Global Journals LaTeX JournalKaleidoscope™
Artificial Intelligence formulated this projection for compatibility purposes from the original article published at Global Journals. However, this technology is currently in beta. *Therefore, kindly ignore odd layouts, missed formulae, text, tables, or figures.*

An Extended Experimental Evaluation of SCC (Gabow's vs Kosaraju's) based on Adjacency List

, Gulraiz Iqbal[1] and Saleh Alshomrani[2]

[1] King Abdulaziz University

Received: 9 December 2012 Accepted: 31 December 2012 Published: 15 January 2013

Abstract

We present the results of a study comparing three strongly connected components algorithms. The goal of this work is to extend the understandings and to help practitioners choose appropriate options. During experiment, we compared and analysed strongly connected components algorithm by using dynamic graph representation (adjacency list). Mainly we focused on i. Experimental Comparison of strongly connected components algorithms. ii. Experimental Analysis of a particular algorithm.Our experiments consist large set of random directed graph with N number of vertices V and edges E to compute graph performance using dynamic graph representation. We implemented strongly connected graph algorithms, tested and optimized using efficient data structure. The article presents detailed results based on significant performance, preferences between SCC algorithms and provides practical recommenddations on their use. During experimentation, we found some interesting results particularly efficiency of Cheriyan-Mehlhorn-Gabow's as it is more efficient in computing strongly connected components then Kosaraju's algorithm.

Index terms— graph algorithms, directed graph, SCC (strongly connected components), transitive closure.

1 Introduction

raphs are widely used in computer, mathematics as well in chemistry, biology and physics. Pair wise 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 (web pages are vertices and hyperlink are edges) and Molecules (vertices are atoms and edges are bond between them) all benefits from the pair wise model [5,6,16] . There are some additional examples of common graph based data.

? Traffic Networks, Locations are vertices and routes are vertices in traffic networks.
? Scientific citation Network, Papers are vertices and edges are citation between papers.
? Computer Network, PC's are vertices and network connections / devices are edges.
? Social Network sites, People are vertices and their acquaintances are edges.

Graph represent a collection of elements (Vertices or Nodes) V and connection between those elements are links known as edges E. Edges often have an associated weight and direction where edges weight might carry important data strength, importance or cost of an edge.

The sections of this paper are divided as following. The introduction section provides an overview of the relevant research in this area along with graph notation and its application. Section 2 explains the extensive literature review such as current java graph libraries available, graph representation techniques and basic graph algorithms and scc graph algorithms. In section 3, we discuss the implementation, and section 4 of the model is based on our experiments. Finally section 5 and 6 presents conclusions and some important future directions respectively.

2 a) 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 ? V × V is a set of directed graph. If (v, u) ? 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 ofedges are unimportant (v, u) ? E â??" (u, v) ? 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 [6] .

3 II.

4 Literature Review

The first task is to design and develop a flexible graph library such that the graph algorithm can be implemented and tested and their performance is analyzed using the library benchmark. Many graph libraries are available in java as well in other languages. Most of the java libraries use sequential approaches which are slower over large graphs. In [3] Kurt, Stefan, and Peter mention optimization technique. We have also adopted their technique and compared our results. Later on, we will compare our algorithm with other libraries to make it computationally fast.

? Annas, is an open source Java framework suitable for developers and researchers in the field of graph theory, graph structure, algorithms and distributed systems. It has many features such as support for directed & undirected graphs, multi graph, fully(D D D D D D D D)

generic and has capability to export DOT, XML and adjacency matrix files [13].

? Jung, The java universal network / graph framework is an open source library which provides extensive modeling, analysis and visualization tool for the graph or network. JUNG architecture has 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 graph theory, data mining, social network, optimization and random graph generator **??**12] .

? 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 are implemented using structured approach [11] .

? JDSL is an open source data structure library in java using structured approach. It's a collection of java interfaces and classes that implement fundamental data structure. Advance and complex graph algorithms are not available in JDSL library. One of the powerful and safe operations on internal data structure representation is accessors [17] .

During our work we used the existing libraries to implement different strongly connected components algorithms.

