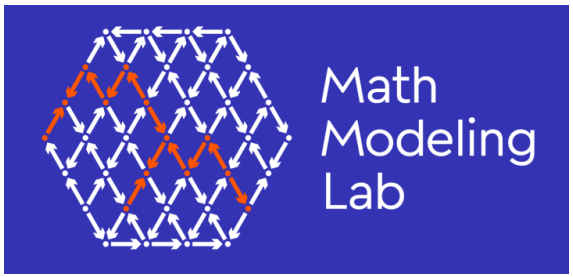
Lecture 8. Directed graphs and Depth First Search

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

May 11, 2023



Overview

1. Directed Graphs
2. Depth First Search and Topological Sorting
3. Finding Strongly Connected Components
4. Applications

Directed Graphs Definitions

• In this lecture we apply specific notation for directed graphs: the set V of vertices and E of *arrows* and maps $h, t : E \to V$ such that $h(e)$ is the head and $t(e)$ the tail of an arrow.
• The *incoming/outgoing* arrows of a vertex v are stored in the dynamic array $In[v]/Out[v]$. In most cases, to store $Out[v]$ is enough and saves memory.
• Recall that undirected graphs can be converted to directed ones by defining two arrows of converse directions per an edge of the undirected graph. This way, many concepts below like e.g. DFS walk apply to undirected graphs as well.
• A *directed path* is a sequence $e_0, e_1, \ldots, e_k$ of arrows such that for $i = 1, \ldots, k$ holds $h(e_{i-1}) = t(e_i)$. The path is called *a cycle* if $h(e_k) = t(e_0)$.
• A graph is called *strongly connected* if for every $u, v \in V$ there is a directed path $(e_0, \ldots, e_k)$ such that $u = t(e_0), v = h(e_k)$. A graph with 1 vertex is also regarded as strongly connected.
• A subgraph $H(U, F) \subseteq G(V, E)$ is a subset $F \subseteq E$ of arrows such that for every $f \in F$ holds $h(f), t(f) \in U$. H is called *full subgraph*, if for every $e \in E$ such that $h(e), t(e) \in U$ holds $e \in F$. Such a full subgraph is fully determined by its vertex subset $U \subseteq V$.

Strongly connected components of a Directed graph

Theorem 1

The set of all strongly connected full subgraphs in G has maximal elements $C_1, \ldots, C_q$ and for every $1 \leq i < j \leq q$ holds $C_i \cap C_j = \emptyset$.

Proof: If $U_1, U_2 \subseteq V$ are such that the corresponding full subgraphs are strongly connected and $v \in U_1 \cap U_2$, then for any $u_1, u_2 \in U_1 \cup U_2$, v is reachable from u_1 and u_2 is reachable from v, hence, $U_1 \cup U_2$ also defines a strongly connected full subgraph. By definition, every vertex defines a strongly connected subgraph, hence the set of maximal strongly connected subgraphs is non-empty. □

Corollary 2

1. *The maximal strongly connected full subgraphs $C_1, \ldots, C_q$ are unique.*
2. *Every directed cycle belongs to one of $C_1, \ldots, C_q$.*

Definition 3

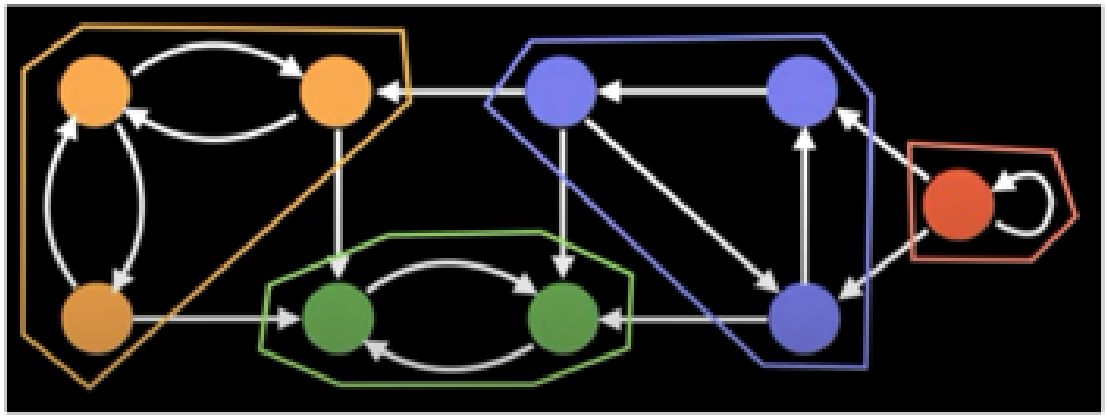
The subgraphs $C_1, \ldots, C_q$ from Theorem 1 are called *strongly connected components* of G.

