Pre-processing and Path Normalization of a Web Graph used as a Social Network

C.Bhanu Teja[1], S.Mitra[2], A.Bagchi[3], A.K.Bandyopadhyay[4]
[1]Tata Consultancy Services, Hyderabad
India
bhanu_teja@gmail.com
[2]Meghnad Saha Institute of Technology, Kolkata
India
susanta_mitra@yahoo.com
[3]Indian Statistical Institute, Kolkata
India
aditya@isical.ac.in
[4]Electronics & Tele-Comm. Engineering Dept. Jadavpur University, Kolkata
India
anupbandyopadhyay@hotmail.com

Journal of Digital
Information Management

ABSTRACT: *Many efforts have been made for compression of web graphs. Most of these methods are suitable for search engines and are centered around encoding links and URLs efficiently. The purpose is to handle a large set of web pages in the main memory against any web based search. The authors of the present paper are interested in studying web graph as a social network and to develop a data model for it. So, suitable compression techniques, which are space efficient for disk based storage, are required. This paper has provided a two level compression technique. In the first level, the structural properties of a graph are studied and strongly connected components are fused to reduce the original graph to a DAG. Paths on this DAG are then stored efficiently using a Path Normalization technique. Space complexity expressions indicate the efficiency of the method. Relevant operators required for accessing the original graph through the compressed representations have also been discussed. Important earlier works have been referred to indicate the requirements of the present approach.*

Categories and Subject Descriptors
E.1 [Data Structures]; Graphs and networks **E.4 [Coding and Information Theory]** ; Data compaction and compression] **H3.4 [Systems and software**]; Information networks

General Terms
Web graphs, Social networking, Data compression

Keywords: Social software, Web search, Path normailisation technique, Compression technique.

Received 7 August 2006; Revised 11 March 2007; Accepted 12 April 2007

1. Introduction

A Social Network is the graph representation of a social community. Here each member of a social community (people or other entities embedded in a social context) is considered as a node and communication (collaboration, interaction etc.) from one member of the community to another member is represented by a directed edge, forming a directed graph or digraph. A social network represents a network of acquaintances between people like, a club and its members, a city or village community, a research group communicating over Internet, a group of people communicating with each other through e-mail messages for a specific purpose etc. Recently, Web has played a major role in the formation of communities (*cyber-communities*) where the members are people from different parts of the globe who join the community for some common interest. The number of communities in the Web is increasing dramatically with time. This community formation is one of the most powerful socializing aspects of the Web and hence Web is also a social network. Out of the many social network models on the Web, the most commonly used one is called a referral system. In such a system, each node in the social network provides a set of links to its acquaintances that in turn become member nodes of the network. In the same way, these new nodes bring their acquaintances to the network again. This way the social network keeps on growing. So, the social network on the Web gives rise to an evolutionary graph. At any instant of time, when a query is raised on such an evolutionary graph, a snapshot of the concerned network, i.e. the node-edge structure at the instant of query, is considered for the purpose of query processing. There are many commercial products like, LinkedIn.com, Ryze.com, Tribe.net [19][20][21] etc. that tend to generate a social network. These Web sites invite people to introduce themselves, and then add their colleagues, or business partners. In this sense, these sites define a social network similar to a referral system. Based on the input of the existing users, new members are taken in the network provided they meet its business requirements. For example, a site that provides matrimonial services, would only consider accepting a new member provided he/she is searching for a suitable match. These commercial products provide different types of services like, employment service, matrimonial service, e-learning service etc. There are even academic and research efforts [32][34] that analyze such referral systems for understanding the social properties and behaviors of the underlying social communities on the Web. A social network on the Web may have thousands of nodes and edges.

Salient features of a web-graph used as a social network are:

- In a social network graph, representing a real social community, an edge (i,j) signifies that node i contacts node j. However, node j will contact node i, only if edge (j,i) is also present in the graph. So, node i contacts node j does not necessarily imply that node j also contacts node i. A two way edge is usually called a reciprocal connection.
- Edges between the nodes may be of different types. While node n_1, i.e. a member of a social community, is contacting another node n_2 for economic reason, node n_3 in the same network may contact node n_4 for academic

reason. An edge-type between two nodes in a social network signifies the type of relationship that exists between two real life entities present in the corresponding social network.

- Different node combinations in a social network may form cycles. A reciprocal connection is the smallest size cycle. There can also be nested cycles. In a cycle or in a nested cycle structure, all edges may or may not be of same type. There can also be strongly connected components in a network, where each node is reachable from any other node within that structure. Size and types of relationships present in a cycle or a nested cycle or a strongly connected component structure provide important information to the social scientists regarding the properties of the concerned social community.

Besides the cycles and nested cycle structures of a social network, social scientists are also interested in other structural properties of a network. For a given social network, a social scientist usually makes query about average in-degree and out-degree of nodes, maximum in-degree and out-degree of nodes, reachability of one node from another, i.e. searching of paths, finding all ancestors or descendents of a node, finding common ancestors or descendents of two or more nodes, finding nearest common ancestor or descendent of two or more nodes etc. [14][15][23][24][28][29][34].

Other than the structural information, depending on the application area, a social network will have node-based information as well. For example, a social network on the Web may offer employment services, where nodes provide information like *qualification, experience* etc. Similarly, another network may offer matrimonial services, where nodes provide information like, *age, marital-status, sex, monthly earning* etc.

1.1 Motivation

Discussions made so far indicate that social scientists make rigorous computation on the node-based and structural information of a graph representing a social network. Each such computation has to access the entire graph related (both node-based and structural) data. Since a social network on the Web may give rise to a graph of thousands of nodes and edges, accessing the entire graph each time contributes significantly to the overall computation time. Moreover, some applications try to search for interesting patterns on the existing data, both node-based and structural [10]. Such social network related applications are quite common in web-based mining [9]. Overall computation time can be reduced to a great extent if the structure-based and node-based selection and searching can be done efficiently. In order to make it effective, the relevant information for both nodes and edges along with common built in structures like cycles, nested cycles, paths etc. may be computed and stored apriori. If any application needs a particular type of computation quite often, such information can also be pre-computed and stored. In short, instead of starting from raw node and edge related data for each type of analysis, some storage and selective retrieval facility should be provided for social network applications involving large graphs. So, a data model needs to be designed primarily for social network applications.

An object-relational data model named SONSYS (Social Network System) has already been proposed for this purpose [25][4]. The SONSYS data model is based on two sub-systems, *Structure-based* sub-system and *Node-based* sub-system. *Structure-based* sub-system processes a Web graph where a user can make queries on the social networkstructure. *Node-based* sub-system, on the other hand, is used for querying on node properties. So a composite query madeon a social network, can be answered partly by its structure and partly by its nodes. SONSYS data model supports the following types of queries on a social network :

- Query on node based information only.
- Graph pattern matching on the communication structure of the social network.
- Composite queries exploiting the above facilities.

The *Structure-based* sub-system discusses about different structural components present in a Web-based social network. Starting from a digraph representing a social community, different steps of preprocessing compresses the original digraph to a DAG. During the process of compression, different hyper-structures are identified. Structure-based sub-system of the generic data model used for modeling a social network needs to consider the storage and retrieval of these hyper-structures. The operators needed for processing queries on these hyper-structures and the compressed DAG, are also considered in designing this sub-system.

The objective of the node-based sub-system is to design a node-based schema depending on the application area under consideration. Since SONSYS model has been developed against an object-relational framework, node-based schema provides a set of underlying relations that store the node and edge properties of the Web graph used in the application under consideration. Properties of Web pages like id or URL, title, size, last date of modification, text, etc. can be stored as node attributes.

Query processing in SONSYS data model considers both structure-based and node-based schemas in answering a query. Depending on a query, only one of the two schemas or both may take part in query processing. Indexing strategies for query processing have also been discussed in the data model.

As mentioned earlier, a graph representing a social network can have thousands of nodes and edges. Moreover it may have cycles, nested cycles and strongly connected components. The present paper has referred these structures as hyper-structures and treated them separately. For efficient storage and retrieval of the structure related components and thus to improve the performance of query processing, this paper proposes a two level compression technique.

- First level involves fusion of each hyper-structure to generate a single node called hyper-node. This process may give rise to another hyper-structure called hyper-edge. This fusion process converts the original digraph ·representing a social network to a directed acyclic graph (DAG). Section 2 discusses this fusion process in detail. Correctness and justification of fusion process have also been established in the same section.
- Second level involves efficient storage and retrieval of path related information. After the fusion process, any query on the original social network is done on the fused graph or the DAG. As discussed earlier, queries on a social network can be divided into two main groups. One that involves the structure and size of different hyper-structures and the other is basically on the node-edge connectivity or path-based query.

For path based queries, any of the two approaches may be taken:

- creation of paths from the edges against query,
- generation and storage of all paths apriori and selective retrieval of paths against query.

