.1.3 Computer Network Applications & WWW

Graphs play a critical modeling role in networks, where vertices are devices e.g. switch routers, etc. where edges are links to connect vertices. Edges may have distance and capacities associated with it [5, 9].

3. Literature Review

To start working on this library extensively literature reviewed on graphs to understand the theoretical and practical approaches to design and develop a flexible graph library suitable for graph algorithms to implement, test and analyze graph performance using the library benchmark. Many graph libraries are available in java as well in other languages. Most of the java libraries are using sequential approaches and slower over large graphs. According to Kurt, Stefan, and Peter mention optimization technique in their paper [3], we also adopted their technique and compare our strongly connected components algorithms of two authors and will compare the results. In our later research we will comparatively analyze with other libraries algorithm to make them fast.

3.1 Jung

The java universal network / graph framework is an open source library provides extensive modelling, analysis and visualization tool for the graph or network. JUNG architecture have flexible support to represent the entities and their relations, such as directed and undirected graph, hyper graphs, and graphs with parallel edges. It also includes a number of graph theory, data mining, social network, optimization and random graph generator [12].

3.2 JGraphT

JGraphT is an open source Java graph library using structured approach to implement graph algorithms. Most of the library classes are generic for the ease of users. In this library several graph algorithms implemented using structured approach [11].

During our work we made several choices one of them is either we have to use the existing libraries and implement different strongly connected component algorithms to analyse the performance after possible optimization or start from the scratch.

4. Graph Representation

According to Mark.C.Chu-Carroll, to represent graphs in computer programs and some compromise between different representations, there are two techniques to represent graphs.

4.1 Adjancy Matrix / Matrix Base Representation

A Graph G with N number of vertices, an adjancy matrix is N×N matrix of 0/1 values, where a pair [a,b] is 1 only if there is an edge between a and b, otherwise 0.
If Graph is undirected then the matrix is symmetric [a,b] = [b,a]. In case of directed graph then [a,b]=1 means that there is an edge from a to b[10].

$$Mi,j = \begin{cases} 1 & (ai, bj) \in R \\ 0 & (aj, bi) \notin R \end{cases} \quad (1)$$

4.2 Adjancylist / List based representations

An alternative representation for a graph G (V, E) is adjancy list. For each vertex we keep a list of all other vertices adjacent to the current vertex. We say that vertex A is adjacent to vertex B if (A, B) $\in$ E. In our experiments we are using adjancy list with minor improvements to avoid from iterations. In our implementation we maintain a list of all nodes adjacent to the current node. Adjancy list time complexity is O (n+m) [10].

Adjancy matrix is suitable when edges don't have data associated with them. In case of sparse graph adjancy matrix is poor in performance and waste huge amount of memory. Comparatively adjancy list is efficient in case of sparse graph, it stores only the edges present in the graph and can store data associated to edges. Although there is no clear suggestion which graph representation is better, we consider the above situation we selected adjancy list representation for our experiments [10].

5. Strongly Connected Components

A directed graph G = (V, E) is strongly connected if there is a path from vertex a to b and b to a or if a sub graph is connected in a way that there is a path from each node to all other nodes is a strongly connected sub graph. If the whole graph has the same property, then the graph is strongly connected [6].

Strongly connected components can be computed using different approaches introduced by Tarjan's, Gabow and Kosaraju's. Two of them like Tarjan's and Gabow algorithm require only one DFS, but Kosaraju's algorithm

required two DFS. We did not included Kosaraju's algorithm in our current experiments.

5.1 Depth First Search Algorithm

Depth first search is a technique to explore a graph using stack as the data structure. It starts from the root of the graph, explore its first child, explore the child of next vertex until reach to the goal vertex or reach to final vertex having no further child. Then, back tracking is used to return the last vertex which is not yet completely explored. Modifying the post-visit and pre-visit, DFS is used to solve many important problems and it takes O (|V|+|E|) steps.