Strongly connected components and condensation

Lemma 4

For a directed graph $G(V, E)$ with strongly connected components $C_1, \ldots, C_q$ consider a graph G' that has vertices $1, \ldots, q$ and an arrow $i \to j$ for every arrow $e \in E$ such that $t(e) \in C_i, h(e) \in C_j$. Then G' has no directed cycles.

Proof: follows from Corollary 2.2. □



2	0	0	0
2	4	0	0
2	1	3	0
0	0	2	1

Definition 5

The graph G' from Lemma 4 is called *condensation* of G.

Direct Acyclic Graphs and Linear Order

Definition 6

G is called a Directed Acyclic Graph (DAG) if it contains no directed cycles.

Definition 7

A *linear order* on a directed graph $G(V, E)$ is a bijection $\mathtt{i} : V \to \{1, \ldots, n\}$. Linear order is called *consistent* with G if for every $e \in E$ holds $\mathtt{i}(t(e)) > \mathtt{i}(h(e))$.

Theorem 8

G has a consistent linear order if and only if G is a DAG.

Proof: "Only if" is clear: if $(u_1, \ldots, u_k, u_1)$ is a cycle, then $u_1, \ldots, u_k$ can't be ordered. If G is a DAG, then we claim that V contains a *sink* $v \in V$ such that $Out[v] = \emptyset$. Indeed, assuming no sink exist, we build a path from any vertex, using *any* outgoing arrow until reach one of previous vertices, thus, yield a cycle. If v is a sink, we set $\mathtt{i}(v) = 1$ and reduce the statement to the full subgraph on $V \setminus \{v\}$ and apply induction.

Topological Sorting

By definition 7, a consistent linear order is such that head vertices precede the tail ones, for each arrow. So a consistent linear order is a completion of the partial order on V encoded in E; finding it is similar to sorting and therefore is called **Topological Graph Sorting**. Theorem 8 yields us a method:

Sequential Topological Sorting: Input: a graph $G(V, E)$, $|V| = n$. Output: 1/0 according to whether G is a DAG or not, and if it is, the consistent linear order `i`

1. for $step = 1 : n$ `i`$[step] := 0; col[step] := 0;$ //initialization
2. for $step = 1 : n$
3. $s := -1$;//looking for a sink s in the residual graph
4. for $v \in V, col[v] = 0$//check the vertices not yet included to `i`
5. $IsSink := 1$;
6. for $u \in Out[v]$ if $(col[u] = 0)$
7. $IsSink := 0$; break//$v \to u$ goes to residual vertex, v is not sink
8. if $(IsSink)$ $s := v$; break
9. if $(s > 0)$//did break in **8**, s is a sink
10. `i`$[step] := s$; $col[s] := 1$;//add s to `i`, exclude from the graph
11. else return 0 //found no sink, hence, a subgraph with a cycle
12. return 1,`i`

Sequential Topological Sorting property

Theorem 9

Sequential Topological sorting finds a consistent linear order in $O(|V|^2 + |V||E|)$, if G is a DAG and returns 0, otherwise.

Proof: First, check that the algorithm works correctly, following Theorem 8. First of all, the usage of color array allows to deal, in each step of **1** loop, with the full subgraph on the vertices colored in 0, and every step decreases the number of such vertices. This is exactly how the inductive proof of Theorem 8 looks like. At a step, in **4,6**, we only take vertices with color 0 into account. Now, if no sink occured in the residual subgraph, then as we show in the proof of Theorem 8, this subgraph contains a cycle. Otherwise, we put the found sink after the previous ones, to `i`, and color it in 1, so exclude out of the graph to be considered next. The complexity follows from the fact that the loop **1** has $|V|$ steps and at each loop step we run over all vertices checking the color, and check the arrows going out of vertices colored in 0, so totally check at most $|E|$ arrows. □

Depth First Search as a Graph Traverse

We are already familiar with DFS as the main enabler of efficient algorithms on Binary Trees, and since a Binary tree is a partiucular example of a directed graph, the below definition of DFS should be clear:

Definition 10

Informally Depth First search is a graph walk that goes down to the arrow at the head of arrow before passing to the next arrow going out of the tail vertex. More specifically DFS is a recursion that takes a vertex v as input and then calls itself for every non-visited vertex $u \in Out[v]$.

Remark 1. Same as BFS, DFS is a graph walk concept rather than a specific algorithm. Unlike of our thinking, in the case of BFS, popular textbooks treat it as a specific algorithm that we called BFSTraverse, and which founds the shortest paths with respect to the path length defined as the number of hops. But in the case of DFS, the textbooks treat it as a concept, same as us.

Remark 2. As we mentioned above, undirected graphs are a particular case of directed ones, so DFS applies to these, too.