However even after pre-processing, a DAG representing a social network on the Web may have thousands of nodes and edges. So in the first approach, computation of paths against queries will make query processing very slow. On the other hand, in the second approach, storage of all pre-computed simple paths of different lengths and then to index them and access them against queries may not make the query processing more efficient. This would also need too much of space. So, a trade-off is necessary, where a few paths and sub-paths will be pre-computed and stored and all the simple paths of the DAG should be computable from them. This process has been called as *Path Normalization.* The present paper describes the normalization process in detail and proves it to be minimal and complete. In Section 3, normalization process has been discussed in detail.

Main advantages that can be derived from the present work are:

- Since the hyper-structures (hyper-nodes and hyper-edges) generated during fusion process are separately stored and indexed, any hyper-structure based query can be processed directly without accessing the compressed DAG.
- After path normalization, depending on query and the corresponding search, existing paths in the normalized path-set may suffice to answer a query. So, no new path may need to be generated for such a query.
- In case a query needs generation of new paths besides the normalized path-set, only minimum number of paths necessary over the existing normalized path-set are generated. This paper furnishes the proof for minimality of such path generation.

After providing the introduction and motivation in Section 1, Hyper-node based compression method has been discussed in Section 2. The compression converts a digraph to a DAG. Section 3 applies the Path Normalization method on the compressed DAG. Section 4 provides the relevant operators needed for structural queries. Section 5 discusses about the earlier research efforts to justify the utility of the proposed approach. Section 6 ultimately draws the conclusion indicating the future efforts needed.

2. Hyper-node based compression

Considering a Web graph representing a social network, this section identifies the different hyper-structures present in such graph. Any strongly-connected-component (SCC) is fused to form a hyper-node. A similar fusion process has been suggested in some cases, a set of edges can also be fused to form a hyper-edge. This section defines all these hyper-structures and then proposes a fusion algorithm to modify the original graph. It has been shown that the fusion process converts the original Web graph to a DAG. Lemma 1, given in this section, shows that the proposed fusion process really generates a DAG. Then Lemma 2 shows that as a result of the fusion process, the original Web graph remains connectivity invariant even after the formation of DAG.

Fusion process definitely alters the paths present in the original graph. Corresponding advantages and limitations on the processing of path based queries have also been discussed in this section.

In order to understand the methods of pre-processing and compression, a sample social network has been considered. The network is shown in Figure 1.

Although a few nodes and edges have been considered here, the explanation will soon show that even this small graph covers all the structural peculiarities of a social network on the Web. As shown in Figure 1, the network initially had 4 nodes (1,2,3,4). Node 5 is the acquaintance of node 4. So, node 5 joined the net.

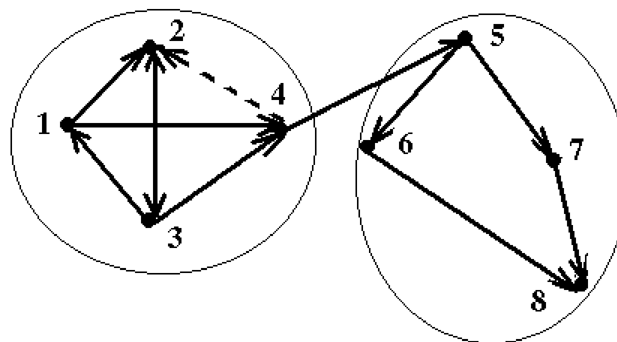


Figure 1. Sample Social Network

the net. In turn node 5 brought nodes 6 and 7 and they again brought node 8 in the network. This way the network keeps on growing. It is assumed that at the time of query, Figure 1 shows the current status of the network. This network consists of the following structural components:

- *Strongly-Connected-Component (SCC) :* A strongly-connected-component is a maximal subgraph of a directed graph such that for every pair of nodes v_1, v_2 in the subgraph, there is a directed path from v_1 to v_2 and also a directed path from v_2 to v_1.

If there exists an operator $R(v_1, v_2)$, such that $R(v_1, v_2)$ = True if node v_2 is reachable from node v_1 (i.e. there exists a path from node v_1 to node v_2), then subgraph G'(V',E') of graph G(V,E) is a SCC, if $R(v_1, v_2)$ = True and also $R(v_2, v_1)$ = True, where $(v_1, v_2) \in V'$.

This definition indicates that a reachability operator *R* will be required, in order to check the existence of paths between any two nodes of a graph. Detail discussion in this regard will be made later.

The sample social network in Figure 1 has two edge-types shown by farm and chain lines. Edge (4, 2), represented by a chain line is of different edge-type, while all other edges represented by farm lines are of same edge-type in Figure 1. Node sequence (1-2-3) represents a strongly connected component when same edge-types are considered, whereas (1-2-3-4) is a SCC considering both the edge-types.

- *Cycle:* If the sequence of nodes defining a path of a graph, starts and ends at the same node and includes other nodes at most once, then that path is a cycle. If in a graph G(V,E), $(v_0, v_1, \ldots, v_n)$ be a node sequence defining a path P in G such that $(v_0, v_1, \ldots, v_n) \in V$ and $v_0 = v_n$, then P is a cycle.

Figure 1 shows three cycles; (1-2-3-1), (2-3-4-2) and (2-3-2). Here cycles have been considered irrespective of the variation in edge-types. The cycles may even be nested. Cycle (2-3-2) is nested within the other two cycles, (1-2-3-1) and (2-3-4-2).

- *Reciprocal Edge:* A cycle having only two nodes is a reciprocal edge. So, a reciprocal edge $(í_1, í_2) \in V$ has directed edge from $í_1$ to $í_2$ and also from $í_2$ to $í_1$. A reciprocal edge is the smallest size cycle.

In Figure 1 (2-3-2) is a reciprocal edge.

- *Hyper-node*: In a nested-cycle structure, the largest or the outermost cycle is defined as a hyper-node. For a graph G(V,E), if there exists a nested cycle structure with a set of cycles such that, $\{C_1 \supseteq C_2 \supseteq \ldots \supseteq C_n)\}$ where C_i is a cycle in G, then C_1 is the hyper-node corresponding to the nested cycle structure. So, a hyper-node represents a SCC.

- *Homogeneous Hyper-node:* If in a hyper-node all the edge-types are same, then it is a homogeneous hyper-node. Let, $\{C_1 \supseteq C_2 \supseteq \ldots \supseteq C_n\}$ be a nested cycle structure in a graph G(V,E) where, C_i is a cycle in G. Now C_1 will be a homogeneous hyper-node if for any pair of edges, $(\acute{\imath}_i, \acute{\imath}_j) \in C_1$ and $(\acute{\imath}_r, \acute{\imath}_s) \in C_1$, $(\acute{\imath}_i, \acute{\imath}_j)$.edge-type = $(\acute{\imath}_r, \acute{\imath}_s)$.edge-type.

In Figure 1 (1-2-3) is a homogeneous hyper-node.

- *Heterogeneous Hyper-node:* In a heterogeneous hyper-node all the edge-types need not be same. In Figure 1, (1-2-3-4) is a heterogeneous hyper-node.

Though by definition, a hyper-node is the largest cycle in a nested cycle structure, the hyper-nodes themselves can also be nested. Since in a homogeneous hyper-node all the edges must be of same type, this hyper-node may be nested within another larger cycle formed by edges of different types resulting a heterogeneous hyper-node. So, a homogeneous hyper-node may be nested within a heterogeneous hyper-node. In Figure 1, homogeneous hyper-node (1-2-3) is nested within heterogeneous hyper-node (1-2-3-4).

2.1 Homogeneous hyper-node based compression

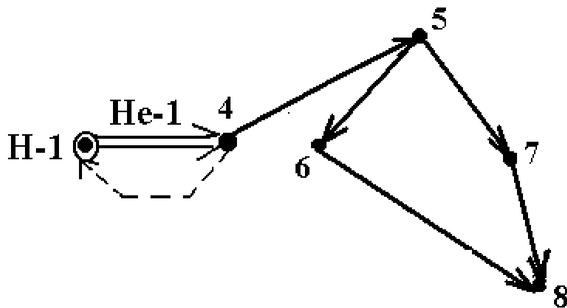


Figure 2. Augmented Social Network

For compression, a hyper-node is fused to a single node and its structural details are stored separately covering both the nodes and edges within it. As a matter of fact, SONSYS system defines a object-relational framework where these hyper-structures are treated as different object types. Following the above principle of compression the sample social net in Figure 1 is augmented to Figure 2 considering homogeneous hyper-nodes only.