5.2 Tarjan's Algorithm

Tarjan's strongly connected components algorithm accept directed graph as an input and in a collection containing all possible components. The algorithm explores all nodes of the graph by using depth first search, it begins from an arbitrary start node and silently ingnore the nodes already visited. The nodes placed in a stack in order they explored and maintaining index number (node.index) as well each node maintain a lowlink which is always lower than the current node index. The current node is the root node of strongly connected components if node.lowlink = node.index. A sub graph is returned if it's determined that it's a strongly connected component [1, 2, 3].

5.3 Cheriyan-Mehlhorn-Gabow Algorithms

Gabow strongly connected component is also same like Tarjan's algorithm. It accepts a directed graph as an input and result containing collection of all possible strongly connected components. It also uses depth first search to explore all the nodes of the directed graph. Gabow algorithm maintains two stacks, one of them contains a list of nodes not yet computed as as strongly connected components and other contains a set of nodes not belong to different strongly connected components. A counter is used to count number of visited nodes, which is used to compute preorder of the nodes [2, 3, 4].

6. Implementation

In our implementation we used only dynamic graph data structure which used linked lists for the adjancy list. The graph generator class make sure that each vertex is stored in consecutive location in adjancy list, in fact dynamic implementation consume more space then static graph data structure. Graph structure package contains interfaces, abstract classes to provide interface to different type of graphs such a Directed Graph. All classes mentioned in our implementation are Generic and user can use them by their own way. Graph package contains many interfaces for different graphs and also interfaces for the different algorithms describing the required methods for algorithm to implement. The undirected graph is not currently used but in future might be included.

The directed graph interface defines many methods to implement where each node represents a unique data member of generic type and two nodes can't be added to the graph if they representing the same node. The second attempt will simply ignored and also multiple edges between two nodes are not allowed. An abstract class node that also serves as an interface for the vertex of directed graph, maintains a list of its successors and predecessors.

7. Experiments

In our experiments we used random graph and complete graph. We performed three sets of experiments. We used random graph with minimal edges varying n=2 as a sparse graph and another with maximum edges n = 100 as a dense graph. In our third set of experiments we used complete graph where edges are n (n-1)/2. Random graphs are not an easy case to analyse because of random nodes, edges and memory allocated them.

In our implementation we used dynamic graph data structure using linked list for the adjancy list. Modern processors are complex computing engines. We report running time on one Intel® Core™ i5-2410M CPU @2.30GHz with 4 GB of memory. We also obtained the similar results on AMD.

We used eclipse version Helios Service Release 2 as IDE for java developers and for our experiment we increased the heap size by providing the argument -Xms128m -Xmx1550m -XX: +UseParallelGC. For recursive calls stack size is also important. In some scenarios on a large number vertices and edges stack over flow error occurs.

7.1 Experiments on Tarjan's Algorithm

Generating six random graph for each graph (Dense, Sparse and Complete) with minimum edges E=2 for sparse graph, maximum edges E=100 for dense graph and two complete graph with n (n-1)/2 edges.
We computed their running time and memory as Figure 1 & Figure 3 shows the difference between dense, sparse and complete graph on N number of nodes. Tarjans's algorithm compute strongly connected components efficiently when number of edges is lower. So edges have a direct impact on its running time and memory.

7.1.1 Time Required by Tarjan's Algorithm

Results generated by our benchmark are represented using line chart for clear understandings in Figure 1 & 2 indicating, as the number of nodes and edges grows Tarjan's strongly connected component algorithm take more time to run. Figure 2 clarify the results of Figure 1 as it displays a minor difference on dense and sparse graph.

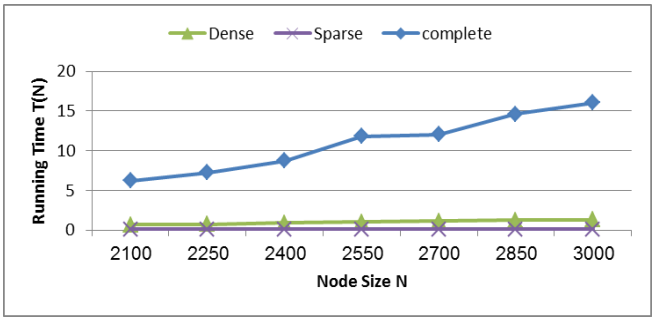

Figure 1 Tarjan's running time (in seconds) difference on Dense, Sparse & Complete Graph in dynamic graph representation