5 a) Graph Representation

There are many possible ways to represent a graph in computer program but according to Mark.C.Chu-Carroll, there are two standard techniques to represent graphs in computer.

i. Adjacency Matrix / Matrix base Representation An adjacency matrix is N×N matrix of 0/1 values, where a vertex V i,j is 1 only if there is an edge between V i and V j otherwise it is 0. If Graph is undirected then the matrix is symmetric V i,j = V j,i . In case of directed graph then V i,j =1 means that there is an edge from V i to V j [10] . Adjacency matrix is useful to add an edge. It requires O(1) time which is equal to the time for the verification of an edge between two vertices but an extra computational effort is required. Adjacency matrix required extra memory to store large graphs. Few algorithms require knowledge of their adjacent vertices which results O (|V|) complexity [10,16] .

ii. Adjacency list / List based representations An alternative representation for a graph G (V, E) is based on adjacency list. For each vertex we keep a list of all the vertices adjacent to the current vertex. We say that vertex V i is adjacent to vertex V j if (V i , V j) ? E. It requires less memory and in some particular situations it outperforms adjacency matrix such as it gets the list of adjacent vertices with in O (1). In our experiments we use adjacency list with a few improvements to avoid iterative procedure. In our implementation we maintain a list of all nodes adjacent to the current node. The time complexity for adjacency list is O (n+m) [10,16] .

The adjacency matrix is more effective when edges don't have data associated with them. In case of sparse graph adjacency matrix performance is poor and huge amount of memory is wasted. Adjacency 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 selected adjacency list representation for our experiments [10] .

6 b) Strongly Connected Components

Let G = (V, E) be a directed graph, where C is a strongly connected components (SCC) of V. C is strongly connected if a maximal set of vertices after every two vertices (u, v) ? C are mutually reachable.

There is a path from vertex u to v and v to u or if a sub graph is connected in a way that there is a path from each node to all other nodes. If a graph has the same property, then the graph is strongly connected **??**6. 16] .

Strongly connected components can be computed using different approaches as introduced by Tarjan's, Gabow and Kosaraju's. Tarjan's and Gabow algorithm require only one DFS, whereas Kosaraju's algorithm requires two DFS. In this paper we included Kosaraju's algorithm. The asymptotic analysis of such algorithm on dynamic graph representation algorithm is O(|v|+|E|) and O (|V| 2) on adjacency matrix based implementation. As our implementation is based on adjacency list, it will take linear time to compute SCC which is similar to Tarjan's and Gabow's algorithm on dynamic graph representation.

Our previous experiments indicate that Tarjan's algorithm is slower than Gabow's algorithm [16] .

7 c) 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 it reaches the target matrix or to the final matrix which has no further child. Then, back tracking is used to return the last vertex which is not yet completely explored. Modifying the post-visit and previsit, DFS is used to solve many important problems and it takes O (|V|+|E|) steps. i. Pseudo-code: DFS 1. DFS (v): visits all the vertices reachable from v in depth-first order. 2.

Mark v as visited 3.

for each edge v ?u Year (depth first search), initially with an empty stack of vertices V and pushing vertices onto the stack as recursion which started from vertices V and after completion of traversal vertices V will be available in the stack. To obtain reverse graph, all the edges of graph are reversed. It starts with the top vertex on the stack and traverses from that vertex. All vertices are reachable from that vertex such that it forms strongly connected components. By removing SCC from the stack and repeating the process with the new obtained top of the stack, stack will be empty and a list of SCC is collected.

i. Pseudo-code: SCC Input : DAG G= (V, E) Output : Set of strongly connected components Let S be an empty stack While S does not contain all vertices Choose an arbitrary vertex v not in S Start DFS (V) Push (u) on S Reverse the direction of all edges to obtain transpose graph.

For vertex v with label n?.1 and find all reachable vertices from v and group them as an SCC.

8 e) Cheriyan-Mehlhorn-Gabow Algorithms

Gabow strongly connected component is also similar to Kosaraju's algorithm. It accepts a directed graph as an input and result contains a 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 which are not yet computed as strongly connected components and other contains a set of nodes that do not belong to various strongly connected components. A counter is used to count number of visited nodes, which is used to compute preorder of the nodes **??**2, 3, and 4].

