Topological Sort Implementation

Narimount

December 2019

1 Algorithm Description

We implemented the topological sort, using **DFS** as figure 1 suggests. In order to implement **DFS**, we obtained the adjacency list of the vertices from the input text file and followed the DFS algorithm mentioned in the book. We first initialized the color of all nodes to white and the time counter to zero. We start by looking at the depth of vertices one by one. Each time a vertex is checked to see if it is already visited (Black or Gray) or not(White). Once a white node is found, the **DFS_Visit** function will be called on that node say u. During **DFS_Visit**, all the adjacent nodes to u will be examined, and each white neighbour will be visited. Once all the outgoing links of u are explored, u will be blackened and its *finishing time* will be recorded. Once a node is finished it will be added to the top of the *order list*. In order to find out if the graph is asyclic or not, we simply look for a back edge, i.e., an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by **DFS**. An standard implementation of the DFS algorithm is mentioned in the book at page 604. But in our implementation for topological sort there is no need to have the overhead of recording and storing the discovery time as we are only dealing with the finish times. Moreover, the difference between black and gray nodes can be relaxed in our implementation. These two modifications will lead to better constants in the complexity analysis. (However we have been strict on the difference between gray and black in our implementation to verify the algorithm is terminating correctly.) One can simply comment the corresponding lines to have slightly a better time complexity. The **DFS** algorithm is of $\theta(V+E)$. Using an aggregate analysis, looking at the implementation of the book, (We have performed same analysis for our implementation later on.) The initialization will be done in $\theta(V)$. The time spent finding white nodes will be done in $\theta(V)$. The overall time spent in **DFS_Visit** is $\theta(E)$ since it is called once for each node and takes $\theta(deg(u))$ once called on node u.

2 Data Structure

We use linked-list for representing the graph (and the adjacency lists). We defined an structure for each node where it holds a value for index of its desti-

nation node in the linked list and points to the structure of the next node. The graph consists of an array of adjacency lists of the nodes where the associated node is the head of the list. The time complexity of forming this structure is mentioned in detail in next sections. We could also use adjacency matrix for representing the graph. But that would introduce more complexity. We note that we do not need to make any direct queries to the adjacency object hence of course using linked lists will lead to better performance in terms of time and space complexity, although the implementation is harder.

3 Code Description and time analysis

The code consists of two parts as it is mentioned in the comments of the source file. The first part deals with parsing the data from input and forming the adjacency matrices of all nodes. As mentioned above, we prefer linked list to matrices when finding the adjacency matrix.
The second part. runs DFS and DFS_Visit almost sames as what mentioned in the book. However we have made small modifications in our implementation. When populating the files we populate file A sequentially. When a node is finished it will be immediately added to the file to slightly improve the time complexity. However B and C have to be dealt with separately after running some analysis on the results.
Please note that the following analysis is for when we are strict on the difference between gray and black for a node and that is only to show that the code is working correctly as expected. If we remove that each node will be charged once less and the time complexity improves.
The asymptotic analysis of time complexity consists of two different parts as follows:

1- **Parsing the input and forming the adjacency list**: Finding the number of vertices takes V operations. Forming the adjacency list for each vertex takes $3 * deg(V)$ operations. Overall, it will make it $3 * E$ operations. Taking into account all operations happening, in the two while loops of the first part $5E + 6V$ operations happen in the first part together with 10 constant operations.

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Figure 1: Topological Sort using DFS

Figure 2: B.txt for first input in1.txt

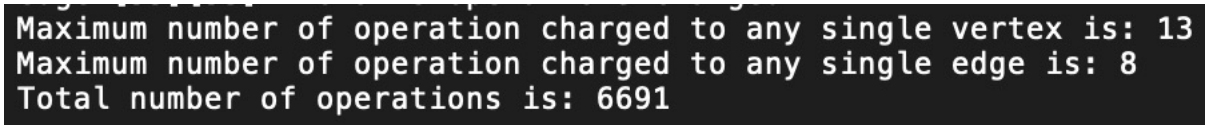

Figure 3: B.txt for first input in5.txt

2-**Running DFS and finding the topological order**: It takes 3 constant-time operations together with $2V$ operations to make the initialization for running **DFS**. Within the **DFS** for each vertex it takes once it is checked that whether the vertex is already visited or not (Functionality of color), which overall will be V operations. In **DFS_Visit**, the counter will get incremented once, the vertex will be marked as visited and a temporary structure will be initiated. Moreover, at the end of the function, the color will be updated, the finish time will be recorded and the time counter will get incremented. Moreover, For each vertex, its neighbors are checked to find out if they are already visited or not in order to move towards the depth of the adjacency list of that vertex. This will charge each edge of the graph once. Moreover, to see if the graph contains a cycle or not, at each vertex the finish-time of its neighbors are counted. If a node exists with finish-time of zero which is already visited, existence of a cycle is implied. This will charge the edges once more. The last time when a when an edge gets charged, is when actually moving towards the end of a linked-list. Given this analysis, during the second part, we may expect a vertex to be charged at most 7 times, while an edge gets charged at most 3 times.

Taking into account both parts we may expect an asymptotic run-time of $13V + 8E + 17$. Please note that in this analysis we have not considered the overhead of text-writing into the files. Moreover, the commands for calculating the overhead are not taken into account and are marked in the code. Now consider the run of the code for the first input ***in1.txt***. As figure 2, shows there is a total number of 413 operations charged to the vertices and the edges. The values that we expect based on the asymptotic analysis is:

$$T_1 = 13(12) + 8(32) + 17 = 430$$

Which is pretty close to the experimental result. For ***in5.txt***, this value is calculated as:

$$T_5 = 13(56) + 8(751) + 17 = 6736$$

Which is again very close to what is seen in figure 3.

4 Experimental Results

For the 5 outputs we have:

First input: The run-time is $1.01ms$. One found topological order is as follows: 1, 7, 8, 11, 10, 2, 6, 12, 3, 5, 4, 9
Total number of operations is: 413

Second input: The run-time is $1.105ms$. One found topological order is as follows: 13, 14, 3, 2, 1, 5, 4, 6, 10, 9, 7, 8, 11, 12
Total number of commands is: 463.

Third input: The run-time is $0.78ms$. The graph contains a cycle and no topological order is found; The program terminates once recognizing that and hence the total number of operations is only: 316.

Forth input: The run-time is $1.94ms$. One found topological order is as follows: 31, 19, 9, 30, 2, 4, 14, 6, 5, 1, 3, 10, 7, 8, 15, 16, 11, 17, 24, 12, 13, 29, 23, 20, 21, 27, 22, 25, 18, 28, 26
Total number of operations is: 1957

Fifth input: The run-time is $2.95ms$. One found topological order is as follows: 53, 47, 56, 3, 42, 33, 5, 54, 32, 21, 43, 8, 55, 9, 22, 27, 16, 17, 19, 14, 50, 45, 44, 29, 1, 15, 4, 39, 31, 23, 20, 12, 40, 11, 46, 28, 26, 36, 6, 38, 35, 2, 37, 7, 41, 51, 18, 10, 13, 24, 25, 30, 49, 34, 48, 52
Total number of commands is: 6691.