Module 14.1: Directed Graphs

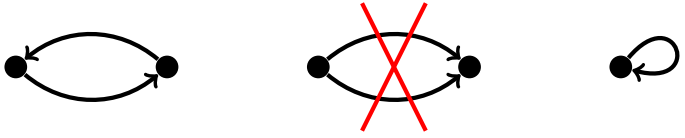
MCIT Online - CIT592 - Professor Val Tannen

Lecture Notes

Directed graphs (digraphs)

A **directed graph (digraph)** $G = (V, E)$ consists of a non-empty set V of **vertices** (nodes) and a set $E \subseteq V \times V$ of (directed) **edges** which are **ordered** pairs of vertices.

This allows **self-loops** $(v, v) \in E$ where $v \in V$ but not for **parallel** edges. We can have **anti-parallel** edges $(u, v), (v, u) \in E$ where $u, v \in V$.



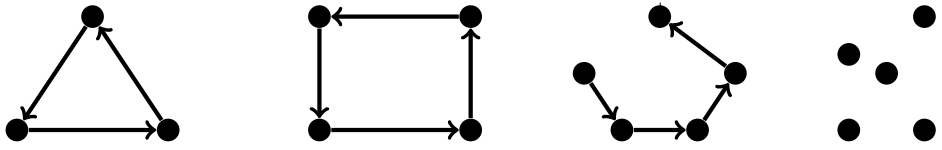
We use the notation $u \to v$ for both the edge (u, v) itself and for the fact that $(u, v) \in E$. $u \to v$ is an **outgoing** edge **from** u and an **incoming** edge **to** v.

More digraph terminology

If $u \to v$ then v is a **successor** of u and u is a **predecessor** of v. Moreover, u and v are **neighbors**, when $u \to v$ or $v \to u$, a symmetric relationship. Since we allow self-loops, a vertex can be its own neighbor in which case it is also its own predecessor and its own succcessor.

Isolated vertices: vertices with no neighbors. **Edgeless digraph:** just like an edgeless graph.

Examples:



Quiz

What is the maximum number of edges that a digraph with n nodes can have?

(A) n^2

(B) $\binom{n}{2}$

(C) $n(n-1)$

Answer

(A) n^2

Correct. Each node can have n successors, i.e. n edges leaving it to every other node and itself. Since there are n nodes, then the maximum number of edges for a digraph with n nodes is n^2

(B) $\binom{n}{2}$

Incorrect. Try to count correctly by considering how many edges can leave from every node (i.e. how many successors each node can have at most).

(C) $n(n-1)$

Incorrect. Recall that digraphs allow self loops.

More Information

A digraph with n vertices and n^2 edges might merit a name that includes **complete**. However, to my knowledge, this is not standard terminology and I will avoid it in this course.

Degrees

The **outdegree** of a vertex u is the number of successors of u, same as the number of outgoing edges from u, denoted $\text{out}(u)$.

Similarly, the **indegree** of a vertex u is the number of predecessors of u, same as the number of incoming edges to u, denoted $\text{in}(u)$.

The **degree** of a node is $\deg(u) = \text{out}(u) + \text{in}(u)$. A node of indegree 0 is called a **source** and a node of outdegree 0 is called a **sink**.

Proposition. The sum of the outdegrees of all vertices equals the sum of the indegrees of all vertices and further equals the number of edges. Therefore, the sum of the degrees equals twice the number of edges.

Proof. Every edge is an outgoing edge and an incoming edge so we can count in three different ways.

Directed walk and path

A **directed walk** is a non-empty sequence $u_0, u_1, \ldots, u_k$ such that $u_0 \to u_1 \to \cdots \to u_k$. We call this a directed walk **from** u_0 **to** u_k of **length** k.

A **directed path** is a walk with no repeated vertices.

For every vertex v there is a directed path of length 0: v.

Do not confuse the directed walk of length 0, u with the directed walk of length 1 given by the existence of a self-loop: $v \to v$. Walks of length 0 are paths but walks of length 1 given by self-loops are *not* paths.

The theorem "where there is a walk, there is a path" holds in digraphs also, with a similar proof.

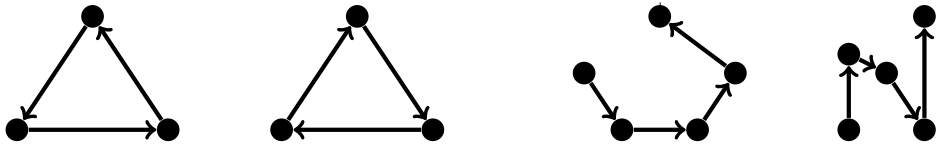
Directed cycles

A **directed cycle** is a closed walk $u_0 \to \cdots \to u_k \to u_0$, with $u_0, \ldots, u_k$ all distinct. The **length** of the cycle is $k + 1$.