Depth First Search Details

Before introducing DFS, let's present its overall structure including:
A) The main method that we call **DFSTraverse** (a.k.a. DFS-Visit, Exlore etc.) has the role similar to BFSTraverse, that is, starting from a root vertex, to explore the paths to get to every reachable vertex.
B) A flag $Visited[v]$ of the vertex $v \in V$ that is false until the vertex is visited and flag assigned with true.
C) An array $Prev$ such that $Prev[v]$ is the vertex such that v was explored as a non-visited vertex in $Out[Prev[v]]$. This concept is similar to BFS.
D) A simple outer loop **DFSMain** that calls DFSTraverse for each vertex that hasn't been reached by previous calls of DFSTraverse.
E) The auxillary concept of **Clock** increasing on every step of DFSTraverse. Each vertex gets assigned with two **Timestamps** $s[v], f[v]$ corresponding to the moments of color changes. The usage of $s[v], f[v]$ will become clear in further applications of DFS.
F) Two functions,**Previsit, Postvisit** handle $s[v], f[v]$ and are a convenient place to add specific code handling further applications of DFS.

Depth First Search Metacode

To simplify the description of several functions. let's assume that a directed graph $G(V, E)$ and current *Clock* have a global visibility. Instead of global variable, *Clock* can be passed as parameter to methods with the right to change.

Previsit : Input: $v \in V$: $Clock := Clock + 1$; $s[v] := Clock$;

Postvisit: Input: $v \in V$: $Clock := Clock + 1$; $f[v] := Clock$;

DFSTraverse: Input: $v \in V$

1. $Visited[v] := true$;
2. $Previsit(v)$;
3. for $u \in Out[v]$
4. if $(visited[u] = false)$
5. $prev[u] := v$; $DFSTraverse(u)$;
6. $Postvisit(v)$;

DFSMain:

1. for $v \in V$: $Visited[v] := false$; $Prev[v] := 0$; $s[v] := f[v] := 0$; //initialization
2. $Clock := 0$;
3. for $v \in V$
4. if $(Visited[v] = false)$
5. $DFSTraverse(v)$;

What is *DFSTraverse* Result?

Lemma 11

Let $v \in V$ and $N \subseteq V$ be all vertices u with $Visited[u] = false$ at $Clock = s[v]$. Then $DFSTraverse(v)$ builds a subtree with root v and $prev[u] \to u$ arrows and this subtree consists of all vertices in N reachable from v by a path.

Proof: The *Visited* flag prevents from twice visiting a vertex, so the graph of $prev[u] \to u$ arrows will be a tree and only reachable vertices from N be visited. To check that *every* reachable vertex $u \in N$ is visited, consider a path $v \to \ldots \to u$ and find an edge e on it such that $Visited[t(e)] = true$ and $Visited[h(e)] = false$, in contradiction to the algorithm. □

Remark 1: In partucular, the first call of *DFSTraverse* will build a tree covering all vertices reachable from its root in G. The further calls of *DFSTraverse* from *DFSMain* may not visit some reachable vertex, if it has been visited previously.

Remark 2: The *Visited* flag is necessary at least for the cases, when G has cycles: allowing to visit vertices again would lead to an infinite loop.

Depth First Search Complexity and Arrow Type

Lemma 12

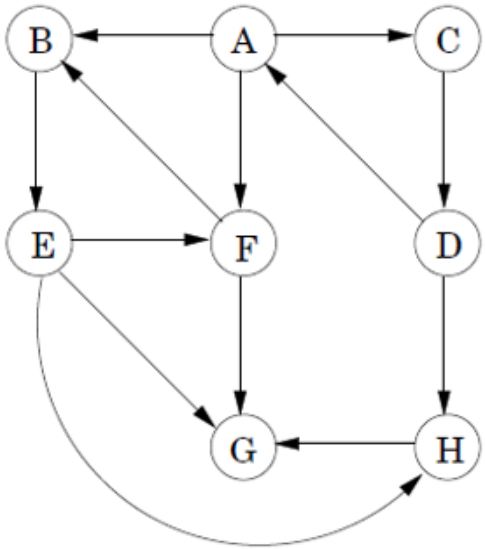
DFSMain builds a forest of vertex disjoined rooted trees covering the whole V in $O(|V| + |E|)$ complexity.

