**TOPOLOGICAL ORDERING: THEORY AND**

**REAL WORLD APPLICATIONS**

**BY:**

XXX

**CS 253 - 70**

**CENTRAL CONNECTICUT STATE UNIVERSITY**

**ABSTRACT**

This paper looks at the various implementations of Topological ordering. It is one of the most useful sorting method for graphs which has many real world applications. This paper will focus on the theoretical implementations of two different algorithm which build off of another basic algorithms by changing them to some degree. First the definition and prerequisites for Topological ordering will be discussed. Then the paper will focus on the implementation of Topological ordering with two algorithms: Kahn’s algorithm and Tarjan’s algorithm. The efficiency of both these algorithms will also be discussed. Finally some real world applications of Topological ordering, one of which is the answer to the shortest path problem, albeit with the prerequisites of Topological ordering making the answer only work for a particular class of graphs called DAGs (Directed Acyclic Graphs).

**METHODOLOGY**

We became interested in Topological ordering after learning that it is used for scheduling applications everywhere. It is such a simple, yet powerful, concept that is used by large-scale software such as all operating systems to take care of the services that they are currently running.

We started looking for definitions of what exactly a topological ordering is in theoretical manner. Breaking down the definition into small pieces, we set out to find everything about all the things required for Topological ordering. There are only one major thing that is needed for topological ordering (or sometimes called topological sort): Directed Acyclic Graphs. We studied all the properties of DAGs, and found out how they are different from general graphs, and what their uses and applications could be in the real world.

Then knowing the prerequisite, we started looking for different implementations of topological ordering. We found two different algorithms, one being recursive (Tarjan’s algorithm) while the other one was iterative (Kahn’s algorithm). Both algorithms have the same efficiency, therefore it is just a matter of choice. Finally, we started looking for some real world applications of topological ordering, and there were many unique applications. Topological ordering is used by something as simple as a compiler, and by something as complex as a fully developed software MS Excel. While looking for some practical applications, we found that topological ordering can also be used to solve the shortest path problem (even on graphs with negative weights) and having better efficiency than algorithms created for the shortest path problem itself. Although the only catch is that because topological ordering is being used, the graph has to be a DAG, and it must have a topological order.

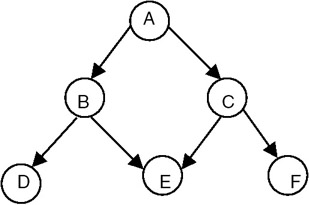
Topological ordering is an immensely powerful concept with vast applications. Its major appeal is the very easy implementation which could be used as a base for complex applications.

**Directed Acyclic Graphs**

Directed Acyclic Graphs are special types of graphs, with (**u**, **w**) edges where **u -> w**. The edges connect two vertices of a graph which contain any type of Objects or data (such as int, double, etc.). These directed edges can only be traversed in the direction of the arrows. To traverse from **w->u**, there needs to exist another edge which points to **u** from **w**. The three properties that DAGs have to satisfy regarding the precedence among vertices are:

* Irreflexivity, i.e. for all s ∈S, s -/-> s
* Asymmetry, i.e. for all s, t ∈S, if s → t, then t -/-> s
* Transitivity, i.e. for all s, r, t ∈S, if r → s and s → t, then r → t

Where S is a set of vertices.

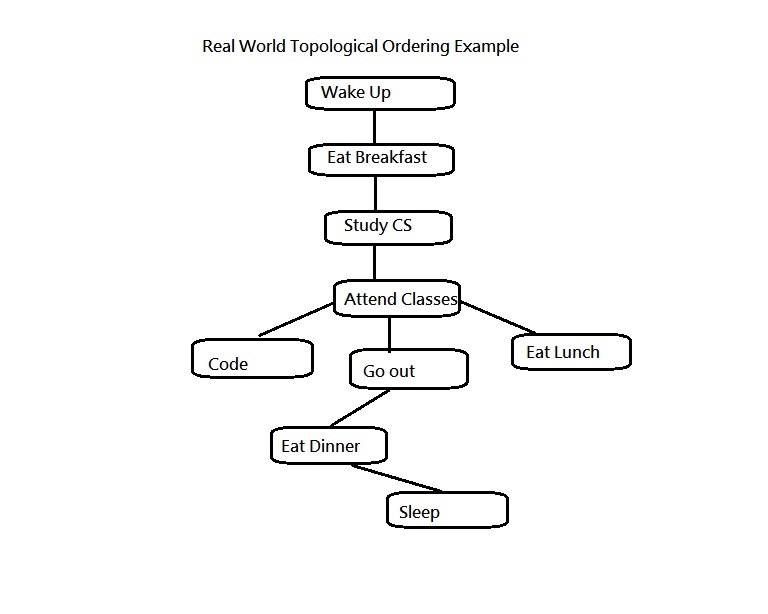
One example of a DAG is given below: 

In the above example, the only way to start traversing from the top is from A to either B or C. B cannot be reached, if the traversal is currently at vertex D. One traversal of the above DAG is

A->B->C->D->E->F. Note that vertex E can only be accessed if and only if B and C have both been visited before.