Nodes within a hyper-node H may be connected to other nodes and hyper-nodes outside H. After H is fused to a single node as part of graph compression process, all these edges external to H but connected to its different nodes will now be connected directly to H. In homogeneous hyper-node (1-2-3) of Figure 1, nodes 1, 2 and 3 are connected to the external node 4 by edges (1,4), (4,2) and (3,4). After fusion, in Figure 2, these three edges are (H-1,4), (4,H-1) and (H-1,4). Now, the edges (1,4) and (3,4) in the original graph are of same edge-type and they are both mapped as edges (H-1,4) and (H-1,4) in the augmented graph. The compression process will fuse these two identical edges of same type and direction to a single edge He-1 defined as hyper-edge. So, the hyper-edge He-1 covers the edge-set {(1,4), (3,4)} of the original graph. Similar to hyper-node, hyper-edge is also treated as a separate object data type in the object-relational schema of SONSYS.

- *Hyper-edge:* If any node p outside a hyper-node H is connected to more than one node belonging to H with same edge-type and in the same direction, all such edges will be fused to only one edge as a hyper-edge. This hyper-edge will now connect p to H. A hyper-edge may connect a hyper-node with a node or another hyper-node.

2.2 Heterogeneous hyper-node based compression

The compression process so far, has considered fusion of homogeneous hyper-nodes only. However in Figure 2, further compression is possible if cycles formed by different edge-types are also considered and such cycles are also fused to form heterogeneous hyper-nodes. After such compression, the final augmented form of the original graph is shown in Figure 3.

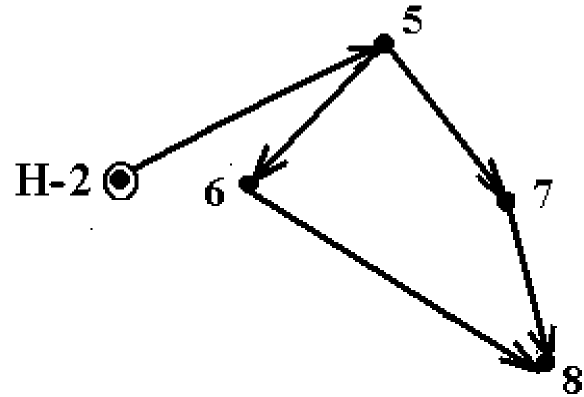


Figure 3. Final Augumentation

As can be seen in Figure 2, homogeneous hyper-node H-1 and node 4 formed a cycle with hyper-edge He-1 and another edge (4, H-1) of different edge-type. So, fusion process creates heterogeneous hyper-node H-2 as shown in Figure 3.

It is found that after the final augmentation, the original graph turns into a directed acyclic graph (DAG).

Lemma 1: *Hyper-node and hyper-edge based compression turns a digraph to a DAG.*

Proof :

Since graph isomorphism is a well known NP-complete problem, instead of showing the original graph (Figure 1) and the augmented graph (Figure 3) to be isomorphic, an effort has been made to prove the lemma by contradiction.

i. Since a graph representing a social network may have nested cycles and by definition, a hyper-node is generated by fusing the largest cycle in a nested cycle structure, the cycle detection and fusion algorithm recursively fuses the cycles from the innermost (smallest) to the outermost (largest) one.

ii. If any stage of compression retains a cycle in the graph, the next pass of the recursive process detects it and fuses it. So, at the end of the fusion process, the graph would not have any cycle.

iii. At each stage of cycle detection and fusion algorithm, any edge coming from any node outside a cycle structure and incident to any member node of that cycle, will be incident to the fused node after fusion. So, connections coming from outside a cycle structure are retained in the augmented graph after fusion.

iv. At each stage of cycle detection and fusion algorithm, any edge leaving any member node of a cycle structure connecting any node outside the cycle, will also be leaving from the fused node after fusion. So, outgoing connections from member

nodes of a cycle are retained after fusion.

v. Since hyper-node is the largest cycle in a nested cycle structure, therefore, according to Step iii and iv, the fused node retains all connections outside its structure after fusion.

vi. Step v leads to a situation where a hyper-node may be connected to a node or another hyper-node by more than one edge of same type and direction. From definition it follows that these edges are fused to hyper-edge causing the original digraph to become a DAG.

Lemma 2: *Conversion to DAG from original digraph is connectivity invariant.*

Proof :

The compression process while fusing the hyper-nodes apparently alters the node-edge connectivity structure present in the original graph. More than one edge between two nodes/ hyper-nodes may fuse to a hyper-edge. An edge between two nodes in the original graph may turn into an edge between a node and a hyper-node or between two hyper-nodes after compression. However, any structural query though placed on the compressed DAG should be able to explore through the hyper-structures to dig out the original connectivity and answer the query accordingly. So it is necessary to show that the compression technique leaves the graph connectivity invariant even after fusion and generation of DAG.

i. Since, hyper-node and hyper-edge structures are separately retained as structural object data type in the object-relational schema, the nodes, edges and their interconnections encapsulated within such structures are retained in the individual object instances.

ii. Step v of the proof of Lemma 1 establishes that after fusion, a hyper-node retains its connections (incoming and outgoing edges) with the rest of the graph structure outside its own structure.

iii. Steps iii and iv of Lemma 1 also show that the connectivity structure present in the original digraph before fusion is retained through each stage of the fusion process till the final augmentation to DAG, making it connectivity invariant.

2.3 Effect of hyper-node based compression on paths

Salient features of hyper-node based compression process are:

- Compressed graph is a DAG and a path detection algorithm will not encounter any cycle.
- Hyper-nodes and hyper-edges are stored as separate object instances under proper object types in an object-relational schema, and therefore, their internal structures are not visible to the compressed graph.
- As a result of the compression process, a path in the compressed graph will be formed by simple nodes and/ or hyper-nodes and simple edges and/or hyper-edges.
- Structure and length of paths will change when transformed from original to compressed graph. However, a path query should return the actual sequence of nodes belonging to the original graph before compression.

Now, this compression process offers some advantages as well as some limitations in the enumeration of paths.

Advantage:

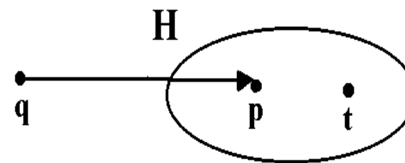


Figure 4. External connection from Hyper-Node

As shown in Figure 4, let p be a node in a strongly-connected-component (SCC) H having N nodes within it. Node p is connected to a node q outside the SCC. It is the only connection from any node within the SCC to q. Now, if a path detection algorithm searches for paths between node q and any arbitrary node t, within the same SCC, it may enumerate to a maximum of (N-1)! paths. It happens just because of the SCC structure, since all nodes within it are reachable from each other.

Now, at the time of compression, the SCC is fused to a single node to form the hyper-node H. As a result, the edge (p,q) from internal node p to external node q with respect to H maps to an edge (H,q) in the compressed graph. So for any node t within H, the system will enumerate only one path (H,q) from H to q instead of a maximum of (N-1)! paths. In turn, if the system wishes to pre-compute and store the paths, it will have to store only one path instead of a maximum of (N-1)! paths. The detail of the hyper-structure stored in the underlying object-relational system will store the detail (node and edge components) of the hyper-node H. However, this advantage in storage may cause a limitation in the enumeration of paths in the original graph against a path query.

Limitation: A path detection algorithm, when executed on a graph, returns paths that are either pre-computed or generated at run time containing simple nodes and edges. However, when such algorithm is executed on the compressed graph as proposed in this paper, the paths may have hyper-nodes and hyper-edges within it. So, the path detection algorithm needs additional steps to break the encapsulation of those structures (hyper-nodes and hyper-edges) to get the paths involving simple nodes and edges belonging to the original graph. It is, therefore, apparent that execution time is more for path detection in the compressed graph. As shown in Figure 4, if a query needs to find all possible paths from node t to node q, the system would first find the edge (H,q) and then would enumerate within the hyper-node all possible paths from node t to node p, which can be (N-1)!, where N is the number of nodes in the hyper-node. If the path detection algorithm is executed on the original graph, it can avoid the extra step of enumerating the edge (H,q). So, if number of paths between hyper-node H and any external node q is more than one, there will be proportionate increase in the total cost for path enumeration.

3. Path Normalization

Since a Web graph representing a social network may contain thousands of nodes and edges, computation of paths against queries may make the query processing very slow. Moreover, Section 2.3 has shown that hyper-node based compression, though space efficient, may take even higher time for path enumeration.

The alternative approach will be to pre-compute and store all simple paths. However for a large graph, storage of all pre-computed simple paths of different lengths and then to index them and access them against queries may not make the query processing efficient. This would also need too much of space. So in order to alleviate such problem, a trade-off is necessary. Here, a few paths and sub-paths will be pre-computed and stored and all the simple paths of the DAG should be computable from them. This process has been termed as Path Normalization [6].

This section describes the path normalization process in detail and proves it to be minimal and complete. A modified DFS algorithm has been provided for this purpose. Time and space complexity of the algorithm has also been discussed. *Minimality Constraint* of the normalization process ensures that an edge appears only once in the normalized path set. Lemma 4 proves the minimality. *Completeness Constraint,* on the other hand proves that all simple paths present in the original graph can be generated from the normalized path set and no extra path gets generated either. A reconstruction algorithm to this effect has been provided in this section.