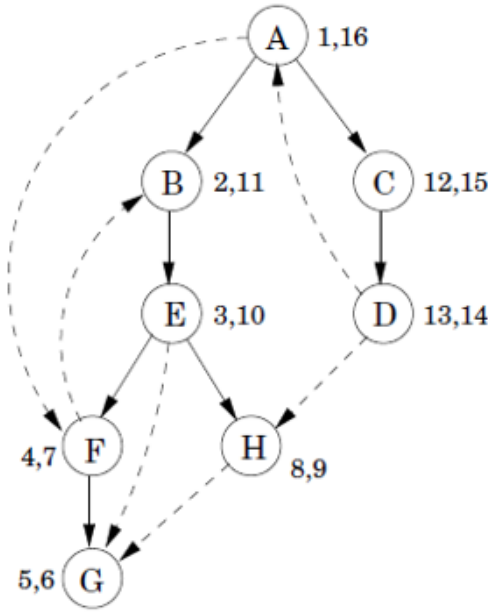
Proof: During every call of *DFSTraverse* from *DFSMain* starting from a non-visited vertex as a root, a tree is built by construction. Because we never touch a visited vertex, all trees are disjoint and obviously cover the whole V. Each $DFSTraverse(v)$ has complexity $|Out[v]|$, if not counting that of recurisive call, so totally, $O(|E|)$ and *DFSMain* visits every vertex, so $O(|V| + |E|)$. □
The tree forest emerging of *DFS* allows to classify all arows, as follows:

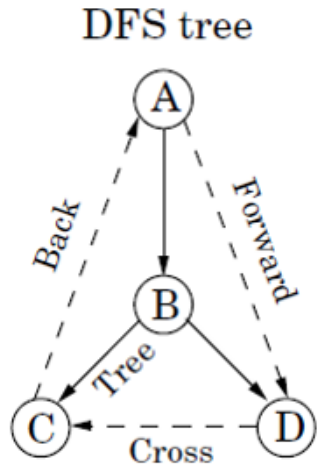
Definition 13

Tree arrows are those $prev[v] \to v$ found in *DFS*. *Forward* go from a vertex to a descendant in a tree, except for the children in the tree. *Backward* go to an ancestor in the tree. Other arrows are called *Cross*.

DFS Example







DFS Forest (Tree), $s[v], f[v]$ for every $v \in V$, forward and backward arrows.
Assumption: the vertices in $Out[v]$ are ordered alphabetically.

Arrow Type and Time intervals

Lemma 14

1. *For any* $v, w \in V$ *exactly one of the three statements holds:*
a. *Intervals* $[s[w], f[w]]$ *and* $[s[v], f[v]]$ *do not intersect*
b. $[s[w], f[w]]$ *contains* $[s[v], f[v]]$ *and* v *is a descendant of* w *in a DFS tree.*
c. $[s[v], f[v]]$ *contains* $[s[w], f[w]]$ *and* w *is a descendant of* v *in a DFS tree.*
2. *Moreover, in case* **a** *if* $w \to v \in E$ *then* $f[v] < s[w]$.

Proof: **1:** From the code it is clear that if w is a descendant of v, then the call of $DFSTraverse(w)$ is recursive from line **5** of $DFSTraverse(v)$ for one of $u \in Out[v]$, when $Clock > s[v]$ and $f[v]$ will be assigned after the recursion ends up, so after $f[w]$. Conversely, if the interval for w is inside that for v, then $s[w], f[w]$ can't be assigned before executing line **3** or after executing line **6** of $DFSTraverse(v)$, so it has happened within $DFSTraverse(u)$ for one of $u \in Out[v]$.
2: If $w \to v \in E$, then $Visited[v] = true$ when executing lines **3-5** of $DFSTraverse$, otherwise, v would become a descendant of w. □

DFS and Topological Sort

Theorem 15

1. *G is a DAG if and only if it has no backward arrows.*
2. *If G is a DAG, then $f[t(e)] > f[h(e)]$ for any $e \in E$.*
3. *If G is a DAG, then ascending order on f timestamp is consistent.*

Proof: 1. A backward arrow $u \to v$ together with the path from v to its tree descendant u is a cycle. Conversely, let $v_0 \to v_1 \to \ldots \to v_k \to v_0$ be a cycle. Without a loss of generality we may assume that $s[v_0]$ is minimal out of $s[v_i], i = 0, \ldots, k$. This means that, in the time slot $s[v_0] - 1$, when $DFSTraverse(v_0)$ is called, other vertices in the cycle haven't yet been visited. Hence, by Lemma 11, all $v_i, i > 0$ are descendants of v_0 in a DFS tree, so $v_k \to v_0$ is indeed a backward arrow. **2.** The statement is straightforward Lemma 14.1 for forward arrows. By **1**, no backward arrows occur. For the cross arrows this is Lemma 14.2. **3.** follows from **2.** □

DFS and Topological Sort Implementation

The Topological sorting can be performed based on *DFSMain* with adding some $O(1)$ additional steps described below. To check that the graph is a DAG, by Lemma 15.1 it is enough to check that no backward arrows occur. For this after lines **4-5** in *DFSTraverse* implementation above we add 2 more lines:

4. if ($visited[u] = false$)
5. $prev[u] := v$; $DFSTraverse(u)$;
6. else if $IsBackward(v \to u)$
7. $IsDAG := 0$

where *IsDAG* is either a global variable returned at the end of the sorting or a local one that allows to break all loops and return 0 whenever a cycle is found.

