5CCS2FC2: Foundations of Computing II

Graph Algorithms

Week 5

Dr Christopher Hampson

Department of Informatics

King's College London

Topological Sorting

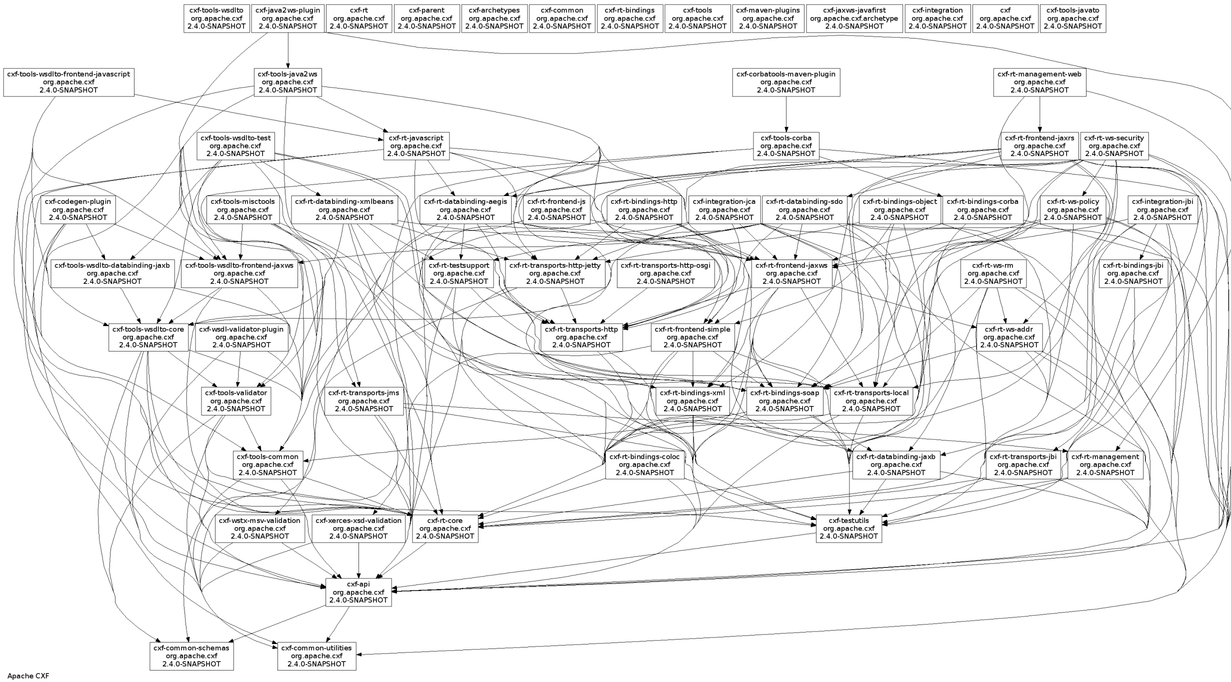
Directed Acyclic Graphs (DAGs)

- **Directed Acyclic Graphs (DAGs)**
  - A graph $G = (V, E)$ is said to be a **directed acyclic graph** if it is **irreflexive** and does not contain any **cycles** of length ≥ 2

    > if there is a path $u \rightsquigarrow v$ then there is no path $v \rightsquigarrow u$

    (for all vertices $u, v \in V$)

Example: Software Dependency Graphs



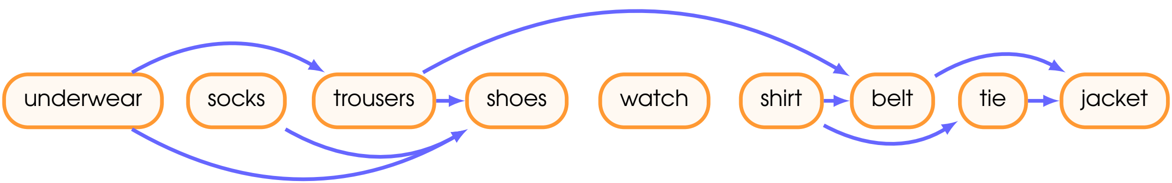
(http://cxf.apache.org/docs/cxf-dependency-graphs.html)