DAG generated as the outcome of the compression process, is stored as adjacency lists. The paths are computed from the adjacency lists. As discussed earlier, queries on a social network are mainly of two types; hyper-structure based query or path-based query. A path based query will be either of the type:

"Find all paths passing through nodes n_a and n_b"

or of the type:

"Is there a path between nodes n_a and n_b"

While the first query returns all possible paths between nodes n_a and n_b, second one returns a boolean result *True* or *False.*

One simple way for path enumeration is to generate the DFS (Depth First Search) tree from each node of the network. Each branch of each such tree will provide a path. Here, a path is defined as a node sequence. Once again, since the DAG is the outcome of a compression process, a node may be either a simple node or a hyper-node. However, this phenomenon is transparent to the path normalization process. Now the paths obtained from DFS trees may be stored with proper ids. They may be indexed and then retrieved directly against appropriate queries. However, many of these paths will have redundant node sequences used for storage. If any two such paths are considered,

- they may either be isolated (i.e. two paths may have mutually exclusive node sequences) or
- one may be totally covered by the other (i.e. one node sequence is the subset of the other) or
- two paths may have a common sub-path that can be obtained by the intersection of the corresponding node sequences.

The word normalization has been borrowed from relational database design theory. In relational normalization, total attribute set, called universal relation, is decomposed into smaller subsets of attributes generating separate relations and thereby arriving at a relational schema for the problem under consideration. The main purpose of such normalization process is to avoid redundancy of data satisfying the inter-attribute dependencies. Besides preservation of inter-attribute dependencies, normalization process reduces repetition of data to a great extent. However, since lossless. In other words, the decomposition process should be such that the original universal relation can be reconstructed by using all the decomposed relations. Since all queries in a database can be answered using a universal relation, main advantage of normalization is the saving of space by avoiding data redundancy.

Path normalization process also tries to achieve same type of advantages for a DAG. If all paths from all nodes are generated and stored, many redundant node sequences will be stored. Normalization process should ensure that only non-redundant paths and sub-paths are generated and stored. So, redundancy is avoided in the storage of edges. As a result of path normalization, an edge should appear only once in the normalized path-set. However, similar to relational normalization, path normalization process should also ensure lossless decomposition. In other words, all the simple paths of the original DAG should be computable from the normalized path-set. So, the path normalization process should maintain two essential properties:

Redundancy Avoidance: Path Normalization process should ensure that no edge appears more than once in the decomposition process and subsequent storage. So redundancy avoidance in path normalization process is complete. It is the *Minimality Constraint* of the normalization process. With the help of a simple heuristic rule for the decomposition process, this constraint can be satisfied.

Lossless Decomposition: Since the normalization process stores only a subset of all the possible simple paths and subpaths, it becomes essential to prove that all the simple paths of the DAG under consideration are computable and no extra path can be generated. It is the *Completeness Constraint* of the normalization process. With the help of a reconstruction procedure on the normalized path set, completeness of the decomposition can be proved. For this purpose, some additional information are also stored during the decomposition process.

As mentioned earlier, generation of all simple paths for a DAG can be done by generating DFS trees from each node. A modified DFS algorithm can be used for the generation of normalized path-set. As the first step to achieve this end, all nodes need not be considered for generation of DFS tree.

Lemma 3: *In a digraph, for each edge (n_a, n_b), DFS tree generated from node n_a, totally covers the DFS tree generated from node n_b.*

So, in a DFS tree, if n_b is a descendent of n_a, then dfs(n_b) Í dfs(n_a), where, dfs(n_i) is the DFS tree drawn with node n_i as the root.

Proof of this lemma has been omitted, since it is directly derivable from the well-known Parenthesis Theorem of depth-first search and proof of the theorem is available in all standard texts on graph algorithms [5][12].

So, for any edge (n_a, n_b), DFS tree from node n_b is a sub-tree of the DFS tree from node n_a. Hence, any path from node n_b will be a sub-path of some path from node n_a. In other words, to avoid redundancy, if the paths from node n_a are stored, paths from node n_b need not be stored. So, the first step of normalization is to choose only those nodes of the graph where from the DFS trees and thus the paths will be generated and stored. Naturally, any such node should not have an ancestor. Since the compressed graph is a DAG, the path generation process will not encounter any cycle. The different types of nodes present in a DAG are:

Definition 1. Source

If for a node n_i, $ind(n_i) = 0$, then n_i is a Source, where $ind(n_i)$ represents the in-degree of the node n_i.

So, DFS tree generated from node n_i will not be covered by any DFS tree generated from any other node.

Definition 2. Sink

If for a node n_i, $outd(n_i) = 0$, then n_i is a Sink, where $outd(n_i)$ represents the out-degree of the node n_i.

So, any path reaching node n_i cannot extend any further and n_i is a sink.

Definition 3. Anchor Node

Anchor Nodes are mainly of two types,

1. if for a node n_i, $outd(n_i) > 1$, and n_i is not a Source, then n_i is an Anchor Node,

2. if for a node n_i, $ind(n_i) > 1$, and n_i is not a Sink, then n_i is an Anchor Node,

So, for a node n_i, if both $ind(n_i) > 1$ and $outd(n_i) > 1$, then also n_i is an Anchor Node.

Anchor nodes play a very important role in the normalization process. The anchor nodes and their corresponding in-degrees or out-degrees determine the cardinality of the set of non-redundant paths and sub-paths that need to be generated during path normalization.

Using the above definitions of nodes, a Simple path is defined as:

Definition 4. SimplePath

A Simple Path is a path from a Source to a Sink. So, if n_a is a Source and n_b is a Sink, then the node sequence (n_a, ……., n_b) represents a Simple Path.

In the normalization process, DFS trees are drawn only from sources. According to Lemma 3, DFS tree from any other node will be the sub-tree of a DFS tree drawn from some source. So, paths generated from other nodes will be sub-paths of the paths generated from sources. DFS trees from all the sources, will provide all possible simple paths present in the graph. Any path from any node other than the sources will be a sub-path of a simple path already considered. So, it is sufficient to generate, store and index only the simple paths to cover the total set of paths present in the graph.

Though the simple paths provide a unique set of paths for a graph, storage of such paths is not free from repetition. Two simple paths may have considerable overlap. If both the paths are fully stored, the edges belonging to the overlapping portion will be stored more than once. Since the graphs under consideration have hundreds of nodes, storage of all simple paths may give rise to too many redundant sub-paths. Path Normalization process intends to store only non-redundant paths and sub-paths, so that an edge in the graph is stored only once. Now, if a graph has n simple paths, normalization process can make n^2 order comparisons among the paths to find the overlapping sub-paths between any two simple paths and store them only once. However, that process would be computationally prohibitive for a real-life application involving hundreds of nodes and edges. So, a better method for normalization is required.

3.1 Generation of Normalized Set of Paths

If any three nodes of a DAG are considered, they can be connected in only three ways; *Chain, Cap* and *Cup.*

The ***Chain structure,*** as shown in Figure 5, is a sequence of three nodes where two consecutive nodes are directly connected. So in this structure, the DFS tree generated from node n_2 in the graph is inevitably a part of the DFS tree generated from n_1. Similarly, the DFS tree generated from node n_3 in the graph is a part of the DFS tree generated from n_2 and hence of n_1 as well.

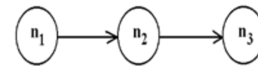


Figure 5. Chain Structure

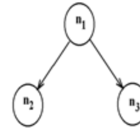


Figure 6. Cap Structure

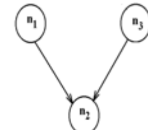


Figure 7. Cup Structure

So, the DFS tree from node n_1 will generate all possible paths from n_1 and will cover all paths starting from n_2 and n_3.

The ***Cap structure,*** of Figure 6, shows two nodes n_2 and n_3, linked to a root node n_1 giving it a cap-like structure. Hence, the DFS tree generated from n_1 will totally cover the DFS trees generated from both n_2 and n_3. So, the paths generated from n_2 and n_3 are all contained in the paths generated from n_1.

So in Figure 5 or in Figure 6, if node n_1 is considered to be a Source, DFS trees and consequently the paths from nodes n_2 and n_3 need not be considered for storage. The paths generated from n_1 will cover them.

The ***Cup structure,*** as shown in Figure 7, exhibits two nodes n_1 and n_3 linked to a node n_2 giving it a cup-like appearance. Here, the DFS tree under n_2, is a part of the DFS tree starting from either n_1 or n_3 because both covers node n_2. So if the paths are generated from both n_1 and n_3, paths from n_2 will appear twice. So, in order to avoid such redundancy, if paths from n_1 are generated, then from n_3 only the sub-paths between n_3 and n_2 need to be generated.