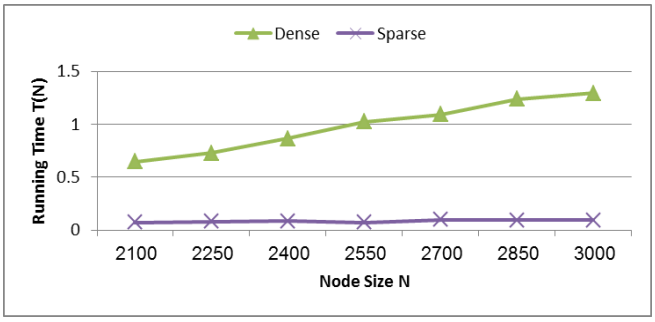

Figure 2 Tarjan's algorithm running time (in seconds) difference on Dense & sparse graph using dynamic representation

7.1.2 Memory Required by Tarjan's Algorithm

A clear impact on memory required can be analysed by Figure 3 which displays that as number of nodes and number of edges increase, Tarjan's algorithm required more memory to complete the task.

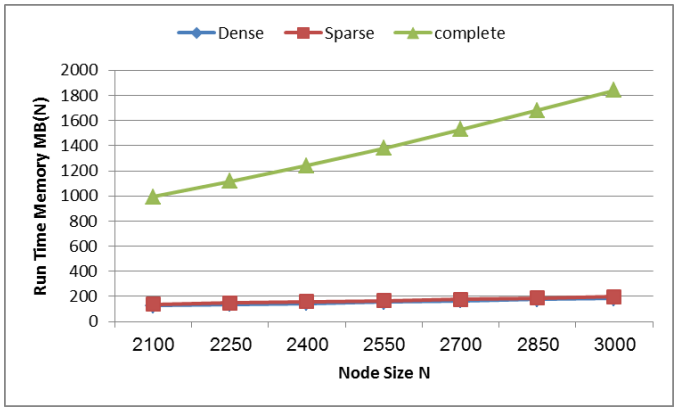

Figure 3 Tarjan's memory difference on Dense, Sparce & complete in dynamic graph representation

7.2 Experiments on Gabow's Algorithm

We had the same set of experiments for Gabow's algorithm, for each graph (Dense, Sparse and complete) we generated six random graph with minimum edges E=2 for sparse graph, maximum edges E=100 for dense graph and two complete graph with n (n-1)/2 edges.

Therefore we computed their running time and memory as Figure 4 & 5 shows the difference between dense, sparse and complete graph on N number of nodes. Gabow's algorithm compute strongly connected components efficiently when number of edges is lower. So edges have a direct impact on its running time and memory. In figure 4 & 5 its bit tricky to analyse the difference between dense and sparse graph as it seems that it take almost zero time and memory to compute strongly connected components. To clarify the results we draw figure 6 which state the clear difference when number of nodes and number of edges increased.

7.2.1 Time Required by Gabow's Algorithm

Results generated by our benchmark are represented using line chart for clear understandings in Figure 4 & 5 indicating, as the number of nodes and edges grows Gabow's scc algorithm take more time to run. Figure 5 clarify the results of Figure 4 as it displays a minor difference on dense and sparse graph.

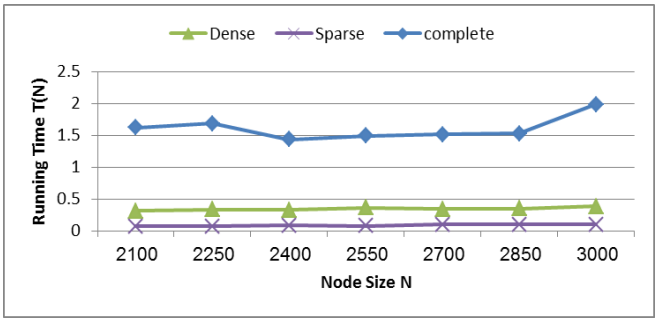

Figure 4 Gabow's running Time (in seconds) difference on Dense, Sparse & Complete in dynamic graph representation

