**Design and Analysis of Algorithms**

**Project Phase 1**

**Group Members**

Ali Kamal (19I-1865)

Daniyal Shaiq (19I-1794)

Musa Dildar (19I-1765)

Saifullah Rahool (19I-1779)

**1. Description of the introduction of the domain necessary to understand the basic concept of the paper (6 marks).**

The problem this paper addresses is topological sorting. Topological sorting is the problem of finding a linear ordering of the vertices in V such that the tail of each edge in E precedes its head in the ordering. This problem arises in a number of applications where the goal is to find a linear ordering of items or activities (represented by the vertices of G) consistent with a set of pairwise ordering constraints (represented by the edges of G). The problem of topologically sorting large DAGs (Directed acyclic graphs) arises, for example, in the application of recent multiple sequence alignment algorithms [Zhang and Waterman 2003, 2005] to large collections of DNA sequences and as a building block for other algorithms for large datasets.

The paper deals with large graphs that cannot fit in the main memory at once, so it needs to be accessed from the external memory in size, which can fit in the main memory. Such problems have very large number of I/O operation – operations that requires disk access and writing.

**2. Brief Discussion of the related work discussed by the authors ((3 marks).**

There are various other related works referred by the author. In fact, the author considers these other works as competitors of the proposed algorithm by the author. Some of these algorithms outperforms the proposed algorithm in some in inputs. One of the approaches the author mentioned is the naive approach for finding topological sorting using Depth First Search. There are other approaches to topological sorting that the author discusses, as they are either natural or were proposed with I/O efficiency or parallelism in mind and, thus, may achieve better performance than the proposed algorithm, at least on certain inputs. The names of the other algorithms are Topological Sorting Using Semi-External DFS (SETS), Iterative Peeling of Sources and Sinks (PEELTS), and Divide and Conquer Based on Reachability Queries (REACH TS).

**3. (a) What is the aim of the paper? What is the problem addressed in the paper?  (b) How the paper is different from the previous work as claimed by the authors. (5 marks)**

**a)**

The aim of the paper is to propose an I/O efficient algorithm - such an algorithm which has less disk read and write operations – for topological sorting, particularly for directed acyclic sparse graphs. So far, there’s no general I/O efficient algorithm for topological sorting of sparse graph; this lack of provably I/O-efficient algorithms for sparse directed graphs has led to

the development of a number of heuristic approaches for solving problems on directed

graphs I/O-efficiently. So, this paper also proposes a heuristic approach for topological sorting large graphs I / O efficiently. One of the approaches for processing large sparse graphs is time forward processing, but it requires a DAG to sorted topologically. Since no general I/O-efficient topological sorting algorithm is known to date, time-forward processing has been applied only in situations where a topological ordering of the vertices can be obtained by using secondary information about the structure of the DAG. A general I/O-efficient topological sorting algorithm would greatly increase the applicability of this technique.

**b)**

There are other sorting algorithms already for topological sorting, but some of them are highly inefficient and some cannot handle graphs beyond its memory, so to solve this backlog this paper provides an algorithm for topological sorting, and it can handle even large graphs that do not fit in the memory. One of the methods discussed previously is Topological Sorting Using Semi-External DFS (SETS), this method outperforms the proposed algorithm in width-first algorithms, but it starts failing as the size of the graph increases or goes beyond the size of the memory. The other algorithms mentioned in the paper (REACH TS) and (PEEL TS) are no match for the proposed algorithm in terms of time complexity and handling the graphs beyond the size of memory. Among these algorithms, only SETS was able to compete with the proposed algorithm for certain width-first algorithms

**4. Details of the methodology adopted to solve the identified problem. (8 marks)**

The basic gist of the algorithm can be summarized in the following lines: If an edge of a Directed Acyclic Graph’s (DAG) has a tail with a lower number than its head, then that edge is considered as “satisfied”.

Initially, the whole of the graph is initialized with certain numbering. Then a loop is run, until the whole of the graph is satisfied, which improves the initialized numbering i-e the loop will run until all heads are less than the tails. The authors also discuss a heuristic whose addition to the procedure that improves the number, led to a tremendous performance improvement by reducing the number of iterations the algorithm needs to find a topological ordering.

