1 Topological Sorting

Displaying discovery and finishing times as pairs \((d,f)\):

\[1 : (1,14) \rightarrow 3 : (2,3) \leftarrow 5 : (17,20) \rightarrow 7 : (18,19) \rightarrow 9 : (7,10)\]

\[2 : (15,16) \rightarrow 4 : (4,13) \rightarrow 6 : (5,12) \rightarrow 8 : (6,11) \rightarrow 10 : (8,9)\]

The resulting “directed spanning forest” (roots in bold):

\[1 \rightarrow 3 \quad 5 \rightarrow 7 \quad 9\]

\[2 \rightarrow 4 \quad 6 \rightarrow 8 \quad 10\]

And the resulting topological sorting is

\[5, 7, 2, 1, 4, 6, 8, 9, 10, 3.\]

Explanations:

- Discovery and finishing times:
1. The clock-counter starts with 0.
2. Every time a new vertex is “discovered” the clock is advanced by 1, and the new time is used as the discovery time of the vertex.
3. And every time a vertex is “finished” the clock is advanced by 1, and the new time is used as the finishing time of the vertex.
4. A vertex is discovered when it is the first remaining undiscovered vertex reachable from the currently explored vertex.
5. And a vertex is finished when all its neighbours have been finished.
6. When a vertex is finished, DFS backtracks.

- The topological sorting is obtained from the DFS run by listing the vertices in reverse order of their finishing times (last finished comes first).

2  Certifying dags and non-dags

A digraph $G$ is a dag (“directed acyclic graph”) if and only if $G$ does not contain a (directed) cycle.

- So the natural certificate that $G$ is not a dag is given by a cycle $C$ in $G$.
- Such a cycle exists if and only if $G$ is not a dag.
- $C$ is represented by a sequence $w_0, \ldots, w_k$ ($k \geq 1$) of vertices of $G$, such that for all $0 \leq i < k$ there is an arc from $w_i$ to $w_{i+1}$, and such that there is an arc from $w_k$ to $w_0$.
- The length of $C$ is $O(|V(G)|)$ (the size of a certificate is linear in the number of vertices).

It is a bit more complicated to certify that $G$ is a dag: We can not simply list of potential cycles (checking that they are not actual cycles), since there can be exponentially many of them. We need an idea ...

We have shown that every dag has a topological sorting.

Can we use a topological sorting as certificate for being a dag?!

It needs to be a shown that a digraph which is not a dag has no topological sorting.

So consider a digraph $G$ which is not a dag, and assume that $G$ has a topological sorting $v_1, \ldots, v_n$ (with $\{v_1, \ldots, v_n\} = V(G)$). So if there is an arc from $v_i$ to $v_j$, then $i < j$. Since $G$ is not a dag, there exists a cycle $w_0, \ldots, w_k$ in $G$. Let
\( i(j) \) be the index of \( w_j \) in the topological sorting, i.e., \( w_j = v_{i(j)} \). Since there is an edge from \( w_j \) to \( w_{j+1} \), and from \( w_k \) to \( w_0 \), we get

\[
 j(0) < j(1) < \cdots < j(k) < j(0),
\]

which is impossible. Thus \( G \) cannot have a topological sorting. QED

- So a natural certificate that \( G \) is a dag is given by a topological sorting of \( G \) (i.e., an ordering \( v_1, \ldots, v_n \) of the vertices of \( G \) such that if there is an edge from \( v_i \) to \( v_j \), then \( i < j \)).

- Such a topological sorting exists if and only if \( G \) is a dag.

- The length of this certificate is \(|V(G)|\) (so the size of a certificate is (again) linear in the number of vertices).

Final remarks:

- For a digraph \( G \), as explained in the lecture, we can decide in linear time whether \( G \) is a dag (or not):
  1. Run DFS on \( G \).
  2. If for every edge \((v, w)\) in \( G \) we have \( f(v) > f(w) \), then \( G \) is a dag, and we can extract a topological sorting (in linear time).
  3. Otherwise we found a cycle, and also this cycle can be extracted in linear time.