9 Implementation

In our implementation we used only dynamic graph data structure that used linked lists for the adjacency list. The graph generator class makes sure that each vertex is stored in consecutive location in the adjacency list, as a fact dynamic implementation consumes more space then static graph data structure. The graph structure package contains interfaces and abstract classes to provide interface to different types of graphs such as Directed Graph. All classes mentioned in our method are Generic and user can use them by their own style. Graph package also contains many interfaces for different graphs and interfaces for the different algorithms describing that describe prerequisite method for the algorithms. The undirected graph is not currently used in our method, but it can be considered in future.

We have used a lot of interfaces and abstract classes which helps in implementation of the graph classes. The directed graph interface defines many methods such that each node represents a unique data member of generic type and two nodes can't be added to graph if they representing the same node. The second attempt will be ignored and also multiple edges between two nodes are not allowed. An abstract Node<E> class node that also serves as an interface for the vertex of DirectedGraph<E> interface, each node maintains a list of its successors and predecessors. Abstract Node<E> class also defines a set of protected methods that can be used to add and remove adjacent nodes. They should only be used by implementers of the DirectedGraph<E> interface. A public integer data member num is introduced to avoid externally constructed mappings between the node and some integer (e.g. a dfs GraphGenerator class implementing the interface of directed graph is specially designed for testing and benchmarking. Initializing the graph generator class by providing an instance of a class implementing the directed graph interface, all graphs generated are the instances of that class. This class is used to generate random, acyclic, dense, sparse and complete graphs. number). It should only be used internally and never be a part of any public interface since its interpretation might be changed from one algorithm to another.

10 IV.

11 Experiments

In our experiments we used GraphGenerator class to generate sparse and dense graph. Graph with minimal edges E=100 considered as a sparse graph and graph with maximum edges E = 500 is a dense graph. We designed benchmark which generates six graphs of same size as input and measure the run time computing strongly connected components of given graphs; we computed average time to obtain the(D D D D D D D D)

Year performance of specific algorithm on a specific number of nodes and edges. We also calculated standard deviation that indicates upper bounds and lower bounds to visualize the variations and outliers in the data set using error bars on chart. Analysis of random graphs is also not easy because they contain random nodes, edges and dynamic memory.

In our experiments we used dynamic graph data structure using linked list for the adjacency list. We use intel® Core? i5-2410M CPU @2.30GHz with 4 GB of memory for computing our algorithms.

We have used eclipse version Helios Service Release 2 as IDE for java developers in our experiments. We increased the heap size by providing the argument -Xms128m -Xmx1550m -XX: +UseParallelGC.

For recursive calls stack size is also important. In some scenarios such as on a large number of vertices and edges, stack over flow error occurs.

12 a) Experiments on Kosaraju's Algorithm

In these experiments, a set of random graph for each graph (Dense and Sparse) with minimum edges E=100 for sparse graph and maximum edges E=500 for dense graph is generated. Figure 1 shows the running time difference between dense and sparse graph on N number of nodes.

Kosaraju's algorithm compute strongly connected components efficiently with increase in number of nodes or increase in number of edges. So edges have a direct impact on its running time.

13 i. Average Computation Time

Figure 1 presents the results generated by one benchmark methods. It is clear from the figures that with increase in the number of nodes and edges, Kosaraju's strongly connected components algorithm takes more time to run.

14 b) Experiments on Gabow's Algorithm

We had the same set of experiments for Gabow's algorithm, for each graph (Dense and Sparse). We generated six random graph with minimum edges E=100 for sparse graph and maximum edges E=500 for dense graph.

We computed their average completion time and memory storage as the Figure **??** & 4 show the difference between dense and sparse graph on N number of nodes. Gabow's algorithm compute strongly connected components efficiently when numbers of edges are lower. So edges have a direct impact on its running time and memory.

15 i. Average Computation Time

In Figure 4, line chart is used to present the results generated by our benchmark which show that with increase in the number of nodes and edges Gabow's SCC algorithm takes more time to run. The same data is used to compute average run time for each node. Also data is combined to get a unique data that is used to compare Kosaraju's and Gabow's algorithms. In Figure 5 & 6 average completion time is computed on sparse graph (E=100) and dense graph (E=500) for both Kosaraju's and Gabow's algorithm. Performance of both algorithms is remarkable; as Gabow's algorithm take less completion time and variation then Kosaraju's algorithm. Kosaraju's algorithm is simple in implementation. for both Kosaraju's and Gabow's algorithms but their runtime is different.