IsBackward: Input is an arrow $v \to u \in E$. Output: true iff $v \to u$ is backward:
return $Visited[u]$ and $s[u] < s[v]$ and $f[u] = 0$

Remark: Condition $f[u] = 0$ in *IsBackward* means $DFSTraverse(u)$ didn't finish. By Lemma 15.3, the ascending order on f label is consistent. To get it we initialize a counter $j := 0$ in *DFSMain* and add $j := j + 1$; $\mathtt{i}[j] := v$; to *Postvisit*.

Theorem 16

[D.Knuth, 1968] Topological Sorting based on DFS has complexity $O(|V| + |E|)$.

Finding Strongly Connected Components

• We introduced above the concept of strongly connected components and the question arises, how to compute these?
• A straightforward approach is to start from a vertex v, and identify its component $W \ni v, W \subseteq V$, then reduce the problem to the full subgraph on $V \setminus W$. But how to compute W?
• By Lemma 11, by vanishing *Visited* and calling *DFSTraverse* from v, one can get all vertices reachable from v. Then searching back from these vertices, one can find the components, in principle, but the emerging algorithm does not look to be very efficient.
• It was therefore a surprise when Robert Tarjan had given in [R.Tarjan, 1972] the first linear compexity, $O(|V| + |E|)$ algorithm finding all strongly connected components.
• We however introduce another algorithm, also linear, referring to [Aho,Hopcroft,Ullman, 1983].

Strongly Connected Components Algorithm

Definition 17

Let $G(V, E)$ be a directed graph. The *transposed* graph G^T is the one with transposed adjacency matrix, $G^T(V, E^T)$, where $E^T = \{u \to v | v \to u \in E\}$.

StronglyConnectedComponents: Input is G, output - components of G.

1. `i` = *DFSTopologicalSorting*(G);//find ascending order of $f[v]$

2. Sort V in the reverse order to `i`

3. Build G^T, where V is sorted as above.

4. *DFSMainSCC*(G^T);

DFSMainSCC called in line **4** is *DFSMain* with the following addendum:

1. for $v \in V$ *Visited*$[v]$:= *false*; *Prev*$[v]$:= 0; //initialization

2. *SCCList* dynamic array of vertex sets

3. *Clock* := 0;

3. for $v \in V$

4. if (*Visited*$[v]$ = *false*)

5. *DFSTraverse*(v); *SCCList.pushback*$(BFS(G, v))$;

//*BFS* collects the vertices of the tree with root v and *prev*$[u] \to u$ edges

Strongly Connected Components Algorithm Example

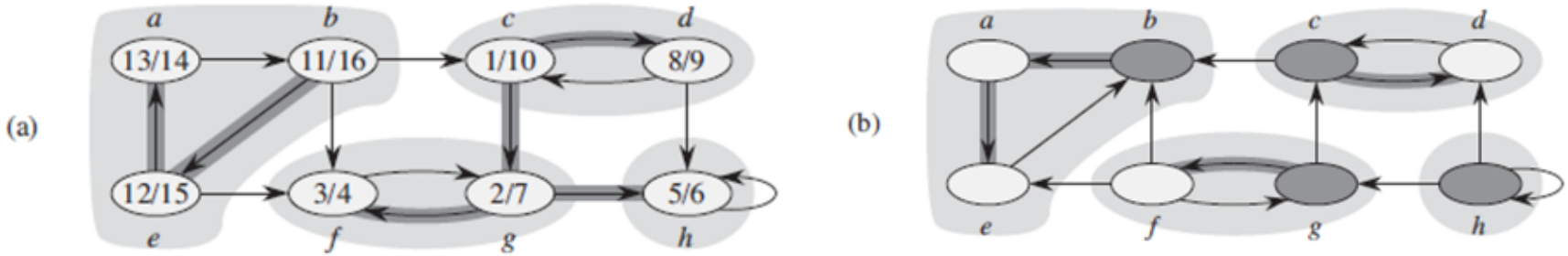


Fig. (a) shows graph G and the result of $DFSMain(G)$: $s[v]/f[v], v \in V$ and shadowed tree arrows. Also shadowed the strongly connected components.
Fig. (b) shows the transposed graph G^T, the root vertices of the trees built by $DFSMainSCC(G^T)$ (b, c, g, h) in grey, and tree arrows.