- So for being a dag or not we not only have short certificates (which can be easily verified), but also these certificates can be easily constructed.

## 3 All binary search trees

We want to enumerate all binary search trees with four nodes, labelled by \( 1, 1, 2, 2 \). If the root is labelled by \( 1 \), then the other \( 1 \) can be placed either left or right. If the other \( 1 \) is placed left, we get

\[
1 \quad 1 \quad 2 \quad 2, \quad 1 \quad 1 \quad 2 \quad 2.
\]

And if the other \( 1 \) is placed right, and it is the root of the sub-tree, we get

\[
1 \quad 1 \quad 2 \quad 2, \quad 1 \quad 1 \quad 2 \quad 2.
\]
And if the other 1 is placed right, but 2 is the root of the sub-tree, we get

\[ \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \] 

Finally, if the root is labelled by 2, then the other 2 can be placed left with 1 as the root of the sub-tree:

\[ \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \] 

left with 2 as the root of the sub-tree:

\[ \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \] 

or right:

\[ \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \quad \begin{array}{c}
1 & 2 \\
1 & 2 & 1 \\
& 2 & 1
\end{array} \] 

So altogether there are 14 possible binary search trees.

### 3.1 Counting the number of binary search trees in general

Didn’t we forget a possibility? Can we be really sure??

And what about the number of binary search trees with labels 1, 2, 3, 4? Is it different? Actually, it is the same, and this holds in general:

**Lemma 3.1** Consider an arbitrary sorted sequence \(a_1, \ldots, a_n\) of integers, where \(n \in \mathbb{N}\). Then the number of binary search trees \(T\) with \(n\) nodes and with labels exactly these numbers (in other words, the inorder-traversal of \(T\) yields \(a_1, \ldots, a_n\)) is equal to the number of binary trees with \(n\) nodes.

**Proof:** If we have binary search trees \(T_1, T_2\) with the same inorder-sequence of node-labels, and where \(T_1\) is equal to \(T_2\) as a binary tree (forgetting the labels), then \(T_1\) must also carry the same labels as \(T_2\) (i.e., \(T_1, T_2\) are equal as binary search trees), since from the binary tree and the inorder-sequence we can reconstruct the labelling. Thus two different binary search trees for sequence \(a_1, \ldots, a_n\) must yield to different binary trees (with \(n\) nodes).

On the other hand, given an arbitrary binary tree \(T\) with \(n\) nodes, we can label \(T\) so that we get a binary search tree \(T'\) with inorder-sequence \(a_1, \ldots, a_n\).
(note that this label is unique by what we’ve shown above): If the left subtree $L$ of $T$ has $p \in \mathbb{N}_0$ nodes and the right subtree $R$ of $T$ has $q \in \mathbb{N}_0$ nodes (thus $p + q + 1 = n$; for the sake of a simpler argument we allow here empty “trees”), then label $L$ recursively by the first $p$ elements $a_1, \ldots, a_p$ of the sequence, and label $R$ recursively by the last $q$ elements $a_{p+2}, \ldots, a_n$, while the root gets label $a_{p+1}$. 

So the task is to count the number of binary trees with $n \in \mathbb{N}$ nodes. Now it is known that the number of binary trees with $n$ nodes is equal to $C_n \in \mathbb{N}$, the $n$-th Catalan number, where we have that $C_1, \ldots, C_5$ is $1, 2, 5, 14, 42$. See [http://en.wikipedia.org/wiki/Catalan_number](http://en.wikipedia.org/wiki/Catalan_number).

