ENEE641 Project Report: 
Topological Sort Algorithm

Ren Mao 
School of Electrical and Computer Engineering 
University of Maryland 
Email: neroam@umd.edu

Abstract

This article describes how to apply the "peeling zero in-degree" topological algorithm to a directed graph to find out one topological order, or report that it contains a cycle. It describes how to implement the algorithm, and analyzes its time complexity. At last, it gives the experimental results for the inputs provided and validation statements connecting results with analysis.

I. Code Overview

In this project, I apply the topological sort algorithm on given directed graphs. The code is written in C and compiled with gcc in Linux Fedora 17 environment. The main program and its usage are below:

```c
int main(int argc, char *argv[]) {
    Graph mygraph;
    CreateGraph(argv[1],&mygraph);//Building the graph
    int queue[mygraph.vexnum],cycle;
    cycle = TopoSort(&mygraph,queue);//Toposort Algorithm
    PrintSort(argv[1],cycle,queue,mygraph);//Output the results
    return 0;
}
```

Usage: ./TopologicalSort [FileName]

The main program has three parts as below:

- **CreateGraph(argv[1],&mygraph):** This subfunction opens the input file and creates the graph structure. If the input file exists in the path, it will scan the input file and store the data in structure “mygraph” by interpreting it as directed graphs. The input should be given in plain text format, where vertices are numbered from 1 to \( V \). Each row includes a vertex, followed by zero or more other vertices which are interpreted as directed edges.

- **TopoSort(&mygraph,queue):** According to the graph structure, it will apply topological algorithm to find one topological order and store the data in array “queue”, or return a flag to report that it contains a cycle. Meanwhile, there are some extra codes for this sorting part to count the number of commands charged to this algorithm.

- **PrintSort(argv[1],cycle,queue,mygraph):** This subfunction will output the sorted order vertices and the operations charged to vertices and edges, or report that it contains a cycle if the flag "cycle" is true. The output format depends on the filename of input "argv[1]". If the file name is "in1.txt", it will display the detailed information according to specification part A&B, or it will display the overall number as format part A&C. At last, it will give the number of vertices and edges, and report the complexity respect with \( V \) and \( E \).

II. Graph Interpretation

To describe a graph in program, I firstly define a structure of Vertex and then use that verex structure to define a graph structure as below:

```c
typedef struct Vertex{
    int data;
    int ops;
    struct Vertex *adj;
}Vertex;
typedef struct 
    { 
    Vertex vertices[MAX_VER_NUM]; 
    int vexnum; 
    int c_ops; 
}Graph;
```
The Graph structure contains three elements as below:

- **vertices[MAX_VER_NUM]**: This array is to store all the vertices of the graph. It has a maximam capacity of MAX_VER_NUM which is predefined as a marco(being set to 100 as default). Every element in this array is a Vertex structure which represents a vertex.
- **vexnum**: This is the total number of vertices in the graph.
- **c_ops**: This is to count the number of constant commands, such as declaration of variables, in topological sorting algorithm.

The Vertex structure could be used to store vertices and edges. Only the array “vertices” in structure Graph means vertices, others represent edges. They contains the same type of elements but different meaning:

- **data**: For vertices, it stores the number of vertices from 1 to V ; for edges, it stores the array index of the dest vertex of the directed edge.(e.g: vertex 3 is stored in array vertices[2], the data of vertices[2] is 3, edge(1,3) is stored with data 2 in a Vertex structure in the link list of vertices[0])
- **ops**: To count the operations charged to corresponding vertex or edge.
- ***adj**: For all the elements in array vertices, this pointer points to a edge from the vertex which is stored in the vertices element. It could be NULL in case some vertex has no edges pointing out. For the elements in the linked list for edges, this pointer is pointing to the next edge element, which has the same source vertex.

### III. Algorithm Description

In this project, ”peeling zero in-degree” topological algorithm is implemented to find out one topological order of given directed graph \(G=(V,E)\) or report that it contains a cycle. If \(G\) contains an edge \((u,v)\), then vertex \(u\) should appear before \(v\) in the ordiering. The algorithm implemented is given in Algorithm 1.

#### Algorithm 1 Topological Sort

**Initialization:**

Set \(S = \emptyset\); Calculate indegree\([1,2,\ldots,V]\); \(Q \leftarrow \) set of all nodes with indegree zero;

**Iteration:**

1. while \(Q\) is not empty do
2. \(Q \leftarrow Q - \{v\}\)
3. \(S \leftarrow S \cup \{v\}\)
4. for all node \(u\) with an edge \(e\) from node \(v\) do
5. \(\text{indegree}[u] \leftarrow -\)
6. if \(\text{indegree}[u] = 0\) then
7. \(Q \leftarrow Q \cup \{u\}\)
8. end if
9. end for
10. end while
11. if \(S\) does not include all the vertices then
12. return cycle flag
13. else
14. return sorted \(S\)
15. end if

To analyze this algorithm, in the initialzaiton stage, it should be \(O(V + E)\) in time complexity since it will visit all edges to calculate the indegree for all vertices. In iteration stage, since every vertex is visited by twice, one is to get into \(Q\) and the other is to get out of \(Q\), and every edge is visited by once, to remove the edge from the graph. Therefore, this algorithm should be efficient enough. More detailed asymptotic analysis will be givin in next section.