Topological Sorting

- **Topological Sort**
  - A **topological sorting** of a Directed Acyclic Graph $G = (V, E)$ is a sequence of vertices $Q = \langle v_1, v_2, \ldots, v_n \rangle$ such that

$$\boxed{\text{If} \quad v_i \rightsquigarrow v_j \quad \text{then} \quad i < j \qquad \text{for all} \;\; i, j \leq n}$$

(*i.e.* all the arrows point 'downstream' from v_1 to v_n)



Topological Sorting

Topological Sort

Step 1) Select any unsorted node $u \in V$ and add u to a stack.

Step 2) While the stack is not empty

Step 2.1) Identify the vertex u at the top of the stack

(but do not remove yet!)

Step 2.2) If $\textbf{Adj}(u)$ is empty or have all been visited, pop u from the stack and add u to the front of the sorted queue Q,

Step 2.3) Else, add all unsorted successors $(\textbf{Adj}(u) - Q)$ to the top of the stack

Step 3) Repeat from Step 1 until all vertices are sorted.

(this implementation employs a Depth-First-Search strategy)

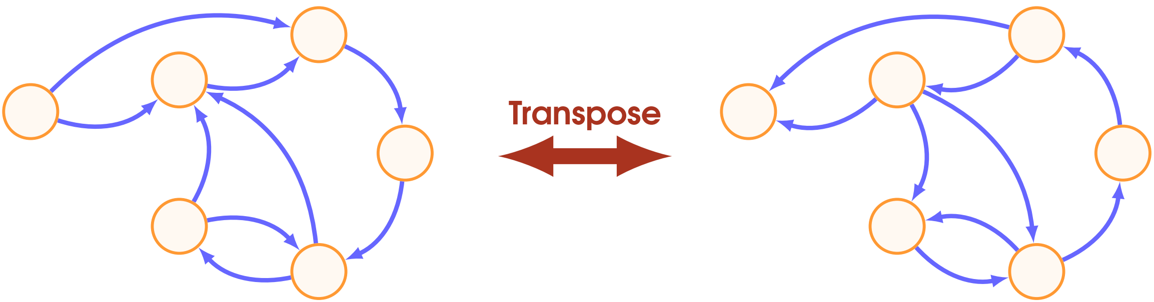
Strongly Connected Components

Transpose Graph

- **Transpose Graph**
  - The **transpose** of a graph $G = (V, E)$ is the directed graph $G^{\mathsf{T}} = (V, E^{\mathsf{T}})$, where

$$\boxed{(a, b) \in E^{\mathsf{T}} \quad \Longleftrightarrow \quad (b, a) \in E}$$

(*i.e.*, G^{T} is the same as G with all the arrows reversed)

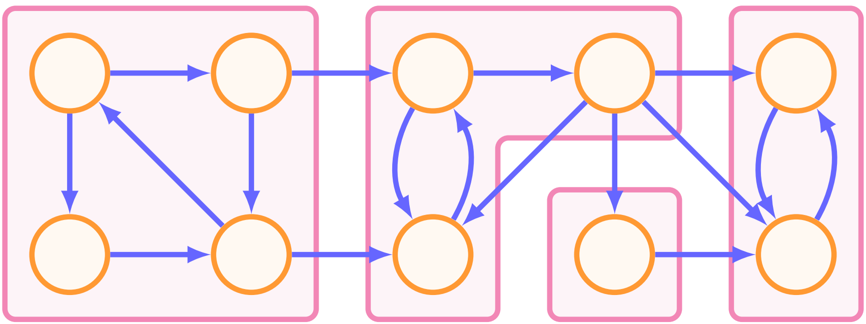


Strongly Connected Components

- **Strong Connected Component (SCC)**
  - A **strongly connected component** of a graph $G = (V, E)$ is a subset $C \subseteq V$, such that

    there is a path $u \rightsquigarrow v$ and a path $v \rightsquigarrow u$, for all $u, v \in C$