Once again in Figure 7, if both node n_1 and node n_3 are Sources, all the simple paths from both n_1 and n_3 will be generated and stored. However, all the paths from node n_2 will be sub-paths for both sets of simple paths. Path normalization process should stop this redundant storage of sub-paths. So, if the path generation starts from node n_1, all the simple paths from n_1 are generated and stored. In case of node n_3, only the edge (n_3, n_2) is stored, since the paths from n_2 have already been considered as sub-paths of the simple paths generated from node n_1. Similarly, if the path generation starts from node n_3, all the simple paths from n_3 are generated and stored. In case of node n_1, only the edge (n_1, n_2) is stored, since the paths from n_2 have already been considered as sub-paths of the simple paths generated from node n_3.

So depending on the starting node, the normalized sets of paths in the above two cases are different. Informally speaking, normalization process generates and stores non-redundant set of paths only, and the resultant set of paths obtained out of the normalization process is not unique. In order to avoid this problem, a standardized node encoding scheme needs to be used. Agrawal, Borgida and Jagadish [3] proposed a node encoding scheme in 1989, which subsequently has become popular and is widely used for numbering the nodes in a graph. The present paper has also followed the same nomenclature. As a result, node 1 in the DAG always refers to a Source node and the path normalization process also starts from this node making the normalized path set unique.

Hence path normalization results in a set of paths for the DAG that contains simple paths as well as sub-paths shared by two or more simple paths considered only once. Using the normalized set of paths, all the simple paths of the graph can be generated. The reconstruction process developed for this purpose, has been discussed in Section 3.2.

3.1.1 Modified DFS Algorithm

```
DFS (G)
 begin
        for each vertex u Є V (G) do
        begin
        π color[u]← WHITE
            (u)←NIL
        end;
        for each vertex v Є V(G)
        if ind(v)=0 then DFS_VISIT (v)
        end;
DFS_VISIT (v)
begin
        if Adj[v] ≠ ∅ and color[v] ≠ BLACK then
        begin
           color[v] = GRAY
            for each vertex x Є Adj[v] do
           if color[x] = WHITE or GRAY then
            begin
                 color[x] = GRAY
                path = path U (v,x)
                DFS_VISIT(x)
           end
           else
           begin
              path = path U (v,x)
              store path with new-id
              path = ∅
           end
        end
        else
         begin
           color[v] = BLACK
           store path with new-id
           path = ∅
        end
        color[v] = BLACK
  end;
```

Figure 8. Modified DFS Algorithm

The modified DFS tree generation algorithm proposed in this paper basically follows the nomenclature of the original DFS algorithm as given in [12]. In order to describe the tree generation process, the nodes of a graph are assigned with different colors depending on the state of execution of the corresponding algorithm. The assigned colors are:

WHITE: A node that has not been visited yet. Initially all nodes are WHITE.

GRAY: A node that has been traversed at least once but all paths out of it have not been traversed.

BLACK: A node is made black when all paths out of it are traversed.

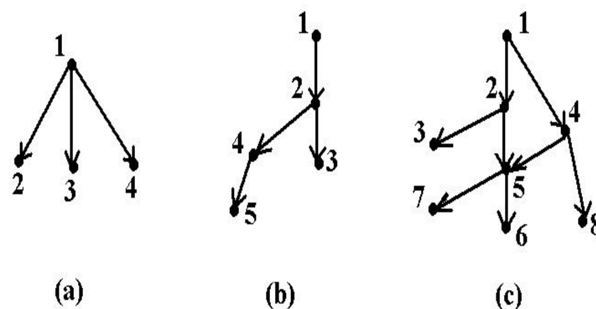
In the algorithm,
V(G) : Set of all nodes of a DAG (G) for which the algorithm is executed.
π (u) : Predecessor of a node u.
Adj(u) : Set of all adjacent nodes of a node u.
$\emptyset$: Represents a set to be null.
DFS : Function to generate the DFS tree. It takes the Graph under study as input.
DFS_VISIT: Function that takes a node as input and traverses its outgoing directed edges to reach its adjacent nodes. It is done recursively to generate the paths.

The modified algorithm of Figure 8, recursively generates the normalized set of paths from each Source. The salient features of the algorithm are:

- In the normalized set of paths generated from this algorithm, the starting node of any member path (simple path or a sub-path) is either a Source or an Anchor node, that is already a member of another path but still has successors.
- Any member path in the normalized set of paths either terminates at a Sink or at a node which has already been colored BLACK (i.e. all its successor nodes have already been visited).

If path generation is done only for Source nodes, standard DFS algorithm will generate all the simple paths. As a result the overlapping sub-paths among these simple paths will be repeated. The modified algorithm will consider an edge only once in the normalized set of paths.

3.1.2 Redundancy Avoidance



Figures 9. Sample DAGs

Salient features of path normalization algorithm can be explained from the sample DAGs shown in Figure 9. All the DAGs have a single Source node. DAG shown in Figure 9(a) has no Anchor node and so both standard DFS and the modified DFS should generate same path-set {(1,2), (1,3), (1,4)}. Figure 9(b) has only one Anchor node. Here, standard DFS will provide all simple paths as {(1,2,3), (1,2,4,5)}, whereas the modified algorithm will generate the normalized path-set as {(1,2,3), (2,4,5)}. Similarly, DAG in Figure 9(c) has the simple paths as obtained from standard DFS are, {(1,2,3), (1,2,5,6), (1,2,5,7), (1,4,5,6), (1,4,5,7), (1,4,8)}. On the other hand, normalized path-set as obtained from the modified algorithm is, {(1,2,3), (2,5,6), (5,7), (1,4,8), (4,5)}. These sample DAGs show that an edge is never repeated in the normalized path-set (e.g. edge (1,2) in Figure 9(b) is not repeated in normalized path-set). Moreover, even the number of paths in normalized path-set may be less than the total number of simple paths present in a DAG (e.g. in DAG of Figure 9(c), normalized path-set has 5 members against 6 simple paths present in the DAG).

Lemma 4: *Path normalization process avoids redundancy completely.*

Proof:

Redundancy avoidance will be considered complete only if no edge is repeated in the normalized set of paths. The path normalization process or the modified DFS algorithm executes a recursive function DFS_VISIT that terminates either on a Sink node or if the visited node has the color BLACK. Once again, as soon as all the descendents of a node are visited or if a node does not have any descendent, its color is made BLACK. So, the recursive function DFS_VISIT

never accesses any node beyond a BLACK node. So, a node once made BLACK (i.e. its descendents are already visited), edges beyond it will never be visited twice. Recursively, the DFS_VISIT function turns the color of the nodes from a Sink to a Source as BLACK eeping no scope of visiting the same edge more than once and subsequently to add them to a path and to store them. Hence the redundancy avoidance is complete and the path normalization process satisfies *Minimality Constraint.*

3.1.3 Time and Space complexity of Path Normalization

The time complexity of the standard DFS algorithm is O(V+E) [12]. Since path normalization uses a modified DFS algorithm, its time complexity is also O(V+E). However, since normalization process does not visit an edge more than once, in actual application the execution time for modified DFS algorithm in path normalization process will be less than that is required in a standard DFS for the same DAG. This saving of time will increase with the increase of number of anchor nodes in a DAG.

The number of simple paths generated by the standard DFS algorithm:

$$N_{SP} \leq N_S + N_S \cdot \prod_{u \in NA} |Adj[u]|$$

The number of paths (simple paths and sub-paths) generated in the normalization process:

$$N_{NP} \leq N_S + \sum_{u \in NA} |Adj[u] - 1|$$

where, N_{SP} = number of simple paths,
N_S = number of source nodes,
NA = number of anchor nodes,
N_{NP} = number of normalized paths.

$\prod$ = Product function. (It is different from π (u) used in the modified DFS algorithm of Figure 8, where π (u) represents the predecessor of a node u).

$\sum$ = Function for Summation.

From the expressions of N_{SP} and N_{NP} it is apparent that for a real life network, the cardinality of the normalized path set is much less than the number of simple paths present in the graph. Moreover the normalized set of paths contains many sub-paths that are much smaller than any simple path present in the graph. Similar to time complexity analysis, saving of space will increase with the increase of number of anchor nodes in a DAG.

3.2 Reconstruction process

As discussed earlier, all simple paths of a DAG should be computable from the normalized path-set. Moreover, a path query on a DAG may need to construct one or more simple paths from the normalized path-set. So, a proper reconstruct process is necessary. However, *Completeness Constraint,* as mentioned earlier should be satisfied. In other words, reconstruction process should not only generate all the simple paths present in original DAG from its normalized path-set, but it should also ensure that no additional path gets generated.

As discussed earlier in Section 3.1, the popular node encoding scheme described in [3] is used for the DAG generated after fusion and as a consequence, node 1 always refers to a Source node. So, the normalized path set generation algorithm always starts with a simple path starting from node 1.