Why does the SCC Algorithm work?

• Before describing the features of the algorithm, let's consider the particular example of a DAG. In that case every vertex constitute a separate SC component.
• Since G is a DAG, by Theorem 15, `i` is consistent for G, hence, *DFSMainSCC* considers vertices in an order consistent for G^T. Therefore, the first vertex is sink in G^T so DFS tree is that vertex only, and same for others.

Lemma 18

1. *If u, v belong to the same SC component W, then every intermediate vertex on a path from u to v belongs to W.*
2. *Every SC component belongs to one tree build by DFS for G.*
3. *SC components of G and G^T are the same.*

Proof: 1. Every intermediate vertex has a path from $u \in W$ and to $v \in W$, and by definition, W is a maximal subset with mutual connectivity of each two vertices. So the intermediate vertices belong to W. **2.** Let W be a SC component and consider the earliest moment (value of *Clock*) when we call *DFSTraverse*(v) for any $v \in W$. At that moment all vertices in $W \setminus \{v\}$ aren't yet visited and by definition, are reachable from v, hence, by Lemma 11 W will be covered by the tree with root v build by *DFSTraverse*(v). **3.** is obvious.

The f values on SC components

For a subset $W \subseteq V$ denote by $s(W)$ (resp. $f(W)$) the minimum (maximum) of $s[w]$ ($f[w]$) for $w \in W$.

Lemma 19

Let $W_1 \neq W_2$ be two SC components and $e \in E$ such that $t(e) \in W_1, h(e) \in W_2$. Then $f(W_1) > f(W_2)$.

Proof: By Lemma 18.2 for some DFS trees, T_1, T_2 hold $W_1 \subseteq T_1$, $W_2 \subseteq T_2$. If $T_1 \neq T_2$, then by Lemma 14.2 we have more: $f(W_2) \leq f(T_2) \leq s(T_1) \leq f(W_1)$.
Now assume $T_1 = T_2$ and let w be the vertex in either W_1 or W_2 with $s[w] = MIN(s(W_1), s(W_2))$. If $s(W_1) < s(W_2)$, then by Lemma 11 the subtree with root w includes both W_1 and W_2; indeed, both W_1, W_2 aren't visited at $Clock = s[w]$ and arrow e makes both W_1, W_2 reachable from w. Hence, $f[w]$ is the maximum of f in the subtree, in particular, $f[w] = f(W_1) > f(W_2)$.
Finally, if $s(W_2) < s(W_1)$ then there is no a path from W_2 to W_1, because e is a path in reverse direction. Hence, DFS from w will finish at $Clock = f(W_2)$ without visiting W_1, so $f(W_1) > f(W_2)$ in this case, too. □