There are no cycles of length 0. A self-loop gives a cycle of length 1. A cycle of length 2 consists of two vertices and edges between them in opposite directions (antiparallel edges).

Digraph isomorphism is defined similarly to the undirected case.

Examples:



Activity

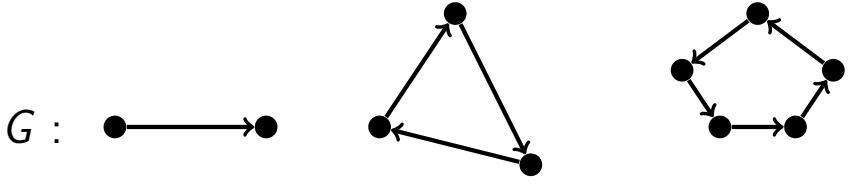
Consider digraphs in which all the vertices have indegree 0 or 1 and outdegree 0 or 1.

Question. Prove or disprove the statement that the only such digraphs are either path digraphs or cycle digraphs.

In the video, there is a box here for learners to put in an answer to the question above. As you read these notes, try it yourself using pen and paper!

Activity

Answer. We disprove the statement with a counter example.
Consider the following graph with three "components":



Module 14.2: Reachability, Strong Connectivity

MCIT Online - CIT592 - Professor Val Tannen

Lecture Notes

Reachability

A vertex v is **reachable** from a vertex u when there is a walk (and therefore a path) from u to v. We write $u \twoheadrightarrow v$ for the **reachability relation**.

Proposition. The reachability relation is **reflexive**, i.e., $u \twoheadrightarrow u$ and **transitive**, i.e., $u \twoheadrightarrow v$ and $v \twoheadrightarrow w$ imply $u \twoheadrightarrow w$.

Proof. For reflexivity consider walks of length 0. For transitivity we concatenate walks.

The relation $u \twoheadrightarrow v$ is called the **reflexive-transitive closure** of the (edge) relation $u \to v$.

Activity

DO NOT BUILD THIS ACTIVITY IN COURSERA

Recall the $m \times n$-grid, the undirected graph used in earlier segments for examples. We name its vertices as the elements of the cartesian product $[1..m] \times [1..n]$ so that (k, ℓ) is the ℓ'the vertex in the k'th row.

Make this grid into a digraph by putting direction on the edges: $(k, \ell) \to (k+1, \ell)$ and $(k, \ell) \to (k, \ell+1)$ for $k \in [1..(m-1)]$ and $\ell \in [1..(n-1)]$.

TIKZ OF 3×5 directed grid. (I HAVE THE 3by8 pic somewhere in an earlier segment)

TAKE THE STUDENT THROUGH PROVING

$(k, \ell) \twoheadrightarrow (i, j)$ iff $k \leq i$ and $\ell \leq j$

BY (DOUBLE) INDUCTION ON i and j.

Strong connectivity I

Two vertices of a digraph, u and v, are **strongly connected** when v is reachable from u and u is reachable from v, i.e., $u \twoheadrightarrow v$ and $v \twoheadrightarrow u$.

We denote the **strong connectivity relation** between vertices u and v by $u \twoheadleftarrow\!\!\!\twoheadrightarrow v$.

Proposition. Strong connectivity is

- **reflexive**, that is, $u \twoheadleftarrow\!\!\!\twoheadrightarrow u$.
- **symmetric**, that is, if $u \twoheadleftarrow\!\!\!\twoheadrightarrow v$ then $v \twoheadleftarrow\!\!\!\twoheadrightarrow u$.
- **transitive**, that is, $u \twoheadleftarrow\!\!\!\twoheadrightarrow v$ and $v \twoheadleftarrow\!\!\!\twoheadrightarrow w$ imply $u \twoheadleftarrow\!\!\!\twoheadrightarrow w$.

The proof is similar to what we saw for connectivity and reachability. We omit the details.

Strong connectivity II

A set of vertices is strongly connected when any two of its vertices are strongly connected.

The **maximally** strongly connected sets of vertices are called **strongly connected components (scc's)**.

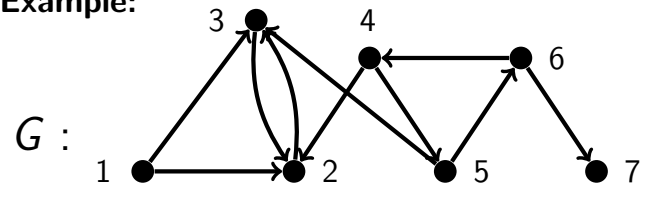
Proposition. Any two distinct strongly connected components are disjoint.

Proof. Suppose, toward a contradiction, that two distinct scc's, S_1 and S_2, have a vertex w in common. Then, for any $u \in S_1$ and any $v \in S_2$ we have $u \leftrightarrows w \leftrightarrows v$ thus $u \leftrightarrows v$. By maximality, $u \in S_2$ and $v \in S_1$. Therefore $S_1 = S_2$. Contradiction.