V.

16 Conclusions

In our research, we analyzed & compared Kosaraju's and Gabow's strongly connected component algorithms to find their suitability for various applications. We produced dense and sparse graphs randomly to compute memory difference of the both the algorithms. We found that Gabow algorithm is shorter, simpler and more elegant. Kosaraju's algorithm takes more time then to Gabow's algorithm on both dense and sparse graph. There are some limitations in our experiments. In a limited data set, we produced six graphs with N=3900, using sparse graph E=100 and dense E=500 to compute average run time memory and average completion time. In future we will develop a large graph with increase in the stack size and java VM heap size.

In this research, we have focused on Kosaraju's and Gabow's algorithms only and data structure used is

Figure 1:

adjacency list. In future, we would implement Brute's algorithm to compute strongly connected components using a hybrid algorithm and as well involving other data structures for graph. [1] [2] [3]

[1] EAn Extended Experimental Evaluation of SCC (Gabow's vs Kosaraju's) based on Adjacency List

[2] Egraph G as an input and performs a recursive DFS An Extended Experimental Evaluation of SCC (Gabow's vs Kosaraju's) based on Adjacency List

[3] An Extended Experimental Evaluation of SCC (Gabow's vs Kosaraju's) based on Adjacency List

[Website] , Jgraph Website . http://www.jgraph.com/ p. .

[Annas Website] , Annas Website . https://sites.google.com/site/annasproject/retrievedon09-2012

[Holten et al. ()] 'An Extended Evaluation of the Readability of Tapered, Animated, and Textured Directed-Edge Representations in Node-Link Graphs'. Danny Holten , Petra Isenberg , Jarke J Van Wijk , Jean-Daniel Fekete . *Pacific Visualization Symposium*, (Pacific Vis) 2011. IEEE.

[Saleh Alshomrani and Iqbal (2012)] *Analysis of Strongly Connected Components (SCC) Using Dynamic Graph Representation, IJCSI*, Gulraiz Saleh Alshomrani , Iqbal . July 2012. 9.

[Shirinivas et al. ()] 'Application of graph theory in computer science an overview'. S G Shirinivas , S Vetrivel , Dr N M Elango . *International Journal of Engineering Science and Technology* 2010. 2 (9) p. .

[Barnat et al.] 'Computing strongly connected components in parallel on CUDA'. Jiri Barnat , Petr Bauch , Lubos Brim , Milan Ceska . *IEEE 2011 International Parallel & Distributed Processing Symposium*,

[Hopcroft and Kosaraju ()] 'Dividing a graph into triconnected compoents'. J E Hopcroft , R E Kosaraju . *SIAM Journal on Computing* 1973. 2 (3) p. .

[Mehlhorn et al. ()] *Engineering DFS based Graph Algorithms, Partially supported by DFG grant SA 933/3-1*, Kurt Mehlhorn , Stefan Naher , Peter Sanders . 2007.

[Barnat and Chaloupka ()] 'Jaco van de Pol, Distributed algorithms for SCC decomposition'. Jiri Barnat , Jakub Chaloupka . *Journal of Logic and Computation* 2011. 21 (1) p. .

[Mark and Chu-Carroll] C Mark , Chu-Carroll . http://scienceblogs.com/goodmath/2007/10/computing_strongly_connected_c *The website Science blog*, p. .

[Gabow ()] 'Path-based depth first search strong and biconnected components'. H N Gabow . *Information Processing Letters* 2000. 74 (3-4) p. .

[Easley and Kleinberg ()] *Reasoning about a highly connected world, Textbook*, David Easley , Jon Kleinberg . 2010. Cambridge University Press.

[Skeina] Steven Skeina . http://www.cs.sunysb.edu/~algorith/implement/jdsl/implement.shtml *The Stony brook algorithm Repository*, p. .

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

[Steinhaus (2008)] *The text book, Comparisons of mathematical programs for data Analysis*, Stefan Steinhaus . July 2008.

[Marije De (2011)] *Towards a Library of Parallel Graph Algorithm in Java, 14 th Twente Student conference on*, Heus Marije De . January 21 st 2011.