During normalization, some data structures are built to store data related to normalized path-set. These data structures and other normalized path related information are used at the time of reconstruction. Each normalized path is stored with an id, a path type and the corresponding node sequence. Normalization process generates 4 types of paths.

- Path type [1,1] : A simple path i.e. a path that starts from a Source node and ends in a Sink.
- Path type [1,0] : A sub-path that starts from a Source node but ends in an Anchor node.
- Path type [0,1] : A sub-path that starts from an Anchor node but ends in a Source node.
- Path type [0,0] : A sub-path that starts from an Anchor node and also ends in an Anchor node.

Normalized path-set obtained from DAG in Figure 9(c) is, {(1,2,3), (2,5,6), (5,7), (1,4,8), (4,5)}. Table 1 shows how these paths are stored.

Path-id	Path-visit	Path-type	Node-sequence
P1	0	[1,1]	1,2,3
P2	0	[0,1]	2,5,6
P3	0	[0,1]	5,7
P4	0	[1,1]	1,4,8
P5	0	[0,0]	4, 5

Table 1. Normalized Path-set'

- For each path, normalized path-set provides the node sequence against its path–id. So, a search from a path to its nodes can be made using this structure by proper indexing against path-id. Since the normalization algorithm always starts from node 1 as generated by the node encoding scheme of [3], the first member of the normalized path set is always a simple path (path type [1,1]) that starts from node 1. A boolean flag Path-visit is associated with each path. Initially it is set to 0 for all paths. For each path it is set to 1 as soon as the path is visited for the first time. However, a node may appear in more than one path. So, to find all paths that contain a certain node, the same structure is not suitable. It needs a reverse mapping facility, i.e. a list, where against each node, the ids of normalized paths are provided where the concerned node is a member. A structure called *Node Bucket* has been made for this purpose. Against each node of a DAG following information are maintained:

Node-id, [in-degree, out-degree], Node-visit, Path-id$_1$ (start-node, end-node) ……. Path-id$_n$ (start-node, end-node)

Node-visit for each node is initially set to 0. It is set to 1 when the node is visited for the first time.
Node buckets for DAG in Figure 9(c) is shown in Table 2.

1	[0,1]	0	P1 (1,3)	P4 (1,8)	
2	[1,2]	0	P1 (1,3)	P2 (2,6)	
3	[1,0]	0	P1 (1,3)		
4	[1,2]	0	P4 (1,8)	P5 (4,5)	
5	[2,2]	0	P2 (2,6)	P3 (5,7)	P5 (4,5)
6	[1,0]	0	P2 (2,6)		
7	[1,0]	0	P3 (5,7)		
8	[1,0]	0	P4(1,8)		

Table 2. Node Buckets

Reconstruction algorithm for a DAG generates all simple paths using its normalized path-set and node buckets. It is a stack-based algorithm. The algorithm is given below :

Reconstruction Algorithm

```
procedure MAIN
begin
      Load files: Normalized Path Sets and Node Buckets // Refer Table 1 and 2
       P-Stack = empty; // primary stack used in reconstruction process
      S-Stack = empty; // secondary stack for exception cases in reconstruction
       leaf(v) = false;    // a Boolean variable assigned to each node v.
                            // Initially leaf(v) = false for all nodes;
                            // leaf (v) = true,  if a node is a leaf.
      int out-count(v);  // stores the current value of out-degree for each node v (outd(v) )
       set pointer bp to the first element of the node bucket referring to node 1;
                            // bp  is a pointer that points to the rows of the node bucket
                           //algorithm starts from first element of the first node bucket
                           // always refers to Node 1 and a path type [1,1]
      for each node n in node buckets
      begin
              out-count(n) = outd(n);
                  if outd(n) = 0 then leaf(v) = true;
              RECONSTRUCT(v);
      end;
              DELETE(p)
end;

procedure RECONSTRUCT(v)
begin
        repeat
                take an element from node bucket for node v;
                refer to corresponding path p in Normalized Path-set;
                begin
                do while Path-visit (p) =1 AND node bucket for node v not empty
                      go to next element of node bucket for node v;
                        Path-visit (p) =1;
                        for each node n in p
                        begin
                                PUSH (n, P-Stack);          //Uses the standard PUSH operator for a stack
                                If out-count(n) <> 0 then out-count(n) = out-count(n)-1;
                                Node-visit(n) = 1;
                         end;
                         if Path-type(p) <> [1,1] then
                                  begin
                                        if last node x pushed in P-Stack has leaf(x)=true then
                                             begin
                                                      save stack content as a new path with new path-id,
                                                      Path-type = [1,1] and Path-visit = 0 in Normalized Path-set;
                                                      Node buckets of involved nodes updated with new path-id;
                                             end
                                        else EXCEPTION(x);
                                  end;
                         repeat
                            POP (w, P-Stack); // standard POP operator to pop the last node in stack
                                if out-count(w) <> 0 then
                                 begin
                                      set pointer bp to node w in node bucket;
                                       RECONSTRUCT(w);
                                 end;
                        until P-Stack = empty;
                           end;
                           set pointer bp to next node n in node buckets;
                           if Node-visit(n) = 0 then RECONSTRUCT(n);
              until all nodes in node buckets are visited;
end;
procedure EXCEPTION(x)
begin
              set pointer bp to node x in node buckets;
              out-count(x) = outd(x);
                  repeat
```

```
begin
      take an element from node bucket for node x;
        refer to corresponding path p in Normalized Path-set;
        out-count(x) = out-count(x) – 1;
        do while Path-visit (p) =1 AND node bucket for node x not empty
                go to next element of node bucket for node x;
        Path-visit (p) =1;
        for each node n in p
        begin
                repeat while n <> x
                        PUSH (n, S-Stack); //Secondary stack is used for Exception cases only
                   for rest of the path p, PUSH (n, P-Stack);
        end;
        if last node t pushed in P-Stack has leaf(x)=true then
                begin
                        save stack content as a new path with new path-id,
                        Path-type = [1,1] and Path-visit = 0 in Normalized Path-set;
                        Node buckets of involved nodes updated with new path-id;
                end
        else EXCEPTION(t);
        repeat
                POP (n, P-Stack);
       until n = x;
 end;
until out-count(x) = 0;
end;
```

```
Procedure DELETE(p)
begin
   for each path p in Normalized Path-set
        if Path-type(p) <> [1,1] then
        delete p from Normalized Path-set;
        resultant Path-set is the reconstructed set of paths;
end;
```

A walk through the algorithm for DAG in Figure 9(c) augments the Normalized Path-set to generate the simple paths (1,2,3), (1,4,8), (1,2,5,6), (1,2,5,7), (1,4,5,6) and (1,4,5,7). The sub-paths generated during path normalization are removed by the process DELETE(p) leaving only the simple paths.

Reconstruction algorithm automatically proves the following Lemmas.

Lemma 5: *Reconstruction process is complete.*

Proof:

• Completeness of the algorithm ensures that from normalized path-set all simple paths can be generated. At the same time, algorithm does not generate any extra path not present in the original graph.

• Since, the stack-based algorithm traverses the normalized path-set and corresponding node buckets controlled by the specified out-degree of each node, it cannot generate any extra path.

• Most of the simple paths are generated by natural concatenation of normalized paths (*as in case of P1, P2 and P3 to generate simple paths 1-2-3, 1-2-5-6 and 1-2-5-7*). Even the exception cases are handled by Step 8. So, the algorithm automatically generates all simple paths of the original graph.

Lemma 6: *Path generation against query is minimal in the reconstruction process.*

Proof:

A path query is of two types :

Find all simple paths through node n_a.

Find all paths between node n_a and node n_b.

For the first case, if n_a is a Source node, simple paths are obtained by considering only the normalized paths in the node bucket of n_a and following the algorithm. Node buckets not involving n_a are not considered and so simple paths not involving n_a are also not generated.

If n_a is an intermediate node, back tracking the node bucket provides all simple paths involving n_a only (e.g. *node 5 tracks P2 to P1 and also P5 to P4*).

In the second case, paths for n_a and n_b are separately computed and intersection of two sets provides the required result.

Algorithm will not traverse all the paths in the normalized path-set or all the node buckets for the above queries. Hence, the path generation process is minimal.

4. Relevant Operators

Since a web-graph used as a social network usually involves hundreds of nodes and edges, it has been pre-processed to convert it to a DAG. Hyper-node based compression, described in Section 2, makes this conversion. Now, by path normalization process, described in Section 3, a limited number of paths and sub-paths of the compressed DAG are generated and stored apriori in order to avoid generation of all paths at query time. So, any query on the original graph representing a social network will access the structural components evolved during compression and/or the paths of the compressed DAG through the normalized path-set and node buckets. Now in order to process such queries, appropriate operators need to be defined to navigate over the compressed graph structure and the normalized paths.

Though this section does not cover the exact algorithms against each operator, it tries to justify the requirements of the operators defined.