**A Typical Student Day**

Before revealing what topological ordering is, an example has been used to illustrate it below as a graph. The graph represents every activity that a computer science major does on an average day.



One of the many topological ordering of the above DAG, can be by shown by creating a list which contains the list of all the vertices of the graph exactly in the order they were vistited in. Note that each activity shown in the DAG is a vertex of the graph:

“Wake Up” -> “Eat Breakfast” -> “Study CS” -> “Attend Classes” -> “Code” -> “Eat Lunch” -> “Go out” -> “Eat Dinner” -> “Sleep”

**Theoretical Definition of Topological Ordering**

Suppose there is a Directed Acyclic Graph (DAG) which has (**u**, **w**) edges where vertex **w can only be visited if and only if u has already been visited**. Then the topological ordering of the DAG is an ordered list of the vertices such that if there is a path from vertex **u** to vertex **w** in the graph, then vertex **u** appears before vertex **w** in the list. Each vertex is visited exactly once, and only after its prerequisite has been visited (and put into the topological ordering list) beforehand. There are two types of algorithms for finding a topological ordering for any DAG:

* Kahn’s algorithm (Uses indegree for all vertices)
* Tarjan’s algorithm (Uses a modified Depth- First Search algorithm)

It is very important to remember that both these algorithms can only work for a DAG, and not for any other graphs because of a DAG being a prerequisite in the definition of topological ordering.

**Tarjan’s Algorithm for Topological Ordering**

The algorithm loops through each vertex of the graph, in a particular direction, initiating a depth-first search that terminates when it hits any vertex that has already been visited since the beginning of the topological sort or if the vertex has no adjacent vertices. This implementation of topological ordering is a modified version of depth-first search making it a recursive algorithm. Its use was deprecated when memory was scarce for computers. But now this implementation is used widely as memory is not an issue at all. The pseudo code for the algorithm follows below. Note that the pseudo code assumes that the graph is made up of Integer objects,

Algorithm topologicalOrder(currentNode, visited[], tempStack):

Visited[currentNode] := true

Iterator<Integer> iterate := adjacencyList[currentNode].iterator()

While(iterate.hasNext())

tempNum := iterator.next()

If(!visited[tempNum])

topologicalOrder(tempNum, visted[], tempStack)

tempStack.push(new Integer(currentNode))

The algorithm keeps going to adjacent nodes recursively into the graph (depth-wise) until the call hits an already visited adjacent vertex, and then it starts coming back from the recursive call by storing each vertex in a stack. At the end of the algorithm, the temporary stack contains the topological ordering of the graph in the reverse order, so it can be popped in the proper order. The visited Boolean array also contains true as a value for every vertex. In this algorithm each vertex in the graph is visited exactly once.

The running time efficiency of tthis implementation is O(|V|2) where |V| is the number of vertices present in the DAG whose topological ordering has been performed. Pushing and popping vertices out of the temporary stack takes a constant O(1) time. And finally, the space complexity of Tarjan’s algorithm is O(V).

To illustrate step by step how this implementation works, an example graph (a DAG) has been shown which will be topologically ordered using Tarjan’s algorithm:

* Suppose the function starts the topological ordering from vertex **a.**
* Next visit the first vertex adjacent to **a**, which is **b**.
* Recursively, visit **b**’s adjacent node, **d.**
* Then **d**’s adjacent node **f**.
* Push **f** into a temporary stack.
* Come back from all the recursive calls, storing each

vertex in this order into the stack: **b->d**.

* Now the stack is **b->d->f**.
* Now go to the second adjacent node for **a**, **c**.
* Visit **c**’s adjacent node **e**, but notice **e**’s adjacent vertices **d** and **f** were already visited. Therefore, it’s not visited and because there are no other adjacent vertices for **e**, the function pushes **e** onto the stack.
* Then **c** and, finally, **a** is pushed onto the stack.
* The stack stores: **a->c->e->b->d->f**.
* The stack contains the vertices in reversed topological order. When the function pops the whole stack, the vertices align into proper topological order.