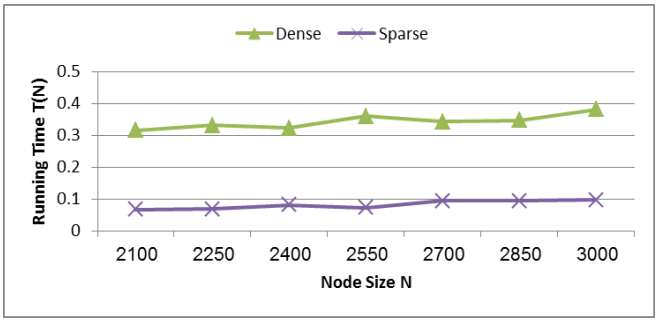

Figure 5 Gabow's algorithm running time (in seconds) difference on Dense & sparse graph using dynamic representation

7.2.2 Memory Required by Gabow's Algorithm

A clear impact on memory can be analysed by Figure 6 which displays that as number of nodes and number of edges increase, Gabow's algorithm required more memory to complete the task.

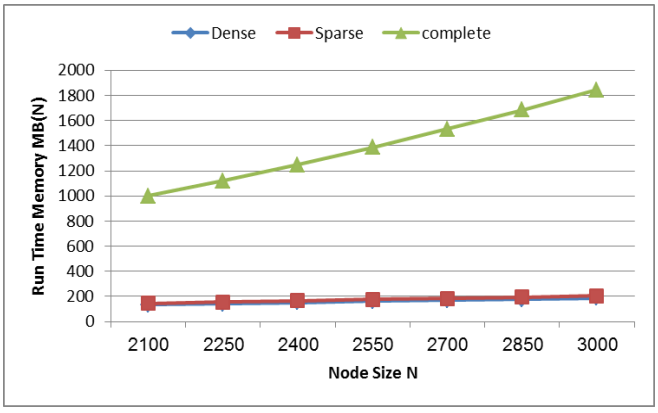

Figure 6 Gabow's algorithm memory difference on Dense, Sparse & Complete in dynamic graph representation

7.3 Comparison of Gabow & Tarjan's Algorithm

Utilizing the same data set, combining data to compare Tarjan's and Gabow's algorithm running time in seconds. In figure 7, time is computed on dense graph where E=100, for both Tarjan's and Gabow's algorithm. One can easily analyse the results that Gabow's taking less time to compute strongly connected components of directed graph.

In Figure 8 running time is computed using sparse graph with minimum edges E=2. In results one can analyse that Gabow's and Tarjan's have the same performance when number of edges are small, Whereas Figure 9 is based on complete graph, where Gabow's algorithm outperform Tarjan's algorithm with a big margin. Gabow's performance remains outstanding as number of nodes and edges increase whereas Tarjan's algorithm remain slow as number of edges increased.

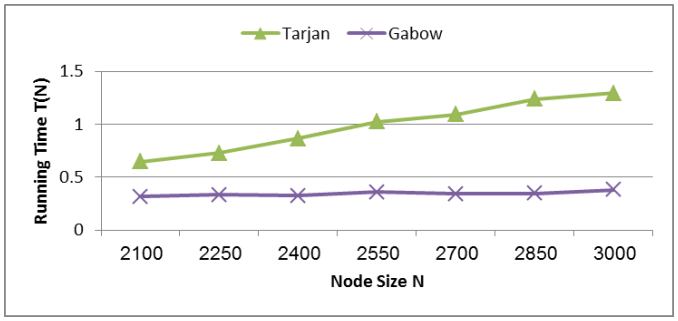

Figure 7 Gabow's vs Tarjan's running time (in seconds) comparison on Dense in dynamic graph representation

