Topological Sorting

Raimondas Čiegis

Department of Mathematical Modelling, email: rc@vgtu.lt

April 20 d., 2025

In this lecture we consider one more important sorting task.

In this lecture we consider one more important sorting task.

We already know a few very fast sorting algorithms!

Why to consider new algorithms?

In this lecture we consider one more important sorting task.

We already know a few very fast sorting algorithms!

Why to consider new algorithms?

My answers:

1. The problem of topological sorting looks similar to classical sorting prroblems, but the similarity is only superficial.

2. The new sorting algorithms can be desribed most efficiently by using graph theory.

We will discuss the tasks that often have to be solved when creating schedules.

We will discuss the tasks that often have to be solved when creating schedules.

1. Transport schedules, when we aim to coordinate the schedules of different buse routes so that passengers can use connecting routes of buses and the duration of transfers should be as short as possible.

We will discuss the tasks that often have to be solved when creating schedules.

1. Transport schedules, when we aim to coordinate the schedules of different buse routes so that passengers can use connecting routes of buses and the duration of transfers should be as short as possible.

2. Many processes in the modern economy and digital technologies consist of a large number of intermediate operations and it is necessary to guarantee the correct sequence of these actions.

We will discuss the tasks that often have to be solved when creating schedules.

1. Transport schedules, when we aim to coordinate the schedules of different buse routes so that passengers can use connecting routes of buses and the duration of transfers should be as short as possible.

2. Many processes in the modern economy and digital technologies consist of a large number of intermediate operations and it is necessary to guarantee the correct sequence of these actions.

3. Parallel computing, when different algorithmic tasks are performed on different processors (cores). Again, we must guarantee the correct order (dependency) of the execution of different computational jobs. For example, this is how images are generated on mobile phones, when you watch a movie or play an interesting online computer game.

We have a set of tasks (vertices)

$$U = (u_1, u_2, \dots, u_N),$$

We have a set of tasks (vertices)

$$U = (u_1, u_2, \dots, u_N),$$

and a set which defines a linear ordering of vertices
(a set of graph edges)

$$C = (u_{i_1} \prec u_{j_1},\ u_{i_2} \prec u_{j_2},\ \dots,\ u_{i_M} \prec u_{j_M}).$$

We have a set of tasks (vertices)

$$U = (u_1, u_2, \dots, u_N),$$

and a set which defines a linear ordering of vertices
(a set of graph edges)

$$C = (u_{i_1} \prec u_{j_1},\ u_{i_2} \prec u_{j_2},\ \dots,\ u_{i_M} \prec u_{j_M}).$$

Here the connection (an edge which connects two vertices)

$$u_{i_k} \prec u_{j_k}$$

defines, that the vertex u_{j_k} can be visited if and only if
all its dependencies u_{i_k} are already visited (tasks are finished).

A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

We want to order all elements of the set U in the way

$$U^{'} = (u_1^{'}, u_2^{'}, \dots, u_N^{'}),$$

that all connections defined in C are satisfied:

A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

We want to order all elements of the set U in the way

$$U^{'} = (u_1^{'}, u_2^{'}, \ldots, u_N^{'}),$$

that all connections defined in C are satisfied:

If a relation

$$u_i^{'} \prec u_j^{'}$$

is specified, then we have the ordering of elements $i < j$.

We will notice that these requirements often do not define a single sorted set and we can find several solutions.

This problem is called topological sorting.

Planting trees

We need to plant three trees.

Planting trees

We need to plant three trees.

For planting the j-th tree let us denote

by d_j – the task of digging a hole,

by p_j – the task of planting the tree in the hole,

by u_j – the task of filling the hole.

Planting trees

We need to plant three trees.

For planting the j-th tree let us denote

by d_j – the task of digging a hole,

by p_j – the task of planting the tree in the hole,

by u_j – the task of filling the hole.

Then we have a set of nine tasks

$$U = \left(d_1, d_2, d_3, p_1, p_2, p_3, u_1, u_2, u_3 \right).$$

Planting trees

We need to plant three trees.

For planting the j-th tree let us denote

by d_j – the task of digging a hole,

by p_j – the task of planting the tree in the hole,

by u_j – the task of filling the hole.

Then we have a set of nine tasks

$$U = \big(\, d_1, \, d_2, \, d_3, \, p_1, \, p_2, \, p_3, \, u_1, \, u_2, \, u_3 \, \big).$$

The order of the tasks is defined by a natural set of relations:

$$C = \big(\, d_j \prec p_j, \;\; p_j \prec u_j, \;\; j = 1, 2, 3 \, \big) \, .$$

It is possible to construct a few topologically sorted sets of tasks.