As explained, we first compute the initial numbering of the graph. To do that, an OutTree is computed. An OutTree is just like a spanning tree, in which that it only has a single source. After this, the EulerTour is calculated. It essentially gives numbering to the graph. Subsequently, we perform Preorder Numbering on the EulerTour, which returns two preorder numberings, one for the EulerTour, and another for the Reverse EulerTour. After this, the Initial Numbering is performed, which checks the number of satisfied edges in both the preorder numbering list for the EulerTour, as well as the preorder numbering list for the Reverse EulerTour. If the regular EulerTour has a greater number of satisfied edges then the Reverse EulerTour, then the preorder numbering list of Regular EulerTour is returned, else the preorder numbering list of Reverse EulerTour is returned.

The ImproveNumbering Algorithm, which is run after the InitialNumbering algorithm, takes the whole graph and returns a specific spanning tree that has the worst number of violations (tail has higher number than head). Then a SatisfyTreeEdges algorithm is run, which takes the above violated tree, and “satisfies” it, and returns a new numbering. Now that the violated tree is satisfied, we need to satisfy the rest of the graph. Hence, an algorithm is run to satisfy the rest of the graph. Finally, we get the ordering, and sort it, and count the number of satisfied edges.

**5. How the authors have provided the evidence to justify their claims. In this  
section you can discuss the results provided in the paper. (8 marks)**

In the paper, the author presents the values of experiments he did on his algorithm. In the experiment, the author runs his algorithm along with other algorithm, and at the end the author compares the results to justify his claims. The author has used different structure of the graphs - such as Random Graphs, Width-One graph, layered graphs, semi layered graphs, low width graphs, grid graphs, and width graphs – along with different number of nodes for each category. The main goal of our experiments was to compare the algorithms, study how they are

affected by the structure of the input graph, and use the results to recommend which

algorithm to use if there is a prior knowledge of the graph structure. Table I (attached below) shows the running times of the algorithms on different input graphs. In order to bound the

time spent on the experiments, the following rules were used: Each algorithm was

given at least 10 times as long to process a given input as it took ITERTS to process the

same input. If it did not produce a result in the allotted time, it was terminated. This

is thus indicated by dashes in the table, with superscripts indicating the amount of time

given to the algorithm. If ITERTS took more than one day to process an input and

was consistently faster than the other algorithms on smaller inputs of the same type,

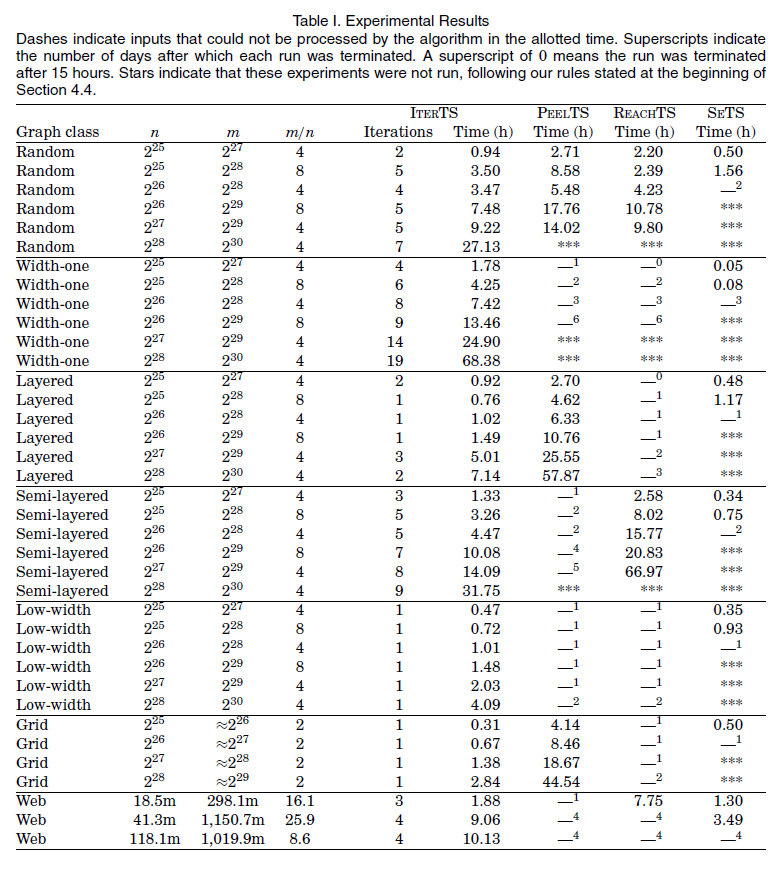
other algorithms were not run on this input. This is indicated by stars in the