**Kahn’s Algorithm for Topological Ordering**

The second algorithm that will be discussed in this paper is called Kahn’s algorithm. This implementation of topological ordering is iterative unlike Tarjan’s algorithm, which is recursive. Kahn Algorithm works by choosing vertices with 0 indegree (number of incoming edges of a vertex) as the eventual topological sort. First find a list of start vertices which have no incoming edges and insert them into a set S. At least one such vertex must exist in a directed acyclic graph.

Then change the indegrees of the adjacent vertices of the vertices inserted in S. If the adjacent vertices’ indegree becomes 0, insert this vertex into S too.

The pseudo code for this implementation is as follows:

* Initialize sorted list to be empty, and a counter to 0
* Compute the indegrees of all nodes
* Store all nodes with indegree 0 in a queue
* While the queue is not empty
* Get a node U and put it in the sorted list. Increment the counter.
* For all edges (U,V) decrement the indegree of V, and put V in the queue if the updated indegree is 0.

The efficiency of Kahn’s algorithm is O(|V|2) because the implementation has to check every vertex and every edge for the whole DAG in a for loop nested inside a while loop. The while loop is going to run V times, and the for loop E times. Finally, the space complexity is O(V). This algorithm uses a queue to store all the vertices which have an indegree of 0, therefore storing vertices in a topological order. Also, there is an array which keeps track of all the visited vertices, so that the algorithm visits every vertex exactly one time. Therefore this algorithm resembles more of the breadth first search in the sense that it examines every path of length *i* before going on to paths of length *i+1*.

**Real World Applications of Topological Ordering**

This paper illustrates various real life applications where topological ordering is used. Of course, the implementation is far more complex than shown in this paper, but these implementations are the building blocks for these complex applications.

**Resolving Dependencies:**

During compilation the compiler has to know which files out of 1000s of source code files are libraries, and other classes which are needed to compile other files.

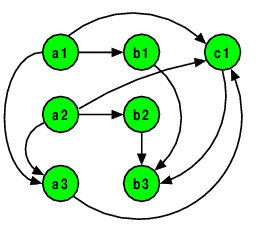
For example, inheritance between classes of a Java program have to be compiled in a particular order. (Where each class is a vertex, and inheritance connections are edges in a graph)

**Scheduling:**

Scheduling applications (such as the example of a typical student day) need to execute tasks in a particular order. Thus, they create each task as a vertex and each prerequisite for a task as an edge in a DAG. Then they use topological ordering to get the schedule for a particular day which includes all the tasks in a proper order.

**Microsoft Excel:**

Ordering of cell formula evaluation in a particular formula. Suppose Excel has to evaluate B1+C2 = E6. The program has to know exactly which cells will be evaluated first. Therefore, Excel creates each cell as a vertex and according to all the formulas it creates the edges in a DAG. Because of the evaluation being in a “must-be-evaluated” relation, Excel has to use topological ordering to evaluate all the cell formulas. Below is an example of cell evaluation process done by MS Excel:



**Special Application for Topological Ordering**

For all the weighted DAGs, we can find their topological order and use that to find the shortest path for all the vertices from a start vertex. This algorithm runs in quadratic time, i.e.

O(|V|2). One major advantage of using this algorithm is that it works for negative weights, unlike Dijkstra’s algorithm. It is also faster because, if there is a topological order between two vertices, then it is assured that there is a path between these two vertices. This means that the function does not have to check for paths between each vertex pair for the whole graph.

The pseudo code for this implementation is given below. Remember that this function expects there to be a topological ordering of the DAG done already and stored in a data structure:

* Let d be an array of the same length as number of V in the DAG;
  + This will hold the shortest-path distances from s.
* d[s] := 0, all other d[u] := ∞
* Let p be an array of the same length as V, with all elements initialized to nil.
  + Each p[u] will hold the predecessor of u in the shortest path from s to u.
* Loop over the vertices u as ordered in T:
* For each vertex v directly following u (i.e., there exists an edge from u to v):
* Let w be the weight of the edge from u to v.
* Relax the edge: if d[v] > d[u] + w, set d[v] ← d[u] + w,
* p[v] ← u.

This shortest path problem’s solution is efficient, but only works on a special subset of graphs and assumes that topological ordering of the DAG has been successful. This makes the algorithm not favorable while choosing a solution to the shortest path problem.