Corollary. The strongly connected components determine a **partition** of the vertices (but not of the edges!).

Reduced graph I

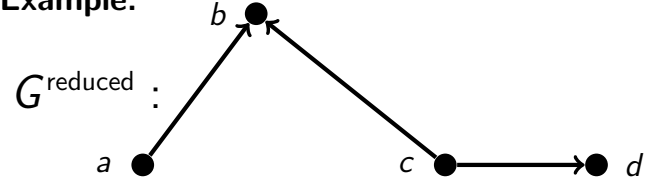
Example:



SCC	Notation
.{1}	a
.{2,3}	b
.{4,5,6}	c
.{7}	d

Given a digraph $G = (V, E)$ its **reduced graph** has as vertices the scc's of G and as edges the pairs (S_1, S_2) where S_1 and S_2 are distinct scc's such that there exist $u_1 \in S_1$ and $u_2 \in S_2$ such that $u_1 \rightarrow u_2$ is an edge in G.

Example:



Reduced	Causes
$.a \rightarrow b$	$1 \rightarrow 2, 1 \rightarrow 3$
$.c \rightarrow b$	$4 \rightarrow 2, 5 \rightarrow 3$
$.c \rightarrow d$	$6 \rightarrow 7$

Reduced graph II

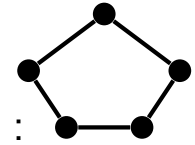
Proposition. The reduced graph has no directed cycles.

Proof. Because the reduced graph has edges only between distinct vertices we cannot have cycles of length 1.

Now suppose, toward a contradiction, that the reduced graph has a directed cycle of length ≥ 2. This cycle has at least two distinct vertices, i.e., two **distinct** SCCs, S_1 and S_2. Let $v_1, v_2 \in V$ such that $v_1 \in S_1$ and $v_2 \in S_2$. We then show (details omitted) that the cycle in the reduced graph implies that there exists a cycle C in G such that v_1, v_2 belong to C. It then follows that $v_1 \leftrightarrows v_2$, hence $S_1 = S_2$. Contradiction.

Quiz

We construct a digraph by assigning direction, arbitrarily, to the edges of the undirected cycle graph C_5 (seen below).

C_5 : 

The resulting digraph has:

(A) 1 or 5 scc's

(B) 2 or 3 scc's

(C) 4 scc's

Answer

(A) 1 or 5 scc's

Correct. If all edges have the same direction there is only one scc. In any other case each node is a scc on its own.

(B) 2 or 3 scc's

Incorrect. Make sure you try out some examples in order to find the right answer.

(C) 4 scc's

Incorrect. Make sure you try out some examples in order to find the right answer.

Module 14.3: Directed Acyclic Graphs (DAGs)

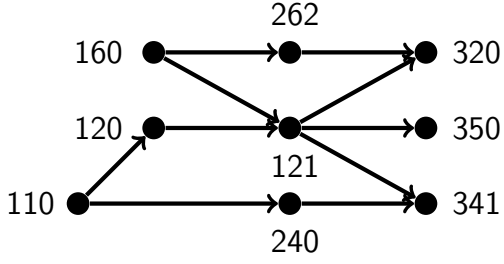
MCIT Online - CIT592 - Professor Val Tannen

Lecture Notes

Directed acyclic graph (DAGs)

A **directed acyclic graphs (DAG)** is a digraph without directed cycles (not even of length 1).

Example: DAG of course prerequisites



Sources: 110, 160

Sinks: 320, 341, 350

Possible timelines for taking **all** these courses:

. 110, 120, 240, 160, 121, 262, 350, 320, 341

. 110, 160, 262, 120, 121, 320, 240, 341, 350

Topological sort

A **topological sort** of a digraph is a sequence σ in which every vertex appears exactly once (i.e., a permutation of its vertices) such that for any edge $u \to v$ in the graph, the vertex u appears in σ **before** (but not necessarily **immediately** before) the vertex v.

Proposition. If a digraph has a topological sort then:

- The first vertex in the sort is a source and the last is a sink.
- The digraph is a DAG.

Proof. The first vertex in the sort cannot have a predecessor because it would have to appear before it. Hence it's a source. Similar for the last.

For any path $u_1 \to u_2 \to \cdots \to u_n$, the vertex u_1 must appear in the topo sort before u_2, which must appear before u_3, etc. If we had a cycle then a vertex in this cycle would have to appear before ... itself! Contradiction.

Quiz

Note that an **edgeless** digraph is always a DAG. If there are n vertices, how many distinct topological sorts does such an edgless digraph have?

(A) $n!$
(B) $\binom{n}{2}$
(C) n

Answer

(A) $n!$

Correct. Since there are no edges in the graph there are no constraints in the ordering of the nodes in the topological sorts; therefore the number of topological sorts is equal to the number of ways to order n nodes, i.e. $n!$.