The SCC Main Theorem

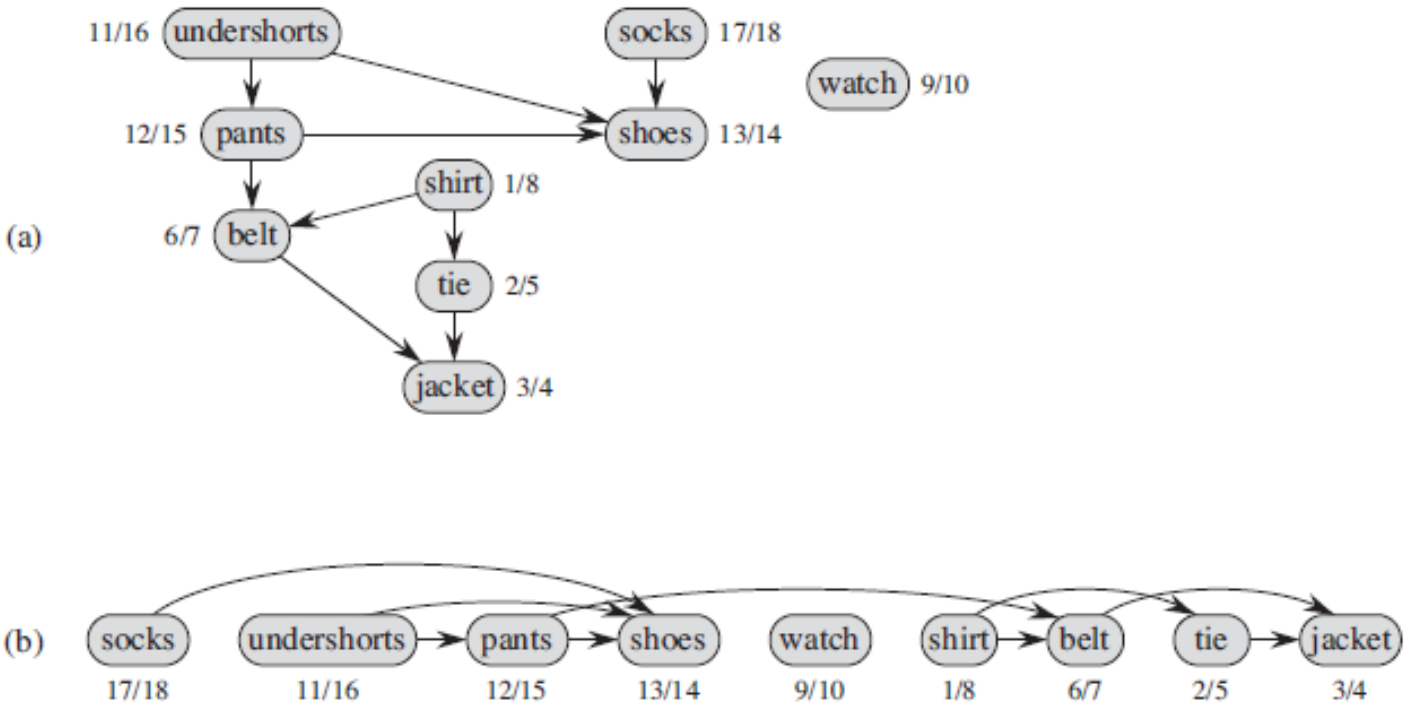
Theorem 20

The Strongly Connected Component Algorithm returns correct SC components in $O(|V| + |E|)$.

Proof: We need to prove that every tree resulting from the call of *DFSTraverse* from *DFSMainSCC* is a SC component. By Lemma 18.2 and .3 it is a union of some components. We then apply induction on the tree sequence. The first tree has as root the vertex w with maximal $f[w]$ so the SC component W of w has maximal $f(W)$. By Lemma 19, in G there are no arrows from other components to W, which means that in G^T there are no arrows from W to other components, hence, the first DFS tree is just W. This is the induction basis. Inductive step is that we consider *DFSTraverse* from $w \in W$ and all previous trees are one SC component each with $f > f(W)$. Similarly to above, there are now arrows in G^T to the components with $f < f(W)$ and those components and W aren't visited at the moment of *DFSTraverse*(w). So the arrows to other components are only those to already visited ones, and DFS tree with root w includes W only. The complexity estimate follows from that for Topological Sorting and that additional efforts in *DFSMainSCC* are all $O(|V| + |E|)$.

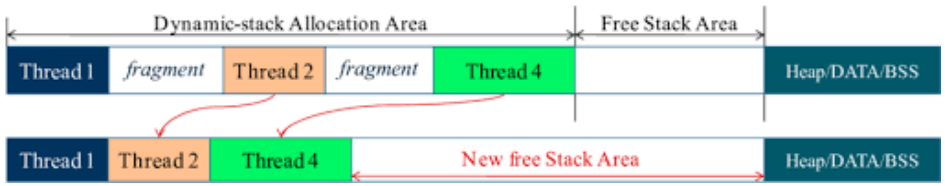
Scheduling Tasks

• In real life we often need to plan tasks with dependencies: some tasks need to be finished before other start. This can be thought of as a directed graph G with tasks as nodes and the arrows $u \to v$ if v should go after u. Applying Topological Sort (to G^T, in our definition), one either finds a loop (so the whole list of tasks can't be performed) or finds out a consitent order that allows to fullfil all dependencies. Below is a popular example on wearing clothes:

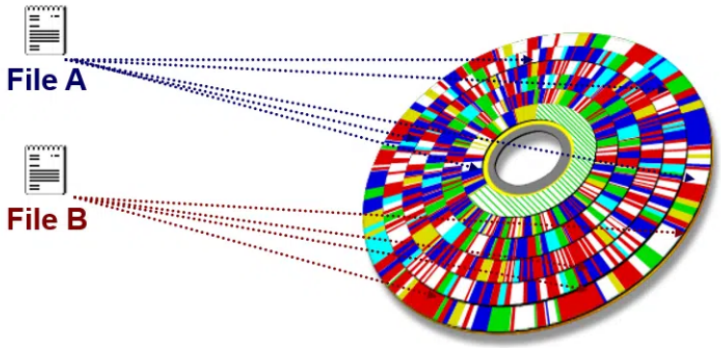


Fragmentation of Data Storage

• In data storage, data units (e.g. files) are deployed onto resources available, then are periodically edited, may grow or be deleted while other unit deployed.
• As the resources usage approach 100 percents, the phenomenon of *fragmentation* appears, with different issues in different level of technology used:
• If the technology is simple or special, every unit has to be a continuos segment in the resource map. In this case it happens that all remaining resources are numerous but small and even average size unit can't be deployed anymore.