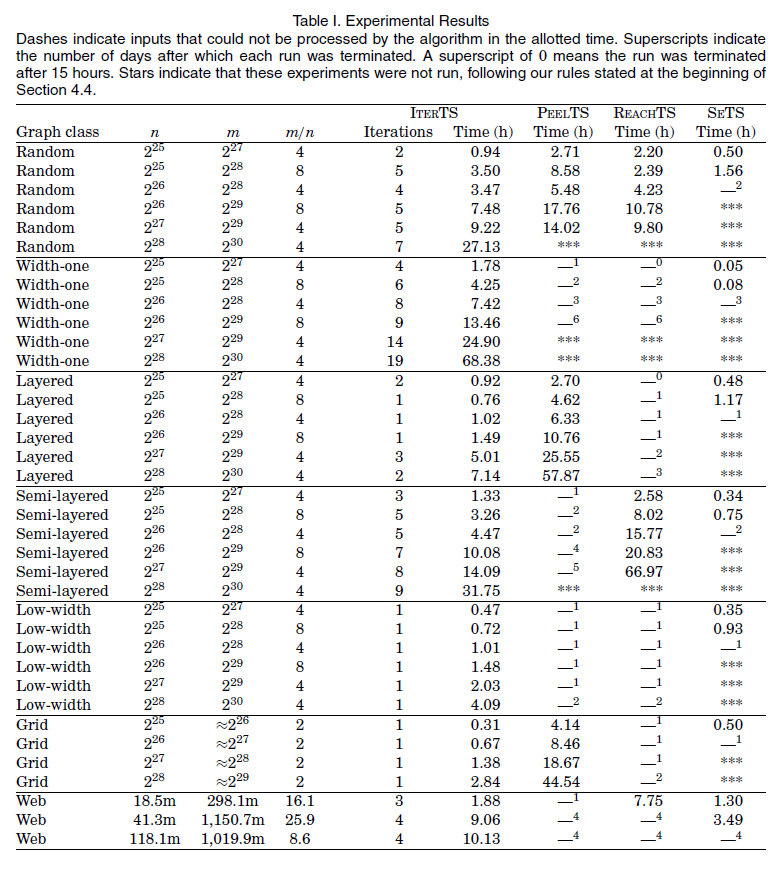
table. Since SETS is a semi-external algorithm and 2 26 vertices do not fit in 1GB of

memory, it was not run on larger inputs if it did not finish in the allotted time on

the smallest input with 2 26 vertices (which was the case for all input types).

All the algorithms were tested and the results are compared in the table below. IterTS outperformed all the other algorithms. To elaborate, random graphs were easy for all algorithms with a factor of less than 2 between running times of each algorithm. On most other times PeelTS and ReachTS weren’t able to process the inputs in the allotted time. PeelTS though able to process layered and grid graphs, was still more than 10 times higher than IterTS. To make it harder for PeelTS to process layered graphs, semi-layered graphs were introduced to eliminate its upper hand in traditional layered graphs. In web graphs, ReachTS was able to process one of the graphs in 4 times the time taken by IterTS but not process bigger graphs while PeelTS wasn’t able to process a single graph in the allotted time. For inputs that fit in memory, SeTS generally outperformed IterTS of which width-one graphs were very easy. For inputs that did not fit in memory however, SeTS was not able to complete a single graph and was outperformed by IterTS immensely. Therefore, it can be concluded that IterTS is indeed the fastest, after which comes SeTS (which is faster in semi-external inputs) while PeelTS and ReachTS could not even compete with the likes of the two. The graph’s structure, while effecting the likes of PeelTS and ReachTS, doesn’t seem to effect IterTS and SeTS too much. IterTS’ performance can be considered robust and independent of the graphs structure. IterTS was ran on different types of graphs of different sizes but with a fixed density and the running time increased linearly for layered and low width graphs as number of iterations is nearly independent of the size of the graph. For random, width-one and semi-layered graphs the iterations increased till termination according to the input size therefore concluding a linear dependence on the input size.





**6. Challenges that can be foreseen during the implementation of the paper. (5 marks)**

Ans