(B) $\binom{n}{2}$

Incorrect. Since there are no edges in the graph then there are no constraints in the ordering of the nodes. How many orderings of n nodes are there?

(C) n

Incorrect. Since there are no edges in the graph then there are no constraints in the ordering of the nodes. How many orderings of n nodes are there?

Sources and sinks in DAGs

Proposition. Every DAG has at least one source and at least one sink.

Proof. Isolated vertices are both sources and sinks. This takes care of the case when the DAG is edgeless. If the DAG has at least one edge we consider a directed path of maximum length, p (which exists by the Well-ordering Principle). Suppose p goes from u to v. We claim that u is a source. (Similarly for v as a sink.) Indeed, suppose, toward a contradiction, that $in(u) \geq 1$ hence there exists an edge $w \to u$. We have two cases:

Case 1: $w \notin p$. Then $w \to u \twoheadrightarrow v$ is a directed path that is strictly longer than p, contradiction.

Case 2: $w \in p$. Then $w \to u \twoheadrightarrow w$ is a directed cycle, contradiction.

Quiz

A DAG is called "fluid" if it has no isolated vertices and if there is a directed path from every source to every sink. The **minimum** number of edges in a fluid DAG with 2 sources and 3 sinks is

(A) 3

(B) 5

(C) 6

Answer

(A) 3

Incorrect. Did you make sure there is a path from each of the two sources to each of the 3 sinks?

(B) 5

Correct. In addition to the 2 sources and 3 sinks we have an intermediate node through which all the source-to-sink paths go.

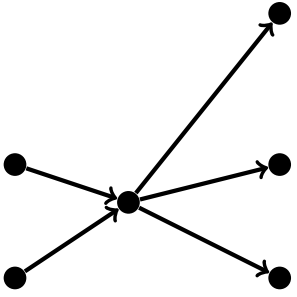
(C) 6

Incorrect. Make sure you capture the *minimum* number of edges.

More information

Bellow you can see a DAG with the minimum number of edges (5) that satisfies the constraints of the question:

G :



Constructing topological sorts I

Proposition. Every DAG has at least one topological sort.

Proof idea and algorithm. The proof is by induction on the number of vertices and it is given in detail in the segment "Proof of topological sorting". Here we sketch the idea of the proof as well as a **recursive** algorithm for constructing a topological sort.

Input: a DAG $G = (V, E)$ with n vertices
Output: a permutation σ of V that is a topological sort.

In the base case of the induction as well as when the recursion **bottoms out**, we have 1 vertex, v, and no edges. Then v is the desired permutation.

Constructing topological sorts II

Proposition. Every DAG has at least one topological sort.

Proof idea and algorithm (continued). In the induction step as well as in the recursive call we observe the following.

G has at least one source, u. Remove u and call G_u the resulting digraph.

Removing u cannot create a cycle since we only **delete** edges. G_u is also a DAG.

G_u has $n-1$ vertices, so by IH it has a topological sort.

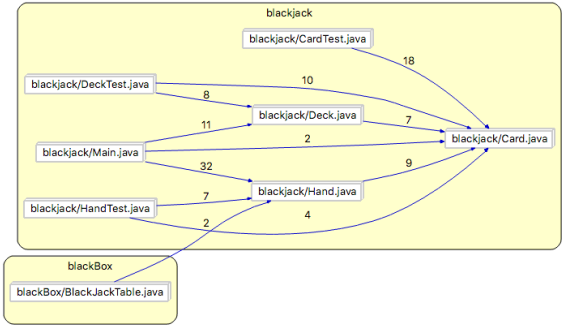
Recursively construct a topological sort of G_u, i.e., a permutation σ of $V \setminus \{u\}$. Then, concatenate u at the beginning. Output $u\sigma$ which is a topological sort of G.

Activity

In software engineering, we use **dependency graphs** to visualize the dependencies of several objects towards each other. Dependency graphs are digraphs where the nodes are the different modules of a project, and there is a directed edge from module A to module B if B depends on A (i.e. B calls some function or refers to some object in A). Each edge has a corresponding number on top of it, which indicate the number of times A has been called from B. In this activity you will apply the algorithm we just introduce to find its topological sort.

Activity

Below you can see an example of a real dependency graph of a project that simulates a Blackjack game, and includes the testing modules as well. Apply the recursive topological sort algorithm to this dependency graph.

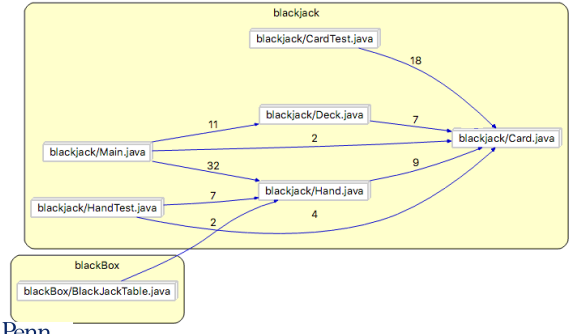


In the video, there is a box here for learners to put in an answer to the question above. As you read these notes, try it yourself using pen and paper!