    (single nodes can be their own SCCs)

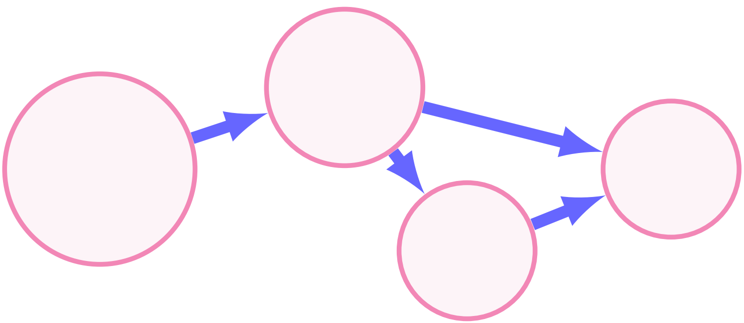


Strongly Connected Components

- **Component Graph**

  - The **component graph** of a graph $G = (V, E)$ is a new graph $G^{\text{scc}} = (V^{\text{scc}}, E^{\text{scc}})$ whose vertices are the strongly connected components of G, and

$(A, B) \in E^{\text{scc}}$	$\iff$	$A \neq B$ and there is some $a \in A$ and $b \in B$, such that $(a, b) \in E$

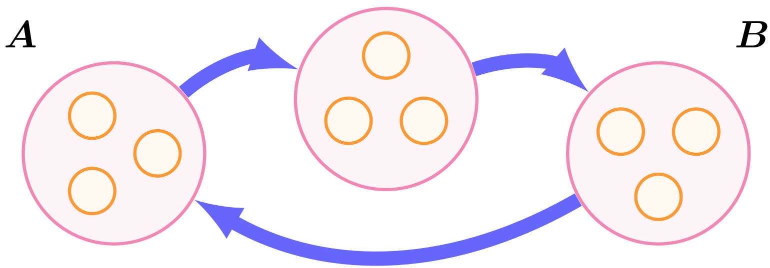


Strongly Connected Components

> **Theorem** The component graph G^{SCC} is a Directed Acyclic Graph.

Proof:

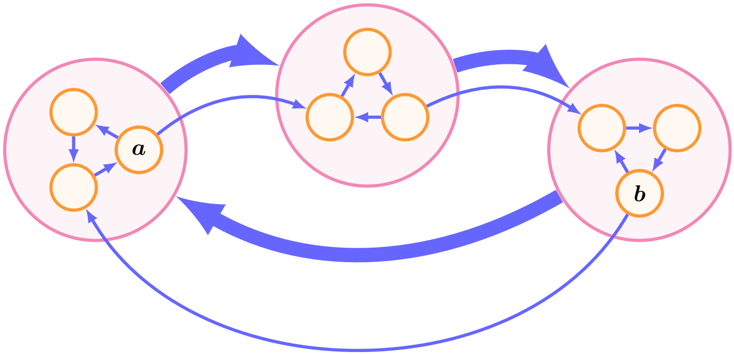
Step 1) Suppose, for contradiction, that there is a **cycle**:



(let A and B to two *distict* SCCs in the cycle)

Strongly Connected Components

Step 2) Let $a \in A$ and $b \in B$, then by definition there must be a path from a to b, and from b back to a,



Step 3) Therefore, a and b must belong to the same SCC, despite choosing $A \neq B$.

Q.E.D.

Strongly Connected Components

Step 1) Perform a topological sort on the graph G to obtain an ordering $Q = \langle v_1, \ldots, v_n \rangle$.