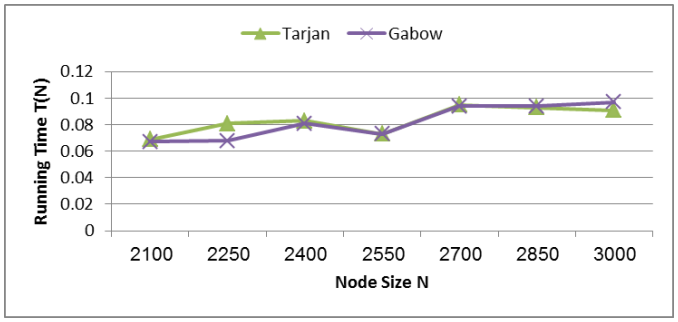

Figure 8 Gabow's vs Tarjan's running time (in seconds) comparison on sparse in dynamic graph

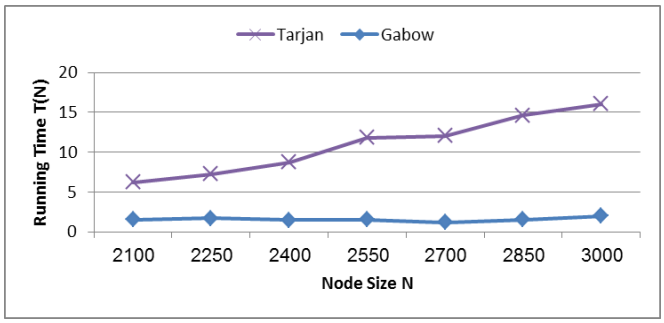

Figure 9 Gabow's vs Tarjan's running time comparison on complete in dynamic graph

According to results obtained from benchmark memory consumption is almost equal on both Tarjan's and Gabow's algorithms but have different statistics on time.

8. Conclusions

In our research, we analyzed & compared Tarjan's and Gabow's strongly connected component algorithms to find their suitability for various applications. We generated dense, sparse and complete graphs randomly to compute memory difference of the both algorithms. According to our implementation and experiments, we found that Gabow algorithm tend to become shorter, simpler and more elegant. Tarjan's performance is similar to Gabow's algorithm when number of edges in a graph is minimum level.

9. Future Work

During this research, our experiments were under some limitations; in worst case scenario we generated six graphs with N=3000 to compute memory and time. In future we can experiment a large graph while increasing the stack size and java VM heap size.
In our current research we were limited to only Tarjan's and Gabow's algorithm, in our future work we can include the algorithms of S.Rao Kosaraju & Sharir and Brute algorithms using different approach to compute strongly connected components.

Appendix A: Tuned Cheriyan-Mehlhorn-Gabow SCC Algorithm Implementation

```
public class GhaiboSCC<E> implements SCC<E> {

private Collection<Collection<Node<E>>> sccList;

private List<Integer> B;
private List<Node<E>> S;

private int c;
Node<E> n;

protected int counterG;

public Collection<Collection<Node<E>>> scc(DirectedGraph<E> graph) {
        S = new ArrayList<Node<E>>();
        B = new ArrayList<Integer>();

sccList = new ArrayList<Collection<Node<E>>>();
Iterator<Node<E>> nodes = graph.iterator();

int N = 0;

        while (nodes.hasNext()) {
                n = nodes.next();
                n.num = 0;
                N++;
                }

                c = N;

Iterator<Node<E>> nodeIt = graph.iterator();
Node<E> node;
        while (nodeIt.hasNext()) {
                node = nodeIt.next();
                if (node.num == 0)
                        DFS(node);
                }

        return sccList;
}

private void DFS(Node<E> v) {
        Node<E> w;
        S.add(v);
        v.num = (S.size() - 1);
        B.add(v.num);
Iterator<Node<E>> node = v.succsOf();

while (node.hasNext()) {

        w = node.next();
        if (w.num == 0)
                DFS(w);
        else {
        while (w.num < B.get((B.size() - 1)))

        B.remove(B.size() - 1);
        }
        }

List<Node<E>> L = new ArrayList<Node<E>>();

if (v.num == (B.get(B.size()-1 ))) {

        B.remove(B.size() - 1);
        c++;

while (v.num <= (S.size()-1)) {
                L.add(S.get(S.size()-1));
        S.remove(S.size() - 1);
}
sccList.add(L);
        }
 }
}
```