For example we can plant each tree separately, then the following set of tasks is used:

$$U' = \big(\, d_1, \, p_1, \, u_1, \;\; d_2, \, p_2, \, u_2, \;\; d_3, \, p_3, \, u_3 \, \big) \, .$$

It is possible to construct a few topologically sorted sets of tasks.

For example we can plant each tree separately, then the following set of tasks is used:

$$U' = (\, d_1,\, p_1,\, u_1, \;\; d_2,\, p_2,\, u_2, \;\; d_3,\, p_3,\, u_3 \,)\,.$$

We can divide the work in another way, first we have to dig all three holes, plant the trees in them, and then bury all three holes (again, the order of the individual trees does not matter):

$$U'' = (\, d_1,\, d_2,\, d_3, \;\; p_3,\, p_2,\, p_1, \;\; u_2,\, u_3,\, u_1 \,)\,.$$

A completely unfortunate, but still not rarely seen option, when the work begins in this order

$$U^* = \left(\, d_1, \, d_2, \, d_3, \;\; u_2, \, u_3, \, u_1 \, \right).$$

A completely unfortunate, but still not rarely seen option, when the work begins in this order

$$U^* = \left(\, d_1, \, d_2, \, d_3, \;\; u_2, \, u_3, \, u_1 \, \right).$$

Two teams are happy that they have done their part of the job, but the total result is ...

Now we will present a rigorous formulation of the topological sorting problem.

We have a directed graph $G = (V, E)$.
Here V is the set of vertices of the graph, and E is the set of edges of the graph, the edges have directions.

Let's assume that the graph G has no cycles.

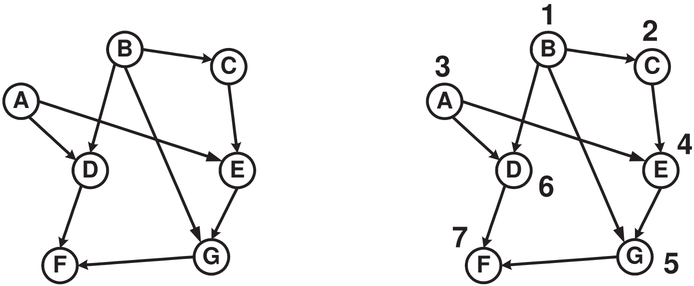
Topological Sorting

The vertices of the graph must be labeled so that each edge connects a lower-numbered vertex to a higher-numbered vertex.

Topological Sorting

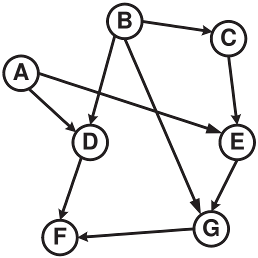
The vertices of the graph must be labeled so that each edge connects a lower-numbered vertex to a higher-numbered vertex.

An example of a solution to a topological sorting problem is presented in the figure.

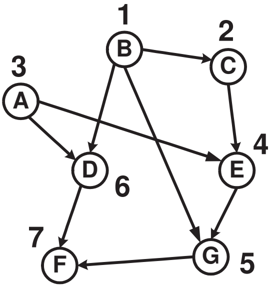


a) the initial unsorted graph

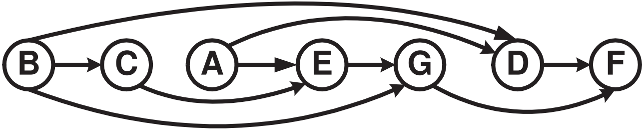
b) a sorted graph



a) the initial unsorted graph



b) a sorted graph



c) a line ordering of vertices of the graph.

Depth-first search method

We will see that we can solve this new and complex sorting problem simply by specifically choosing the order in which the vertices of the graph are visited.

Depth-first search method

We will see that we can solve this new and complex sorting problem simply by specifically choosing the order in which the vertices of the graph are visited.

Let us remember the printing of an arithmetic expression in the three basic forms: prefix, infix, and postfix. This task was solved by properly chosen recursive algorithms for traversing the vertices of a binary tree (which is also a particular type of a graph).

Depth-first search method

We will see that we can solve this new and complex sorting problem simply by specifically choosing the order in which the vertices of the graph are visited.

Let us remember the printing of an arithmetic expression in the three basic forms: prefix, infix, and postfix. This task was solved by properly chosen recursive algorithms for traversing the vertices of a binary tree (which is also a particular type of a graph).