Answer. We start by removing a source of the graph. We start by removing "blackjack/DeckTest.java" and placing it in the beginning of the topological sort, where σ' denotes a random ordering of the set of vertexes minus the vertex "blackjack/DeckTest.java":

$$\sigma = \text{"blackjack/DeckTest.java"}\sigma'$$

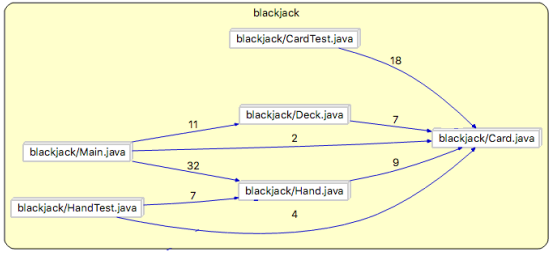
The resulting graph is



Answer (continued). We continue by removing another source, we choose "blackBox/BlackJackTable.java". Again σ' denotes a random sequence of the remaining vertexes that have not been topologically sorted yet (for brevity we will not repeat this definition in the following steps). Thus we have

$$\sigma = \text{"blackjack/DeckTest.java""blackBox/BlackJackTable.java"}\sigma'$$

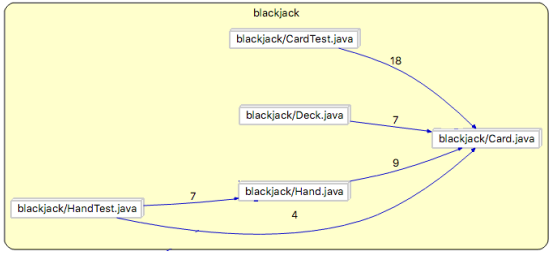
The resulting graph is



Answer (continued). We continue by removing another source, we choose "blackjack/Main.java". Thus we have

$$\sigma = \text{blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java}\sigma'$$

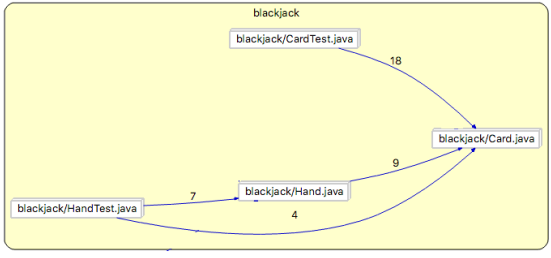
The resulting graph is



Answer (continued). We continue by removing another source, we choose "blackjack/Deck.java". Thus we have

σ = blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java, blackjack/Deck.javaσ'

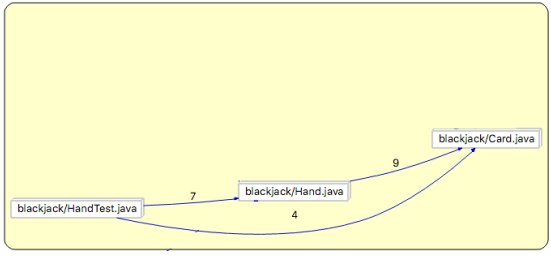
The resulting graph is



Answer (continued). We continue by removing another source, we choose "blackjack/CardTest.java". Thus we have

$$\sigma = \text{blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java, blackjack/Deck.java, blackjack/CardTest.java}\sigma'$$

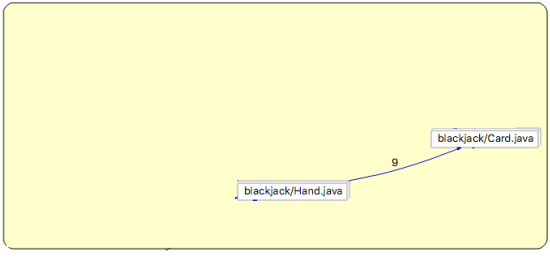
The resulting graph is



Answer (continued). We continue by removing another source, we choose "blackjack/HandTest.java". Thus we have

$$\sigma = \text{blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java, blackjack/Deck.java, blackjack/CardTest.java, blackjack/HandTest.java}\sigma'$$

The resulting graph is



Activity

Answer (continued). Finally, we remove "blackjack/Hand.java" making the topological sort:

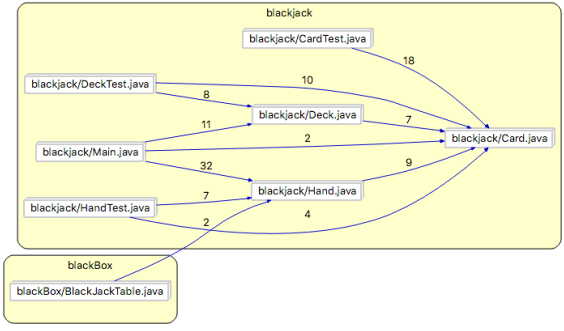
$$\sigma = \text{blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java, blackjack/Deck.java, blackjack/CardTest.java, blackjack/HandTest.java, blackjack/Hand.java}\sigma'$$

And then there is only one vertex remaining the "blackjack/Card.java". Thus, the final topological sort it

$$\sigma = \text{blackjack/DeckTest.java, blackBox/BlackJackTable.java, blackjack/Main.java, blackjack/Deck.java, blackjack/CardTest.java, blackjack/HandTest.java, blackjack/Hand.java, blackjack/Card.java}$$

Answer (continued). This is not the only topological sort, as topological sorts are not unique. Moreover, this is not the only algorithm for computing the topological sort. We now show another algorithm for computing the topological sort: we recursively remove a sink u and concatenate it to the end of the topological sort.

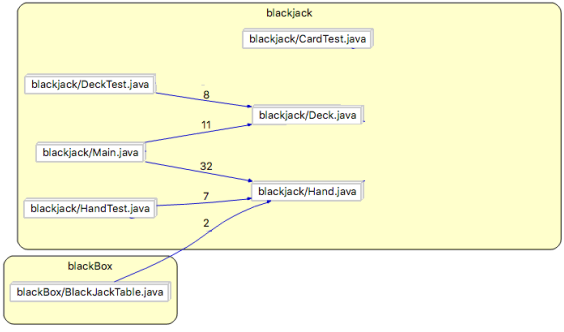
We start with the complete dependency graph below:



Answer (continued). The only sink is "blackjack/Card.java"; we remove it and concatenate it in the beginning of the overall sequence:

$$\sigma = \text{blackjack/Card.java}\sigma'$$

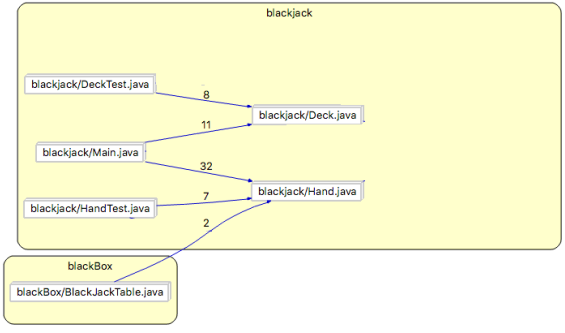
The resulting graph is



Answer (continued). Next, we remove the sink "blackjack/CardTest.java" and concatenate it in the beginning of the overall sequence:

$$\sigma = \text{blackjack/CardTest.java, blackjack/Card.java}\sigma'$$

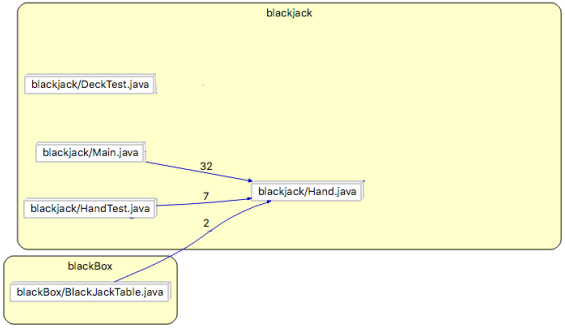
The resulting graph is



Answer (continued). Next, we remove the sink "blackjack/Deck.java" and concatenate it in the beginning of the overall sequence:

$$\sigma = \text{blackjack/Deck.java, blackjack/CardTest.java, blackjack/Card.java}\sigma'$$

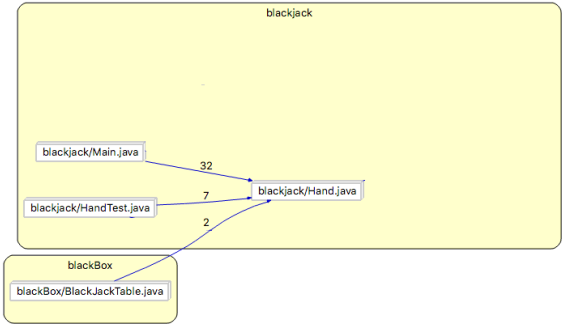
The resulting graph is



Answer (continued). Next, we remove the sink "blackjack/DeckTest.java" and concatenate it in the beginning of the overall sequence:

$$\sigma = \text{blackjack/DeckTest.java,blackjack/Deck.java,}$$
$$\textit{blackjack/CardTest.java}, \textit{blackjack/Card.java}\sigma'$$