At that Wikipedia-page you find that $C_n$ is equal to the number of full binary trees with $n + 1$ leaves, where “full” here means that we can not have nodes with exactly one child. Now it is not too hard to see that there is a bijection from the set of binary trees with $n$ nodes to the set of full binary trees with $n + 1$ leaves, where this bijection is given by the following map: For a binary tree $T$, add one leaf to a node with one child, and add two leaves to a node with no children. For example

\[
\begin{array}{c}
\bullet
\\
\longrightarrow
\\
\bullet
\end{array}
\begin{array}{c}
\bullet
\\
\bullet
\\
\bullet
\\
\bullet
\end{array}
\]

The inverse operation ($T' \mapsto T$) is given by removing all leaves from a full binary tree $T'$ (note that by induction it is easy to prove that a full binary tree with $n + 1$ leaves ($n \in \mathbb{N}_0$) has $n$ inner nodes (nodes which are not leaves)).

4 Creating binary search trees

The sequence of binary search trees generated by $3, 8, 9, 5, 1, 7, 2, 10, 4, 6$ is as follows:

\[
\begin{array}{c}
3, 3, 8, 3, 8, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\\
1, 3, 8, 1, 3, 8, 1, 3, 8
\end{array}
\]
The sequence of heights of these trees is 0, 1, 2, 2, 2, 3, 3, 3, 3, 4. And after deletion of the root we get

1 \quad 4
\quad \quad 8
\quad \quad \quad 9
\quad \quad \quad \quad 10
\quad \quad \quad \quad \quad 6
\quad \quad \quad \quad \quad \quad 7

If we delete the root again, we obtain

1 \quad 5
\quad \quad 8
\quad \quad \quad 9
\quad \quad \quad \quad 10
\quad \quad \quad \quad \quad 6
\quad \quad \quad \quad \quad \quad 7

5 Size-heuristics for the disjoint-forest data-structure

Consider performing \texttt{union}(x, y).

When performing this operation for the disjoint-forest data-structure, after finding the roots \(r_x, r_y\) of the trees belonging to \(x\) and \(y\), in case these roots are different, that is, we have to perform the union of the two disjoint sets, there are two choices: attach \(r_x\) to \(r_y\), or attach \(r_y\) to \(r_x\).

In order to minimise the height of the trees, one would attach the tree with the smaller height to the tree with the larger height. However the size heuristics, which attaches the smaller tree (with the smaller number of nodes) to the larger tree is easier to analyse.

The analysis is similar to the analysis of the weighted-union heuristic for the linked-list data-structure:

1. The “bad event” is that by a union-operation the size of the new tree is bigger than the height of the two old trees.

2. Let \(h_x, h_y\) be the heights of the old trees, and let \(h_z\) be the height of the new tree. In general we have \(h_z \leq \max(h_x, h_y) + 1\), that is, a single union-operation can increase the height only by 1.

3. So the bad event is that we have \(h_z = \max(h_x, h_y) + 1\).

4. The trick is now not to study a single bad event, but to think in an “amortised way”: Follow all nodes \(u\) over their live-times (starting with each node a tree on its own, being combined together by the union-operations). And count for each node \(u\) how often a bad event can happen — this count is then the height of the tree \(u\) ends up.
5. In order for this to be efficient, we have to look a bit closer at the bad event \( h_z = \max(h_x, h_y) + 1 \): For which nodes \( u \) is this bad — for all nodes in both trees?

6. We need another idea: Still the notion of a “bad event” is “too global”, and actually not sharp enough.

7. The appropriate notion is the depth of the (single) node \( u \), the length of the path from \( u \) to the root of its current tree. A “bad event” for \( u \) is then that its depth increases (by 1).

8. We see that precisely the nodes in that tree, which is attached to the other tree, have their depth increased (by 1), while for the nodes in the other tree the depth doesn’t change.

9. Since the smaller tree is attached to the larger, the overall size doubles (at least) in size. And thus, since a tree can have at most \( n \) nodes, increase of the depth can happen at most \( \log_2(n) = \lg(n) \) often. In other words, finally the depth of any node \( u \) is at most \( \lg(n) \).