The Depth-first search strategy is simple: from a given graph vertex we go to the adjacent vertex, that has not yet been visited during this search procedure.

If there are no such vertices, we take one step back and look for a new path from the parent vertex. This way we find all the vertices that can be reached from the chosen starting vertex.

If there are no such vertices, we take one step back and look for a new path from the parent vertex. This way we find all the vertices that can be reached from the chosen starting vertex.

If the graph is not connected, then we repeat the algorithm, taking a new, unvisited starting vertex.

If there are no such vertices, we take one step back and look for a new path from the parent vertex. This way we find all the vertices that can be reached from the chosen starting vertex.

If the graph is not connected, then we repeat the algorithm, taking a new, unvisited starting vertex.

Since we visit the most distant vertices first during the search, we call this method depth–first search algorithm.

Now we will provide details of the depth-first search algorithm.

Now we will provide details of the depth-first search algorithm.

Each vertex of the graph can be in one of three states (we will denote the states by different colors).

Now we will provide details of the depth-first search algorithm.

Each vertex of the graph can be in one of three states (we will denote the states by different colors).

Initially, all vertices are *unvisited* and are colored white.

Now we will provide details of the depth-first search algorithm.

Each vertex of the graph can be in one of three states (we will denote the states by different colors).

Initially, all vertices are *unvisited* and are colored white.

When a vertex v is visited for the first time, it becomes *not new* and is painted gray.

We store the time when it became not new in the array element $d(v)$ (*discovered*).

Now we will provide details of the depth-first search algorithm.

Each vertex of the graph can be in one of three states (we will denote the states by different colors).

Initially, all vertices are *unvisited* and are colored white.

When a vertex v is visited for the first time, it becomes *not new* and is painted gray.

We store the time when it became not new in the array element $d(v)$ (*discovered*).

A vertex is painted black when all edges exiting it have been examined. Such vertices are called *finished*.

The moment when the vertex became black is stored in the array element $f(v)$ (*finished*).

We store the search paths in an array π, the value of its element $\pi(v)$ gives the vertex u from which we first visited v, i.e.

$$\pi(v) = u.$$

Depth-first search algorithm

DepthFirstSearch (G)
begin
(1) **for** ($v \in V$) **do**
(2) color(v) = *white*
(3) $\pi(v) = \text{NULL}$
end do
(4) t = 0
(5) **for** ($u \in V$) **do**
(6) **if** (color(u) == *white*) **then**
(7) DFS_Visit(u)
end if
end do
end DepthFirstSearch

Recursive *DFS_Visit* algorithm

DFS_Visit (u)
begin
(2) color(u) = *grey*
(3) t = t +1, d(u) = t
(4) **for** ($v \in N(u)$) **do**
(5) **if** (color(v) == *white*) **then**
(6) $\pi(v) = u$
(7) DFS_Visit(v)
end if
end do
(8) color(u) = *black*
(9) t = t +1, f(u) = t
end DFS_Visit

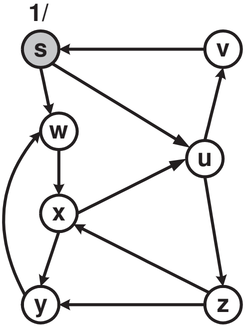
Estimates of the complexity of this topological sort algorithm

For each vertex $v \in V$ we execute the *DFS_Visit* procedure once and the algorithm is repeated as many times as this vertex has neighbors. Therefore, the total size of basic operations for the topological sorting algorithm is equal to

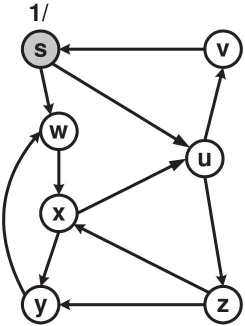
$$\Theta(|V| + |E|).$$

Let us consider the following directed graph:

Let us consider the following directed graph:



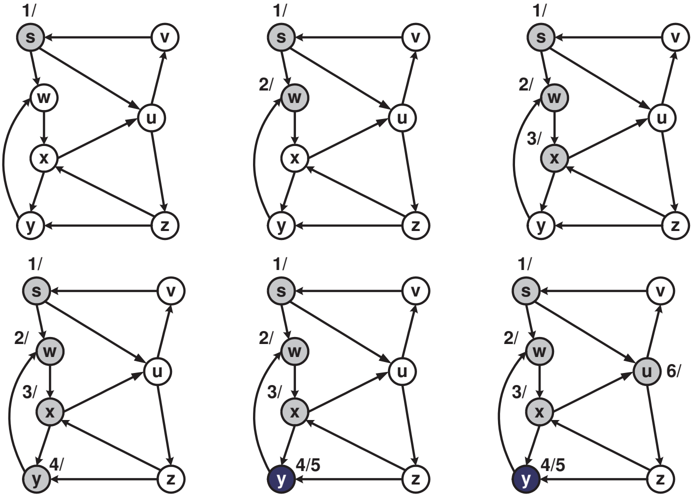
Let us consider the following directed graph:

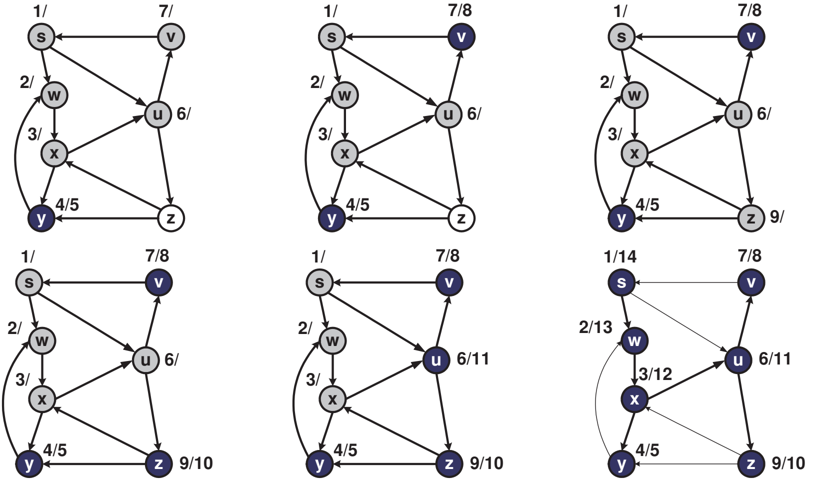


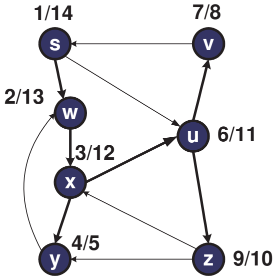
Can we sort it topologically?

We visit the vertices of the graph using the depth-first search method. The process of visiting the vertices after each call to the *DFS_Visit* function is shown in the figure. The vertices are given the values of $(d(v), f(v))$.

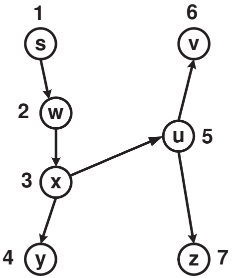
We visit the vertices of the graph using the depth-first search method. The process of visiting the vertices after each call to the *DFS_Visit* function is shown in the figure. The vertices are given the values of $(d(v), f(v))$.




1/
7/
s
v
2/
w
u
6/
3/
x
4/5
y
z
1/
7/8
s
v
2/
w
u
6/
3/
x
4/5
y
z
1/
7/8
s
v
2/
w
u
6/
3/
x
4/5
y
z
9/
1/
7/8
s
v
2/
w
u
6/
3/
x
4/5
y
z
9/10
1/
7/8
s
v
2/
w
u
6/11
3/
x
4/5
y
z
9/10
1/14
7/8
s
v
2/13
w
u
6/11
3/12
x
4/5
y
z
9/10



a) visted vertices

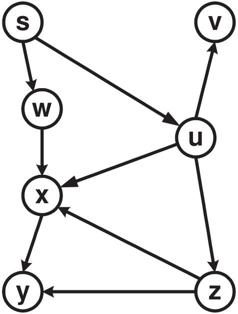


b) the order of visits

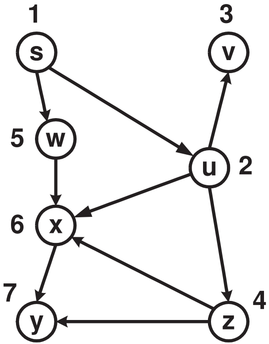
To obtain a sorted set of vertices, we modify the *DFS_Visit* procedure, at the end of which we insert the vertex u. We put it at the beginning of the linear list (it is enough to use a stack):

DFS_VisitSort (u)
begin
(2) color(u) = *grey*;
(3) t = t +1, d(u) = t;
(4) **for** ($v \in N(u)$) **do**
(5) **if** (color(v) == *white*) **then**
(6) $\pi(v) = u$;
(7) DFS_VisitSort(v);
end if
end do
(8) color(u) = *black*;
(9) t = t +1, f(u) = t;
(10) List.InsertHead (u);
end DFS_VisitSort

Topological Sorting : Example



a)



b)