Appendix B: Tarjan's SCC Algorithm Implementation

```
public class TorjanSCC<E> implements SCC<E> {
private int index = 0;
private ArrayList<Node<E>> stack;
private Collection<Collection<Node<E>>> SCC = new ArrayList<Collection<Node<E>>>();

public Collection<Collection<Node<E>>> scc(DirectedGraph<E> graph){
        stack = new ArrayList<Node<E>>();
        SCC.clear();
        index = 0;
```

```
if(graph != null){
    Iterator<Node<E>> nodes = graph.iterator();
            while (nodes.hasNext()) {
                    Node<E> node = nodes.next();
                  if(node.num == -1)
                  {
                    visit(node, graph);
                   }
                  }
                  }
return SCC;
}
private void visit(Node<E> v,DirectedGraph<E> graph){

 v.num = index;
 v.lowlink = index;
 index++;

 stack.add(0, v);

Iterator<Node<E>> nodes = v.succsOf();

Node<E> w;

while (nodes.hasNext()) {
            w = nodes.next();

      if(w.num == -1){
         visit(w, graph);
          v.lowlink = Math.min(v.lowlink, w.lowlink);

       }else if(stack.contains(w)){
          v.lowlink = Math.min(v.lowlink, w.num);
       }
    }
    if(v.lowlink == v.num){
       Node n;

 ArrayList<Node<E>> component = new ArrayList<Node<E>>();

    do{
            n = stack.remove(0);
            component.add(n);
     }while(n != v);
            SCC.add(component);
    }

 }
}
```

References

[1] J.E.Hopcroft and R.E. Tarjan. Dividing a graph into triconnected compoents. SIAM Journal on Computing. 2(3): 135-158, 1973

[2] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska, Computing Strongly connected components in parallel on CUDA, IEEE 2011 International Parallel & Distributed Processing Symposium

[3] Kurt Mehlhorn, Stefan Naher and Peter Sanders, Engineering DFS based Graph Algorithms, Partially supported by DFG grant SA 933/3-1, 2007

[4] H.N. Gabow. Path-based depth first search strong and biconnected components, Information Processing Letters, 74(3-4):107-114, 2000

[5] Marije de Heus, Towards a Library of Parallel Graph Algorithm in Java, 14th Twente Student conference on IT January 21st 2011

[6] Robert Sedgewick, Kevin Wayne, The Text Book Algorith 4th Edition http://algs4.cs.princeton.edu/home/ retrieved on 04-2012

[7] Stefan Steinhaus, The text book, Comparisons of mathematical programs for data Analysis (Edition 5.04) July 2008

[8] Jiri Barnat, Jakub Chaloupka, Jaco van de Pol, Distributed algorithms for SCC decomposition, Journal of Logic and Computation, volume 21(1), 2011, 23-44

[9] David Easley and Jon Kleinberg, Reasoning about a highly connected world, Textbook, Cambridge University Press, 2010

[10] Mark C. Chu-carroll, The website Science blog http://scienceblogs.com/goodmath/2007/10/computing_strongly_connected_c.php retrieved on 03-2012

[11] Jgraph website, http://www.jgraph.com/ retrieved on 03-2012

[12] JUNG (Java Universal Network / Graph Framework) website, http://jung.sourceforge.net/ retrieved on 03-2012

Dr. Saleh Alshomrani is a faculty of Information Systems Department at King Abdulaziz University. He is also serving now as the Vice-Dean of Faculty of Computing and Information Technology, and the Head of Computer Science Department – North Jeddah Branch at King Abdulaziz University. He earned his Bachelor degree in Computer Science (BSc) from King Abdulaziz University, Saudi Arabia 1997. He received his Master degree in Computer Science from Ohio University, USA 2001. He Also earned his Ph.D. in Computer Science from Kent State University 2008, Ohio, USA, in the field of Internet and Web-based Distributed Systems, and he is actively working in this area.

Gulraiz Iqbal is a faculty of Information Systems Department at King Abdulaziz Univeristy. He earned his Master of Software Technology from Linnaeus University Sweden (2009). His research interest is software quality, visualization and graph applications.