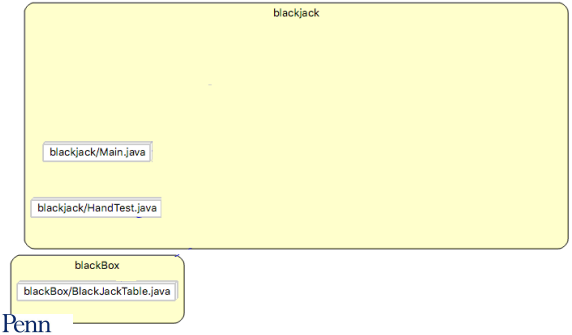
The resulting graph is



Answer (continued). Next, we remove the only sink "blackjack/Hand.java" and concatenate it in the beginning of the overall sequence:

$$\sigma = \text{blackjack/Hand.java, blackjack/DeckTest.java, blackjack/Deck.java, blackjack/CardTest.java, blackjack/Card.java}\sigma'$$

The resulting graph is



Activity

Answer (continued). The last three vertices are disconnected, so it does not matter in which order we place them in the topological sort. Thus, the final topological sort is:

$$\sigma = \text{blackjack/BlackJackTable.java, blackjack/HandTest.java, blackjack/Main.java, blackjack/Hand.java, blackjack/DeckTest.java, blackjack/Deck.java, blackjack/CardTest.java, blackjack/Card.java}$$

Note that this topological sort is different than the first one, but they are both correct. In fact, there are more topological sorts for this dependency graph, can you find them?

Module 14.4: Binary Trees

MCIT Online - CIT592 - Professor Val Tannen

Lecture Notes

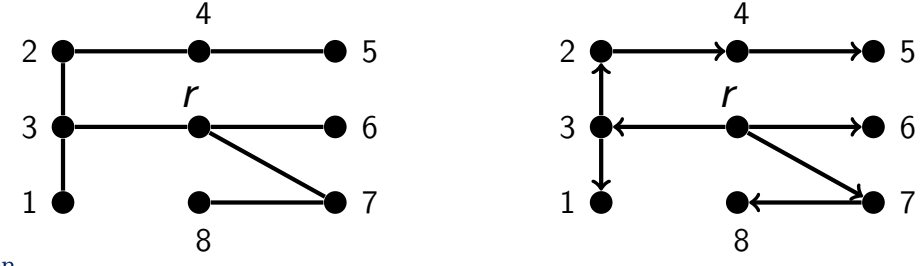
Rooted trees

A **rooted tree** is a pair (T, r) where $T = (V, E)$ is a tree and the vertex $r \in V$ is designated as a **root**.

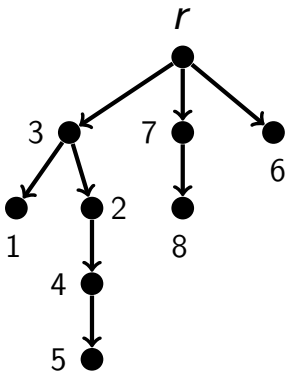
Proposition. Any edge of a rooted tree is traversed in the **same direction** by all unique paths from the root to each of the other vertices.

Using this direction, make the rooted tree into a **digraph**, in fact, a DAG with the root as source and tree leaves (except, possibly, the root) as sinks.

Example:



Rooted tree terminology



root: r

leaves: $1, 5, 8, 6$

child $=$ successor

parent $=$ predecessor

children$(r) : 3, 7, 6$ children$(3) : 1, 2$

parent$(2) : 3$ parent$(5) : 4$

height $=$ distance from root to farthest leaf

height $= 4$

binary (rooted) tree: every node has at most 2 children

Activity : A terminology convention for one-node rooted trees

When a rooted tree has just one node then that node must be also the **root**.

Following our definition from undirected graphs, we call **leaves** the nodes of degree 1 in rooted trees, as we saw in the previous slide.

However, when trees are used as **data structures** (see **binary search trees** below), it is convenient to state that the single node of a one-node rooted tree is **also a leaf**, although it has degree 0. We shall adopt this convention.

For example, this will make the one-node rooted tree a "complete binary tree", as defined in the next slide.

Binary trees I

Proposition. A binary tree of height h has a maximum of $2^{h+1} - 1$ nodes among which are 2^h leaves. This maximum is attained for the **complete binary tree of height** h.

We define a **complete binary tree of height** h to be a rooted tree in which every non-leaf node has two children and all leaves are at distance h from the root.

Proof. By induction on the height, h.

(BC) $h = 0$. A tree of height 0 has only one node, the root, which is also its only leaf. $1 = 2^{0+1} - 1$ and $1 = 2^0$. Clearly it is a complete binary tree. Check.

Binary trees II

Proof (continued).

(IS) Let $k \in \mathbb{N}$ be arbitrary. Assume (IH) that a binary tree of height k has a maximum of $2^{k+1} - 1$ nodes among which there are 2^k leaves and that this maximum is attained for the complete binary tree of height k.