10. Since the effort for a union-operation is linear in the sum of the depth of \( x \) and \( y \), and we have (at most) \( m \) union-operation, we get the upper bound \( O(m \cdot \lg n) \) for the union-operations.

11. For the find-operations we get also a upper bound \( O(m \cdot \lg n) \) for the effort, while for the make-operations we get \( O(n) \).

12. Altogether this makes \( O(m \cdot \lg n) \).

6 Online-graphs

Consider a (finite) graph \( G \) (we assume \( G \) has been fixed). Let \( k \in \mathbb{N}_0 \) be the number of connected components of \( G \), and let the components be \( C_1, \ldots, C_k \). So the \( C_i \) are disjoint sets with \( C_1 \cup \cdots \cup C_k = V(G) \). And let \( s_1, \ldots, s_k \in \mathbb{N} \) be the sizes of these components, that is, \( s_i := |C_i| \).

- The task is to compute \( m \) and \( s_1, \ldots, s_k \).
- The difficulty is that \( G \) is very large, having billions of vertices (i.e. \( |V(G)| \) is of the order \( 10^9 \)).
- We assume that we can store the vertices of \( G \), but not the edges of \( G \).
- As an example consider \( |V(G)| \sim 10^9 \): even if \( G \) is very sparse, then on average every vertex of \( G \) could have 10000 neighbours, and then we would have \( \frac{1}{2}10^9 \cdot 10^4 = 5 \cdot 10^{12} \) edges. This would still be feasible, especially on external memory. However if \( G \) is, say, 100 times larger, then it becomes very difficult, and using external memory can make it very slow.
So we want to consider that the edges are not stored — how can our algorithms be utilised for such “online-computations”?

We discussed basically two methods for computing $m$ and $s_1, \ldots, s_k$:

1. We can instrument graph-traversal, either BFS or DFS:
   (a) When using BFS, it also needs an outer loop, running through all vertices, like DFS.
   (b) A counter $m$ for connected components is introduced, initialised with 0 and incremented by 1 each time in the outer loop an undiscovered vertex is found.
   (c) And in the inner loop another counter $s$ is introduced, also initialised with 0 and incremented by 1 each time in the inner loop an undiscovered vertex is found. At the end of the run of the inner procedure $s$ is the size of the current connected component.

2. Or we use disjoint-sets: each edge means a union-operation in case the endpoints still belong to different components i.e., to disjoint sets. This is the core of the use of disjoint-sets in Kruskal’s algorithm.

These algorithms have the following requirements regarding edge-access:

1. BFS needs for each vertex $v$ at the time of its discovery that we can find all of $v$’s neighbours, i.e., all edges incident with $v$. These edges don’t need to be stored, but we just run through them. And the basic data-structure, the queue for the vertices still to be discovered, only stores (different!) vertices.
   This algorithm would for example be suitable when handling web-pages as vertices: when we encounter a web-page, we run through all the edges, that is, links, in it, and then we can forget about the links (only the identity of the web-pages, something like an extended IP-address, needs to be stored).

2. DFS is a more difficult: When discovering vertex $v$, we only need the one yet undiscovered neighbour, but when returning (backtracking) to $v$, then we need the next undiscovered vertex. Thus we need to store for a vertex some information what this next vertex (next adjacent edge) is. And if we don’t have random access to the neighbours (or adjacent edges) of $v$, then running time increases from linear to quadratic.

3. Very easy on the other hand is the algorithm using disjoint-sets: When an edge comes, it is used, and the order of the edges is irrelevant, and also edges can occur multiple times without problems.
So the algorithm based on disjoint-sets is for online-purpose by far best (when only the connected components need to be identified), since the edges can come in any order. BFS is second: the edges can’t come in arbitrary order, however it is sufficient if we have access to the edges adjacent with a given vertex. DFS is most difficult for online-purposes, since an efficient backtracking w.r.t. the edges adjacent to a given vertex needs to be organised.