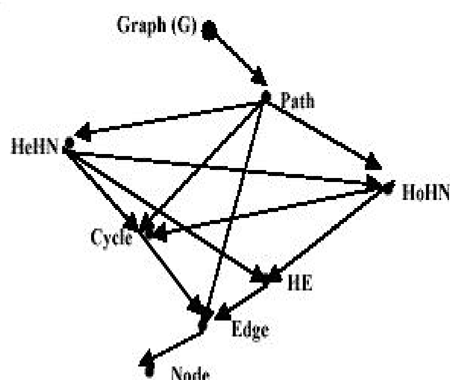


Figure 10. Structural Component Hierarchy

Figure 10 shows the structural component hierarchy as obtained from the hyper-node based compression process of Section 2. So any structural navigation would search through this ancestor and descendent hierarchy and appropriate operators are required for this process.

4.1 Hierarchical Navigation Operators

Ancestor: Ancestor or Containment ($\subset$) operator finds all the structures that are ancestor to a referenced structure as defined in structural component hierarchy of Figure 10. Similar to the method of referencing a node and to navigate a DAG structure described in [3] and [11], a propagate-upward operator or $\Uparrow(S_i)$ has been defined that recursively propagates upwards following the complex structural component hierarchy of Figure 10. It returns all the ancestors of a referenced structure S_i till the original digraph. So by definition,

$\subset(S_i) = \{\text{c-id} \mid \text{c-id} \in \subset(S_i) \cup \Uparrow(S_i)\}$ where, c-id = Structural Component id.

For example, from the sample social network of Figure 1, 2 and 3, ancestor operation on the homogeneous hyper-node H-1 (1-2-3) will return the heterogeneous hyper-node H-2 and the original di-graph.

$\subset$(H-1) = {H-2, G}

Descendent: Descendent ($\supset$) operator finds all the structures that are descendent to a referenced structure as defined in complex structural component hierarchy of Figure 10. Once again, similar to the method described in [3] and [11], a propagate-downward operator or $\Uparrow(S_i)$ has been defined that recursively propagates downwards following the complex structural component hierarchy of Figure 10, and returns all the descendents of referenced structure S_i till the leaves (nodes of original digraph). So by definition,

$\subset(S_i) = \{\text{c-id} \mid \text{c-id} \in \subset(S_i) \cup \Uparrow(S_i)\}$

For example, from the sample social network of Figure 1, 2 and 3, descendent operation on the homogeneous hyper-node H-1 (1-2-3) will return all the nodes and edges within it.

$\subset$ (H-1) = {(1,2), (2,3), (3,2), (3,1), 1, 2, 3}

Ancestor and Descendent operators are two fundamental operators to navigate through the structural component hierarchy generated by the hyper-node based compression process. Since this paper is mainly concerned with the graph structure of a web-graph, it is not concerned with any data that may be stored in each node of such graph. In real life application, the nodes will have data relevant to the application domain. So the fundamental operators ancestor and descendent will have to be modified to accept any node–based predicate and hence to make restricted retrieval of ancestors or descendents.

4.2 Path related Operators

Besides hierarchical navigation, a social network will also involve operators for path searching. Such operators primarily provide three types of information,

1. whether node m is reachable from node n, asking for a boolean output,

2. find all paths through node n, returning a set of paths,

3. find all paths between node m and node n. This operator will also verify using the first operator, whether node m is at all reachable from node n, before enumerating paths between them.

Reachability: Reachability operator (***R***) is a boolean operator that returns true if a node is reachable from a specified node. In the definition of strongly connected component in Section 2, justification for a reachability operator has already been discussed. So, in order to find whether a node y is reachable from another node x, the system should verify whether there exists any path that passes through both x and y where x appears before y in the node sequence. However, enumeration of paths between x and y is not necessary to ensure reachability. [3] has defined a node encoding scheme where each node i is associated with a node number i_n, where i_n indicates the number of the node i. Each node is also associated with an interval of node numbers $[s_n, t_n]$. The interval indicates that node number range for nodes s to t (both s_n and t_n included) are reachable from i.

If s_n covers $[p_n, q_n]$, then the condition that the node t is reachable from node s is,

$$R(s,t) = \begin{cases} \text{true}, & \text{if } p_n \le t_n \le q_n \\ \text{false}, & \text{otherwise} \end{cases}$$

For example, in Figure 1, *R*(4,8) will return true but *R*(8,4) will return false.

Path: Path (ρ) operator returns all possible paths from a source node, where a source is a node with in-degree zero. Enumeration of a path needs a recursion process that navigates from one edge to the other, recursively adding them in a sequence to form a path. [2] and [22] have proposed a recursion operator a over a relation R containing the edges of a graph and D is an attribute of a(R) contains a set of edges forming a path. Exploiting this process, the present effort also defines a similar operator over any digraph G for enumerating paths from any source node s as,

$$\rho(s) = \{p \mid p \in \prod \Delta(\sigma_{from = S}(\alpha(\mathbf{G})))\}$$

where, p indicates any path and σ represents the standard selection operator used in relational algebra.

For example,

for sample network (G) in Figure 3, $\sigma_{from = H\text{-}2}$(G) will return two edge sets {(H-2, 5), (5,6), (6,8)} and {(H-2, 5), (5,7), (7,8)}.

Now to apply σ operator, for each path p, $\prod \Delta$ considers the corresponding edge set and α(G) concatenates them to form a path. Thus two paths (H-2, 5)-(5,6)-(6,8) and (H-2, 5)-(5,6)-(6,8) are formed at the end of ρ (H-2) operation. The detail syntax and semantics of α and Δ are given in [2] and [22].

Enumeration: Enumeration operator (***E***) returns all possible sub-paths between any two nodes. If the concerned nodes are source and sink, then as per the definition of path, all

simple paths will be obtained. The process of enumeration is similar to the enumeration of path as discussed earlier.

An enumeration operator over any digraph G for enumerating paths from any source node s to any target node t is defined as,

$E(s,t) = \{p | p \in \prod_{\triangle} (\sigma_{from = s,\, to = t} (\alpha(\mathbf{G})))\}$ where, p indicates any path.

Connotation of α and $\triangle$ are same as in path operator. For example, in Figure 1, $E(5,8)$ would return 2 sub-paths, {(5,6), (6,8)} and {(5,7), (7,8)}.

Operators in this section cover only those, which are specially designed and considered for searching and navigation on a compressed web-graph used as a social network. An exhaustive list of operators required for executing queries on web repositories has been discussed in [27]. Authors of the present paper have used some of those operators for the implementation of their system but only the operators relevant to the compression and path normalization have been discussed in this paper.

5. Related Work

Sue1 and Yuan [33] have shown different sources of compressibility of web graphs and of the associated set of URLs in order to obtain good compression performance. Here the structure of the graph has been defined by considering a node for each page and a directed edge for each hyperlink between pages. The purpose of the study is to construct highly compressed representations of web graphs that can be stored and analyzed in machines with moderate amounts of main memory. Main operations considered are:

(1) getIndex(URL): Given a URL**,** return the index of the corresponding node in the structure.

(2) getUrl(index): Given the index of a node in the data structure, return its URL.

(3) getNeighbors(index): Given the index of a node in the data structure, return a list of the indices of all its directed neighbors. The authors have discussed some methods of compressing link structures and URL related texts. They have also discussed the possibility of fusing cluster of nodes to a single node but have not tried to implement it. Methods described in this paper are particularly suitable for search engines.

Exploiting random graph models for describing the web, Adler and Mitzenmacher [1] have developed algorithms that are based on reducing the compression problem to the problem of finding a minimum spanning tree in a directed graph related to the original link graph of the web. The salient features are:

· A compression algorithm specifically designed for graph structures with many shared links. Under appropriate assumptions, the running time of the algorithm is $O(n \log n)$, where n is the number of nodes in the graph. The algorithm requires finding a directed minimum spanning tree on a graph associated with the original graph.

· Authors have provided results demonstrating that several natural extensions of their algorithm are NP-Hard.

· Authors have also demonstrated the effectiveness of their approach on a test bed of random graphs derived from the random graph models that have motivated their work.

Boldi and Vigna [7] have considered the web as a graph where each node represents a URL and hyperlinks between them are the edges of the graph. The basic approach is a coding scheme for the links and URLs and the compression is done to such an extent that each link is represented by 3.08 bits. The study has been conducted considering a web graph having 118 Mega nodes and 1 Giga links.

Raghavan and Garcia-Molina [26] have proposed an S-Node representation of web graph for efficient query processing on the web. This paper proposes a two-level representation of web graphs, called an *S-Node representation*. In this scheme, a web graph is represented in terms of a set of smaller directed graphs, each of which encodes the interconnections within a small subset of pages. A top-level directed graph, consisting of "supernodes" and "superedges", contains pointers to these lower level graphs. By exploiting empirically observed properties of web graphs to guide the grouping of pages into supernodes, and using compressed encodings for the lower level directed graphs, S-Node representations provide the following two key advantages:

· First, S-Node representations are highly space-efficient. S-Node representations required little over 5 bits/hyperlink to encode the web graph structure, compared to over 15 bits/hyperlink for a straightforward Huffman-encoded implementation.