Now consider a tree T of height $k+1$. Let r be the root of T. To maximize the number of nodes in T, r must have 2 children which in turn are roots of two other trees T_1, T_2 of height k.

By IH each of T_1, T_2 is a complete binary tree with $2^{k+1} - 1$ nodes and 2^k leaves. Hence T is also a complete binary tree and it has

$$(2^{k+1} - 1) + (2^{k+1} - 1) + 1 = 2 \cdot 2^{k+1} - 1 = 2^{k+2} - 1 \text{ nodes}$$

among which there are $2^k + 2^k = 2 \cdot 2^k = 2^{k+1}$ leaves.

Quiz

How many edges are in a complete binary tree of height h?

(A) 2^{h+1}

(B) $2^{h+1} - 1$

(C) $2^{h+1} - 2$

Answer

(A) 2^{h+1}

Incorrect. Recall the definition of a complete binary tree and experiment with some examples if you have trouble finding the right answer.

(B) $2^{h+1} - 1$

Incorrect. Recall the definition of a complete binary tree and experiment with some examples if you have trouble finding the right answer.

(C) $2^{h+1} - 2$

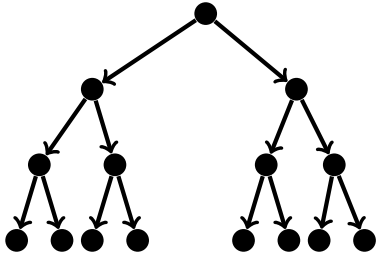
Correct. We compute the sum of indegrees: We have that every node except the root has exactly one incoming edge. It follows that the number of edges is $(2^{h+1} - 1) - 1$.

More information

We reach the same answer by computing the sum of outdegrees instead: we have that every node except the leaves have two outgoing edges. It follows that the number of edges is

$$2(2^{h+1} - 1 - 2^h) = 2(2^h - 1) = 2^{h+1} - 2$$

You can check this answer by considering a complete binary tree of height 3.

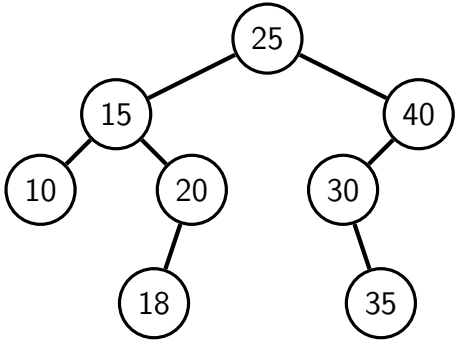


We can count the number of edges to be $2^{3+1} - 2 = 2^4 - 2 = 16 - 2 = 14$.

A complete binary . . . palm tree!

Binary search trees

Binary search trees are, in addition, **ordered**. Each node may have a **left** child and a **right** child.



Stored keys:
25, 40, 30, 15, 20, 10, 35, 18

Search keys:
18, 19, 28

.

OMCIT 592 Module 14 Self-Paced 01 (instructor Val Tannen)

One reference to this self-paced segment, in lecture segment 14.3.

This is a segment that contains material meant to be learned *at your own pace*. We are trying to assist you in this endeavor by organizing the material in a manner similar to the way it is outlined in the recorded segments, however with one additional suggestion.

When you see the following marker:

we suggest that you stop and make sure you thoroughly understood the material presented so far before you proceed further.

Proof of topological sorting

In the lecture segment "Directed acyclic graphs (DAGs)" we stated the following:

Proposition. Every DAG has at least one topological sort.

In the lecture segment we sketched the idea of the proof as well as a **recursive** algorithm for constructing a topological sort. Here we give a detailed proof.

Proof. By induction on n, the number of vertices of the DAG.

(BC) $n = 1$. We cannot have any edges because they would form a cycle of length 1. With just one vertex v and zero edges the topological sort is the sequence v.

(IS) Let $k \geq 1$ arbitrarily. Assume (IH) that any DAG with k vertices has at least one topological sort.

Now take any DAG G with $k + 1$ vertices.

By the proposition on sources and sinks that we proved in the same lecture segment G has a source u (and a sink too, but we don't use it in this proof).

Delete u from G. This deletes also all the edges (if any) outgoing from u. There are no incoming edges to u because u is a source.

The resulting graph G_u must also be a DAG because no directed cycle is created when we just remove edges. Since G_u is a DAG with k vertices we can apply the IH and obtain a topological sort of G_u, call it σ.

Now we claim that the concatenated sequence $u\,\sigma$ is a topological sort of G.

Clearly it is a permutation of the vertices of G. Moreover, for any edge $x \to y$ of G we have two cases:

Case 1: $x \to y \in G_u$. Then x appears before y in σ, hence also in $u\,\sigma$.

Case 2: $x \to y$ is not in G_u. Then $x \equiv u$ and u occurs before y in $u\,\sigma$.