### IV. Asymptotic Analysis

During the Initialization stage, based on the input graphs, I use a indegree array to store the indegree of every vertex by visiting all the linked list of edges. Then, visit all vertices to find the zero indegree vertices and insert them into a queue so that they can be processed as sorted vertices. During the Iteration stage, I process and dequeue every element in the queue until there is no element in the queue which means no zero indegree vertex left by removing all the edges from the processing vertex and decrement the corresponding indegree of the dest vertices. When the iteration terminates, if all the vertices has been processed through the queue, I get the topological oder of the graph, otherwise it means there is a cycle in the graph. Therefore, Topological sort algorithm is implemented as below:
int TopoSort(Graph *mygraph, int queue[])
{
    int indegree[mygraph->vexnum], i, front = 0, rear = -1; mygraph->c_ops++;
    Vertex *p; mygraph->c_ops++;
    //Count c_ops as constant operations
    //Calculate Indegree for Vertices
    for (i = 0; i < mygraph->vexnum; i++){
        indegree[i] = 0; mygraph->vertices[i].ops++;
        //Count vertices.ops as operations for corresponding vertex
    }
    for (i = 0; i < mygraph->vexnum; i++){
        p = &mygraph->vertices[i]; mygraph->vertices[i].ops++;
        while((p = p->adj) != NULL) {
            indegree[p->data]++; p->ops++;
            //Count p->ops as operations for corresponding edge
        }
    }
    //Find Vertices with zero indegree
    for (i = 0; i < mygraph->vexnum; i++){
        mygraph->vertices[i].ops++;
        //Count the if condition below
        if (!indegree[i]) {
            queue[++rear] = i; mygraph->vertices[i].ops++;
            //Count the enqueue operation
        }
    }
    //Loop to peel the zero-indegree vertices
    while (rear >= front ){
        p = &mygraph->vertices[queue[front++]]; mygraph->vertices[queue[front-1]].ops++;
        //Count the dequeue operation above
        while( (p = p->adj) != NULL ) {
            p->ops++; //Count the if condition below
            if (!(--indegree[p->data])) {
                queue[++rear] = p->data; mygraph->vertices[p->data].ops++;
            }
        }
        mygraph->c_ops++;
    //Count the return command below
    return (front < mygraph->vexnum);
}

This code could be partitioned to 5 parts as below:

- **Declaration**: From line 3 to line 4, just do some declaration which gives 2 operations in constant.
- **Calculate Indegree**: From line 7 to line 10, initialize the indegree array with zero which gives \(V\) operations. From line 11 to line 17, it visits all the vertices to get the linked list of edges and visits all the edges to calculate the indgree for every vertex which gives \(V+E\) operations.
- **Find Zero Indegree Vertices**: From line 19 to line 26, visit all vertices to find the zero indegree vertices, which gives \(V\) operations, and insert them into the queue, which gives part of \(V\) operations.
- **Loop Peeling**: From line 28 to line 38, dequeue the vertices in the queue and decrement the indegree of all vertices with edges from the processing vertex by visiting all the edges, and enqueue the zero indegree vertices. Therefore, in whole it gives \(E\) operations for all visits of edges(just once since it will be removed after visited), and \(V\) operations of enqueue and \(V\) operations of dequeue.
- **Return Flag**: If the queue is not being full used, which means not all of the vertices are processed through the queue, the graph has a cycle and it returns the flag ture, otherwise returns flag false. This gives 1 operation in constant.

Above all, this topological algorithm should has total number of operations: \(5V + 2E + 3\). The complexity is \(O(V + E)\).

**V. Experimental Results**

By applying the program on provided inputs, I get the outputs and summerize as following:
<table>
<thead>
<tr>
<th>Input</th>
<th>Vertices</th>
<th>Edges</th>
<th>Operations of Vertices</th>
<th>Operations of Edges</th>
<th>Total Operations</th>
<th>Constant A</th>
<th>Constant B</th>
</tr>
</thead>
<tbody>
<tr>
<td>in1</td>
<td>15</td>
<td>73</td>
<td>75</td>
<td>146</td>
<td>224</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>in2</td>
<td>20</td>
<td>114</td>
<td>100</td>
<td>228</td>
<td>331</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>in3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>in4</td>
<td>40</td>
<td>488</td>
<td>200</td>
<td>976</td>
<td>1179</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>in5</td>
<td>60</td>
<td>1130</td>
<td>300</td>
<td>2260</td>
<td>2563</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>in6</td>
<td>80</td>
<td>2004</td>
<td>400</td>
<td>4008</td>
<td>4411</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>in7</td>
<td>100</td>
<td>3120</td>
<td>500</td>
<td>6240</td>
<td>6743</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Note that the in3 graph contains a cycle. Other inputs get topological order and reports the operations number as the table shown. And it is validated with analysis in section IV with complexity of $O(V + E)$, the constant $A = 5$ and constant $B = 2$ in $5V + 2E + 3$.

VI. CONCLUSION

In this project, I implement the “peeling zero indegree” topological algorithm in C and apply the program on given input graphs to find out their topological order. Analyze the code, I figure out the time complexity as $O(V + E)$. Take all C commands in sort algorithm to validate the time complexity of the algorithm, I get the total operations are $5V + 2E + 3$, which is the complexity of $O(V + E)$. As a result, the analysis and the experiment results are the same, which proves that the code and analysis are correct, as well as the commands counting.