

1 Assignment 10: Depth-first search (DFS)

Allocated time: 2 hours

1.1 Implementation

You are required to correctly and efficiently implement the depth first search algorithm (DFS) (*Chapter 22.3 from [1]*). For graph representation, you should use adjacency lists. You also have to:

- Implement topological sorting (*Chapter 22.4 from [1]*)
- Implement the Tarjan algorithm for detecting strongly connected components

1.2 Minimal requirements for grading

The lack of any of the minimum requirements (even partially) may result in a lower grade through penalties or refusal to accept the assignment resulting in a grade of 0.

- *Demo*: Prepare a demonstration of correctness for each algorithm implemented. The correctness of each algorithm is demonstrated through a simple example (maximum 10 values).
- The charts created must be easy to evaluate as in grouped and added through the Profiler functions as specified by the assignment requirements. The assignment will not be evaluated if it contains a plethora of ungrouped charts. For example, the comparative analysis implies the grouping of the compared algorithms.
- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- *The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and **the correct answer to the questions from the teacher.***

1.3 Requirements

1.3.1 DFS (5p)

Demo: Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print the tree resulted from DFS

1.3.2 Topological sort (1p)

Demo: Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print a list of nodes sorted topologically (should this list be nonempty/if it is why so?)

1.3.3 Tarjan (2p)

Demo: Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print all strongly connected components of the graph

We present Tarjan's algorithm for finding strongly connected components (SCCs) in a directed graph. For background and more details see the Wikipedia entry: Tarjan's strongly connected components algorithm. The pseudocode can also be found here.

1.3.4 Analysis of the DFS performance (2p)

Since, for a graph, both $|V|$ and $|E|$ may vary, and the running time of DFS depends on both, we will make each analysis in turn:

1. Set $|V| = 100$ and vary $|E|$ between 1000 and 4500, using a 100 increment. Generate the input graphs randomly – make sure you don't generate the same edge twice for the same graph. Run the DFS algorithm for each graph and count the number of operations performed; generate the corresponding chart (i.e., the variation of the number of operations with $|E|$).
2. Set $|E| = 4500$ and vary $|V|$ between 100 and 200, using an increment equal to 10. Repeat the procedure above to generate the chart which gives the variation of the number of operations with $|V|$.

References

- [1] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

Algorithm 1 Tarjan's strongly connected components

```
1: procedure TARJAN( $G = (V, E)$ )
2:   Input: graph  $G = (V, E)$ 
3:   Output: set of strongly connected components (sets of vertices)
4:    $index \leftarrow 0$ 
5:    $S \leftarrow$  empty stack
6:   for all  $v \in V$  do
7:     if  $v.index$  is undefined then
8:       STRONGCONNECT( $v$ )
9:     end if
10:  end for
11: end procedure

12: function STRONGCONNECT( $v$ )
13:   // Set the depth index for  $v$  to the smallest unused index
14:    $v.index \leftarrow index$ 
15:    $v.lowlink \leftarrow index$ 
16:    $index \leftarrow index + 1$ 
17:    $S.push(v)$ 
18:    $v.onStack \leftarrow \text{true}$ 
19:   // Consider successors of  $v$ 
20:   for all  $(v, w) \in E$  do
21:     if  $w.index$  is undefined then
22:       // Successor  $w$  has not yet been visited; recurse on it
23:       STRONGCONNECT( $w$ )
24:        $v.lowlink \leftarrow \min(v.lowlink, w.lowlink)$ 
25:     else if  $w.onStack$  then
26:       // Successor  $w$  is in stack  $S$  and hence in the current SCC
27:       // If  $w$  is not on stack, then  $(v, w)$  is an edge pointing to
       an SCC already
28:       // found and must be ignored
29:       // See below regarding the next line
30:        $v.lowlink \leftarrow \min(v.lowlink, w.index)$ 
31:     end if
32:   end for
33:   // If  $v$  is a root node, pop the stack and generate an SCC
34:   if  $v.lowlink = v.index$  then
35:     start a new strongly connected component
36:     repeat
37:        $w \leftarrow S.pop()$ 
38:        $w.onStack \leftarrow \text{false}$ 
39:       add  $w$  to current strongly connected component
40:     until  $w = v$ 
41:     output the current strongly connected component
42:   end if
43: end function
```