· Second, by representation the Web graph in terms of smaller directed graphs, this scheme has provided a natural way to isolate and locally explore portions of the Web graph that are relevant to a particular query. The top-level graph serves the role of an index, allowing the relevant lower-level graphs to be quickly located.

6. Conclusion

The present paper has discussed some compression techniques for a web graph used as a social network. Authors of the present paper have actually studied the possibility of developing a data model for a social network. Since many academic as well as commercial use of the web prefer to see a web graph as a social community, data model developed by the authors are also applicable to web communities. Web can be seen as sets of communities, connected to or isolated from each other. Each community consists of nodes representing the members of the concerned community with edges connecting them. Moreover within the same community all members may not have connection to all other members giving rise to the formation of a set of isolated sub-graphs. This entire paper has discussed about handling one such sub-graph. This can easily be extended to all the sub-graphs present in a community.

A web graph usually involves a large number of nodes and links. So, in order to handle any such graph in a data model, suitable compression techniques need to be developed for efficient use of disk space. The earlier works have shown that besides the effort of Raghavan and Garcia-Molina, none of them has considered compression on the graph structure. The approaches are centered around coding of links and URLs for efficient handling of web graph in the main memory. These approaches are particularly suitable for search engines and not for developing a data model.

Present paper has considered a two level compression technique. In the first level, the structural properties of a graph are studied and strongly connected components are fused to reduce the original graph to a DAG. Paths on this DAG are then stored efficiently using a Path Normalization technique. Space complexity expressions indicate the efficiency of the method. Relevant operators required for accessing the original graph through the compressed representations have also been discussed.

Future efforts will have two approaches :

· To study the performance of proposed compression techniques with different graph and DAG structures.

· A social network involves evolving graph structure, i.e. graph structure changing over time. Present study is restricted to a snap shot of a social network. Temporal variation of social

network and corresponding change in data model and compression techniques need to be studied.

References

[1] Adler, M., Mitzenmacher, M (2001). Towards compressing web graphs. *In:* Proc. of the IEEE Data Compression Conference (DCC).

[2] Agrawal, R (1987). Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, *In*: Proc. IEEE 3rd Int'l Conf. Data Engineering. p. 580-590.

[3] Agrawal, R., Borgida, A., Jagadish, H.V(1989). Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. *In*: Proc. ACM-SIGMOD. p. 253-262.

[4] Bagchi, A., Mitra S., Bandyopadhyay , A. K(2003). SONSYS – A system to model a Social Network. 6th.International Conference on Information Technology, Bhubaneswar, India, 371-374.

[5] Bang-Jense, J., Gutin, G (2001). Digraphs: Theory, algorithms and applications. Springer Monographs in Mathematics.

[6] Bhanu, Teja C (2005). Processing of path queries on Trees and DAGs. M.Tech.(Comp.Sc) Dissertation Report, Indian Statistical Institute.

[7] Boldi, P., Vigna, S(2004). The Web Graph Framework I: Compression Techniques. Intl. WWW Conference.

[8] Bruno, N., Koudas, N., Srivastava, D(2002). Holistic Twig Joins: Optimal XML Pattern Matching. Proc. ACM-SIGMOD.

[9] Chakraborty, S.(2004): Web Mining. Elsevier.

[10] Chen, L., Gupta, A., Kurul, E. M(2005). Efficient Algorithms for Pattern Matching on Directed Acyclic Graphs, *In*: IEEE Intl. Conference on Data Engineering.

[11] Christophides, V., Scoll, M., Tourtonis S (2003). On Labeling Schemes for the Semantic Web. World Wide Web Conference (WWW). 544-555.

[12] Cormen, T.H., Leiserson, C.E., Rivest, R.L(1990). Introduction to Algorithms. MIT Press.

[13] Guting, R(1994). GraphDB: A Data Model and Query Language for Graphs in Databases, *In*: Proc. of Intl. Conf. on Very Large Databases.

[14] Hanneman, R.A (2001). Introduction to Social Network Methods. Online textbook.

[15] Holland P.W., Leinhardt, S (1979). Perspectives on social network research. Academic Press.

[16] http://www2.heinz.cmu.edu/project/INSNA

[17] http://wizard.ucr.edu/~rhannema/networks/text/biblio.html

[18] http://wizard.ucr.edu/~rhannema/networks/text/textindex.html

[19] http://www.LinkedIn.com

[20] http://www.Ryze.com

[21] http://www.Tribe.net

[22] IMPRESS (1993): Specification of Graph Views and Graph Operators. IMPRESS (Esprit project no 6355), Technical Report W7-005-R75.

[23] Kautz. H., Selman. B., Shah. M.(1997). The hidden Web, *AI Magazine*, 18 (2) 27–36.

[24] Leinhardt S(1977). Social networks : A developing paradigm. Academic Press.

[25] Mitra, S., Bagchi, A., Bandyopadhyay, A.K(2006). A data model for a web graph used as a social network, *In*: Proc. First Intl. Conf. on Emerging Applications of Information Technology. Elsevier. 23-26.

[26] Raghavan, S., Garcia-Molina, H (2003). Representing Web Graphs, *In*: Proc. IEEE Intl. Conf. on Data Engineering.

[27] Raghavan, S., Garcia-Molina, H (2003). Complex Queries over Web Repositories, *In*: Proc. Intl. Conf. on Very Large Databases.

[28]. Rao, A.R., Bandyopadhyay, S.(1987): Measures of reciprocity in a social network, *Sankhya : The Indian Journal of Statistics. Series A*. 49. 141-188.

[29] Rao, A. R., Bandyopadhyay, S., Sinha, B.K., Bagchi, A., Jana, R., Chaudhuri, A. and Sen, D.(1998).Changing Social Relations – Social Network Approach. Technical Report, Survey Research and Data Analysis Center, Indian Statistical Institute.

[30] Rosenthal, A., Heiler, S., Dayal U., Manola, F(1986): Traversal Recursion: A Practical Approach to Supporting Recursive Applications. *Proc. ACM-SIGMOD*. 166-176.

[31] Shasha D., Wang, J.T., Giugno, R.(2002): Algorithms and application of trees in graph searching. Proc. ACM-PODS.

[32] Sheng, L., Özsoyoglu, M.Z., Özsoyoglu, G (1999). A graph query language and its query processing, *In*: Proc. IEEE Intl. Conf. on Data Engineering.

[33] Sue1, T., Yuan, J (2001). Compressing the Graph Structure of the Web. Proc. IEEE Data Compression Conference (DCC). 213-222.

[34] Yu, B., Singh, M.P(2003). Searching Social Networks, Proc. AAMAS.

Susanta Mitra is a Professor at the Computer Science & Engg. and IT department, Meghnad Saha Institute of Technology (MSIT), Kolkata, India. He has served in software industries for more than 17 years and 3 years as an Associate Professor at IIIT, Kolkata before joining MSIT. His areas of research for Ph.D. from Jadavpur University, India, included Web Graph analysis and data modeling, social networking and object-relational database technology. His current research interests are in Web data management, Web data mining, data structure and algorithms. He has published papers in international conferences that include publications from IEEE, Elsevier Science. His papers have also been selected for publication in renowned international journals.

Aditya Bagchi is the Chief, Consultancy and Development (a professor's post) at the Computer & Statistical Service Center, Indian Statistical Institute, Kolkata, India. Prof. Bagchi received his Ph.D (Engg.) from Jadavpur University in 1987. He has served in Tata Consultacy Services, Tata Burroughs Ltd, CMC Ltd and Regional Computer Center, Kolkata before joining the Indian Statistical Institute in 1988. Prof. Bagchi has also served as visiting faculty of Jadavpur University and BE Collage (Bengal Engineering and Science University), India. He has served as visiting scientist at the San Diego Super Computer Center, University of California, San Diego, USA and at the Center for Secure Information Systems, George Mason University, Virginia, USA. Prof. Bagchi is a Senior Member of Computer Society of India, member of ACM Sigmod and IEEE Computer Society. His research interests include Access Control and Trust Negotiation algorithms, developing new measures for Association Rule mining and application specific Data Modeling.

Anup Kumar Bandyopadhyay received the B.E. Tel. E., M. E. Tel. E and Ph.D. (Engg.) degrees from Jadavpur

University, Kolkata, India in 1968, 1970 and 1983 respectively. From 1970 to 1972 he worked with the Microwave Antenna System Engineering Group of the Indian Space Research Organization. In 1972 he joined the Department of Electronics and Telecommunication Engineering, Jadavpur University, where he is currently a professor. His research interests include program verification and construction of distributed systems.