• Flexible deployment splits a unit into several discontinuos subsegments, which simplifies deployment but slows down the access.



Defragmentation: Consolidation and Migration

A possible approach to the defragmentation consists in three steps:
A: Find a list of units, which are the most fragmented, then
B: Consolidation: In a model, tentatively remove units (unit pieces, for flexible deployment) from **A** and redeploy on the resources, more clean after removal.
C: Migration: In fact, in **B** we didn't remove anything, because this would need to copy all units to some outside storage. This is not optimal, and what we need is to only copy units one by one from old positions to new ones that have previously been made clean. Mathematically, migration looks as follows:

- Every unit in the list has two positions, old and new, defined in **B**. But these new and old positions overlap drastically. We need to find a feasible sequence of **swaping** units one by one to a free block.
- The simplest model of the migration is the Migration Graph with units as nodes and arrow $u_1 \to u_2$ if u_2 old position occupies the space overlapping with u_1 new position, so we need to migrate u_2 before u_1.
- Then we appply Topological Sorting to the above graph and if no cycles occur, this solves the Migration problem. But what if cycles occur?

Breaking Cycles in Migration and Johnson's algorithm

• If Migration Graph has cycles, then the simplest model will not help.
• Instead of thinking the migration as a simple process of swaping from an old position directly to the new one, we consider using some intermediate position.
• For simplicity, let's assume we have an additional storage of infinite size and we may temporary place any unit to there until the moment when the destination place for it becomes free, and we swap it to the destination position.
• The additional storage has some cost and swaping to there is slow, so we want to minimize the number or the total size of temporary allocated units.
Migration with outer storage: For the Migration Graph $G(V, E)$ to find a subset of nodes $W \subseteq V$ with minimal total size $\sum_{w \in W} size[w]$ such that the full subraph on vertices $V \setminus W$ is a DAG.
• The problem is not polynomial but has a heuristics approximate solution:
A: Call [D.Johnson, 1975] algorithm to find the set C of all **minimal cycles** in G in $O((|V| + |E|)|C|)$.
B: Solve a vertex coverage problem: find a subset of nodes $W \subseteq V$ with minimal total size $\sum_{w \in W} size[w]$ such that every $c \in C$ passes across at least one $w \in W$.
Step **B** is NP-hard but if $|C|$ is not too large, this approach may work efficiently.

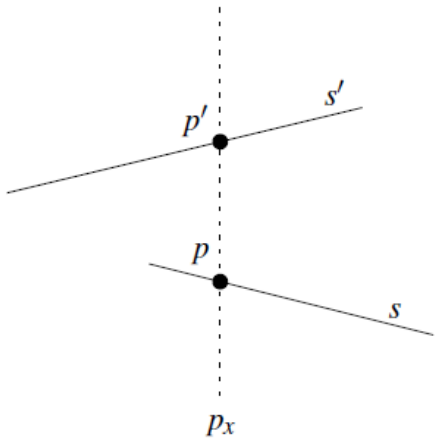
Ordering Segments

• Computational Geometry is extremely rich in specific data structures. In Lecture 1 we mentioned KD-Trees for range point queries on a plane.
• Another type of query is described in [BCKO, Chapter 3], it is about storing disjoint plane segments and effciently finding all segments, which intersect some vertical query segment. Specifically [BCKO, Theorem 10.13] claims that an efficent structure exists and its definition uses the following definition and Lemma:

Definition 21

Let S be a set of n disjoint (paiwise non intersecting) plane segments. For two segments, $s, s' \in S$ we say that s *lies below* s' denoted as $s \prec s'$ if and only if there are points $p \in s, p' \in s'$ such that $p_x = p_{x'}$ and $p_y < p_{y'}$.

Disjoint Segments constitute a DAG



Lemma 22

For a set S of disjoint segments consider the directed graph with $V = S$ and arrow $s \to s'$ whenever $s \prec s'$. This graph is a DAG.

We propose to prove Lemma 22 as an interesting exercise.

By Lemma 22, Topological Sorting applies to finding a consistent order. Then this order is utilized in the query data structure from [BCKO, Theorem 10.13].

References


A. Aho, J. Hopcroft, J. Ullman, *Data Structures and Algorithms*, 1983.


de Berg, M., Cheong, O., van Krefeld, M., Overmars, M., "Computational Geometry". Springer.


D. Johnson, *Finding all the elementary circuits in a directed graph*, *SIAM Journal on Computing*, **4** (1), 1975, 77-84.


D.Knuth, *Fundamental Algorithms*, volume 1, of *The Art of Computer Programming*, 1968.


R.Tarjan *Depth first search and linear graph algorithms*, *SIAM Journal on Computing*, **1** (2), 1972, 146-160.