(this will not be a *true* topological sort due to possible cycles)

Step 2) While the queue Q is non-empty

Step 2.1) Dequeue the first (ungrouped) element u from Q and initialise a new component $S = \{u\}$

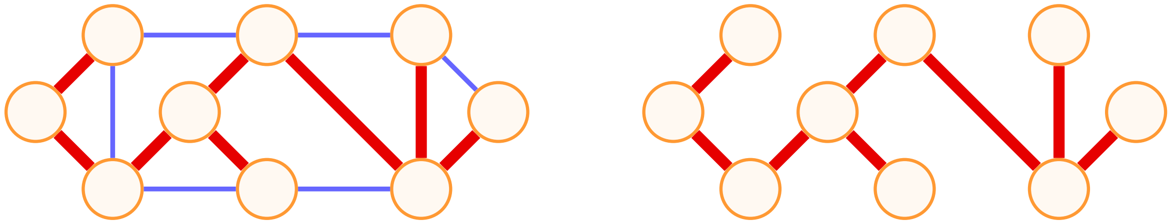
Step 2.2) Perform a DFS from u on the *transpose graph* G^{T}, adding all newly discovered vertices to S.

Step 2.3) Add S to a list of Strongly Connected Components and repeat from Step 2.

Minimum Spanning Tree Algorithms

Minimum Spanning Trees

- **Spanning Tree**
  - A **spanning tree** for a weighted graph $G = (V, E, w)$ is a *tree* $T = (V, E')$ in which each vertex is connected and $E' \subseteq E$,



- **Minimum Spanning Tree (MST)**
  - A **minimum spanning tree** is a spanning tree whose *weight* is minimal out of all possible spanning trees

Kruskal's Spanning Tree Algorithm

Kruskal's Algorithm

Step 1) Sort the edge list $\boldsymbol{E}$ by weight (shortest first)

Step 2) Initalise the set $\boldsymbol{T} = \emptyset$,

Step 3) Dequeue shortest edge $(\boldsymbol{u}, \boldsymbol{v})$ from $\boldsymbol{E}$:

Step 3.1) If $\boldsymbol{T} \cup \{(\boldsymbol{u}, \boldsymbol{v})\}$ is *acyclic*, set $\boldsymbol{T} := \boldsymbol{T} \cup \{(\boldsymbol{u}, \boldsymbol{v})\}$.

Step 4) Repeat from Step 3.

(we need to also consider how to *efficiently* test whether $T \cup \{(u, v)\}$ is acyclic)

Kruskal's Spanning Tree Algorithm

Theorem The worst-case running time for Kruskal's Algorithm is $O(|E| \log |E|)$.

Proof: (sketch)

- **Step 1** takes the longest time and it responsible for the $O(|E| \log |E|)$ upper bound on the running time

$$\boxed{\text{time to sort an array of size } n \;=\; O(n \log n)}$$

- **Step 2** takes constant time $\boxed{\text{step } 2 = O(1)}$

- In **Step 3** we dequeue each edge at most once $\boxed{\text{step } 3 = O(|E|)}$

Kruskal's Spanning Tree Algorithm

- **Step 3.1** can be performed by checking whether u and v are already connected, since a second connection would create a cycle.

$$\boxed{\text{time to check connectedness} \;=\; O(|V|\,\alpha(|V|))}$$

where α is a *very very!* slow growing function that we can forget about.

(see Cormen et al. Section 21.3 for more details)

- Hence, the total **worst-case running time** is given by

$$\boxed{O(|E|\log|E|) + O(1) + O(|E|) + O(|V|\alpha(|V|)) \;=\; O(|E|\log|E|)}$$

Q.E.D.

Prim's Spanning Tree Algorithm

Prim's Algorithm

Step 1) Select a root node $r \in V$,

Step 2) Initialise the set $T = \emptyset$,

Step 3) Add the adjacent edges $\mathbf{Adj}(r)$ to a *priority queue* Q

Step 4) Dequeue the shorted edge (u, v) from the queue,

- **Step 4.1)** If u and v are disconnected in T, set $T := T \cup \{(u, v)\}$.
- **Step 4.2)** Add to the queue Q any new edges adjacent to u or v.

Step 5) Repeat from Step 4.

End of Slides!