Week 5

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 $v \rightsquigarrow u$

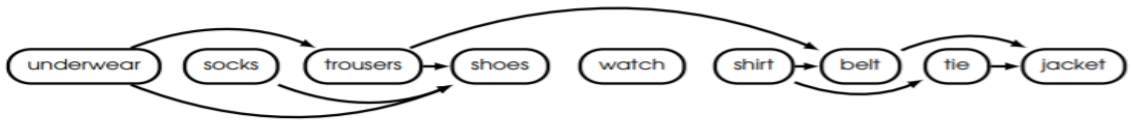
(for all vertices $u, v \in V$)

Topological Sorting

- 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

> If $v_i \rightsquigarrow v_j$ then $i < j$ for all $i, j \leq n$

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



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 **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 (**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 strategy.

We push and pop from Stack when we do DFS and then we add them to the queue to get the sorted portion finished. So, the way to do the algorithm is to pick a random place on the graph and do DFS onto the stack, when there are no more successor then we can pop the node of stack and enqueue it onto the queue, the place where u start from does no matter as the connections should also be the same.

Week 5

Transpose Graph

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

$$(a, b) \in E^T \iff (b, a) \in E$$

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

Strongly Connected Components

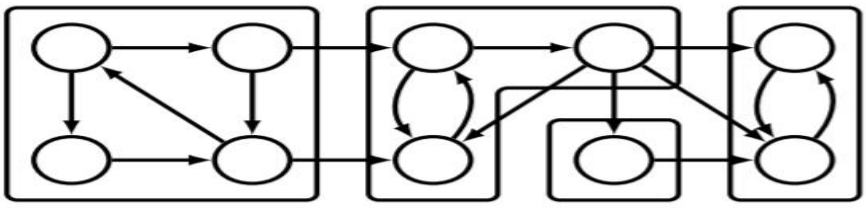
Strongly Connected Components are all those little islands where you can visit all nodes within the cluster, but as soon as you leave you can never go back, you can still move freely.

Any subset where all the things in them there is a path between any two, it does not have to be direct path, it just means that If I pick u and I pick v from the cluster c then you can always go from one to another and get back again.

- **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 $u, v \in C$

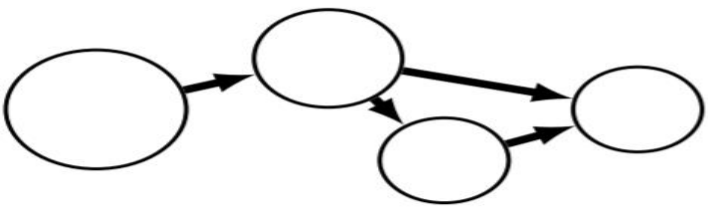
(single nodes can be their own SCCs)



Strongly Connected Components

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

$$(A, B) \in E^{SCC} \iff \begin{array}{l} A \neq B \text{ and there is some } a \in A \\ \text{and } b \in B, \text{ such that } (a, b) \in E \end{array}$$



Week 5

Strongly Connected Components Algorithm

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.

In component graph there should never be a cycle because it would automatically indicate that the nodes are in the same strongly connected components.

1) Perform standard topological sort,
2) When topological sort is done, take the sequence and start DFS from last node don't forget that graph must be transpose. Perform DFS until there are no more successors at that time we can form a strongly connected components from all nodes that are currently on queue.

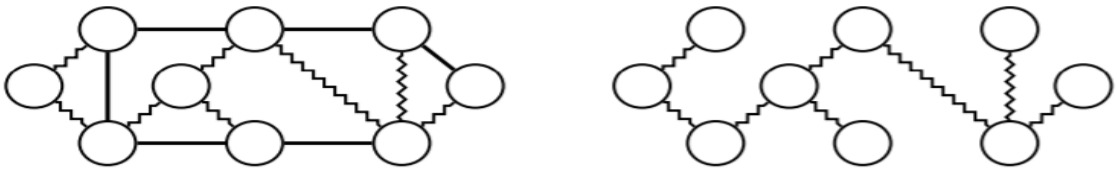
Why do we use DFS and not BFS, does it make a difference?

- There is no specific reason for a particular search algorithm, BFS could work perfectly well with the same time complexity.

Week 5

Minimum Spanning Tree Algorithms

- **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

- This is a path that visits all nodes without any cycles.
- There can be many minimum spanning trees
- Can brute-force it, on small graph.

Kruskal's Spanning Tree Algorithm

T is a minimal spanning tree

So basically, when we start to generate MST write down all nodes that must be visited, and in every iteration cluster them together until they don't form a path which does not include any cycles.

Kruskal's Algorithm

Step 1) Sort the edge list E by weight (shortest first)

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

Step 3) Dequeue shortest edge (u, v) from E:

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

Step 4) Repeat from Step 3.

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

If it does not add a cycle then we can take it and add it to the set, if it adds cycle then we disregard it.

Prim's Spanning Tree Algorithm

T = minimal spanning tree

- It does the same thing as Kruskal algorithm.
- Add nodes to the priority queue when they don't create cycles and they have lightest weights.

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.