# Lecture 2 Topological Sort and MST

A recurring theme in asymptotic analysis is that it is often possible to get better asymptotic performance by maintaining extra information about the structure. Updating this extra information may slow down each individual step; this additional cost is sometimes called *overhead*. However, it is often the case that a small amount of overhead yields dramatic improvements in the asymptotic complexity of the algorithm.

To illustrate, let's look at *topological sort*. Let $G = (V, E)$ be a directed acyclic graph (dag). The edge set $E$ of the dag $G$ induces a *partial order* (a reflexive, antisymmetric, transitive binary relation) on $V$, which we denote by $E^*$ and define by: $uE^*v$ if there exists a directed $E$-path of length 0 or greater from $u$ to $v$. The relation $E^*$ is called the *reflexive transitive closure* of $E$.

**Proposition 2.1** *Every partial order extends to a total order (a partial order in which every pair of elements is comparable).*

*Proof.* If $R$ is a partial order that is not a total order, then there exist $u, v$ such that neither $uRv$ nor $vRu$. Extend $R$ by setting

$$R \; := \; R \cup \{(x, y) \mid xRu \text{ and } vRy\} \; .$$

The new $R$ is a partial order extending the old $R$, and in addition now $uRv$. Repeat until there are no more incomparable pairs. □

In the case of a dag $G = (V, E)$ with associated partial order $E^*$, to say that a total order $\leq$ extends $E^*$ is the same as saying that if $uEv$ then $u \leq v$. Such a total order is called a *topological sort* of the dag $G$. A naive $O(n^3)$ algorithm to find a topological sort can be obtained from the proof of the above proposition.

Here is a faster algorithm, although still not optimal.

**Algorithm 2.2 (Topological Sort II)**

1. Start from any vertex and follow edges backwards until finding a vertex $u$ with no incoming edges. Such a $u$ must be encountered eventually, since there are no cycles and the dag is finite.
2. Make $u$ the next vertex in the total order.
3. Delete $u$ and all adjacent edges and go to step 1.

Using the adjacency list representation, the running time of this algorithm is $O(n)$ steps per iteration for $n$ iterations, or $O(n^2)$.

The bottleneck here is step 1. A minor modification will allow us to perform this step in constant time. Assume the adjacency list representation of the graph associates with each vertex two separate lists, one for the incoming edges and one for the outgoing edges. If the representation is not already of this form, it can easily be put into this form in linear time. The algorithm will maintain a queue of vertices with no incoming edges. This will reduce the cost of finding a vertex with no incoming edges to constant time at a slight extra overhead for maintaining the queue.

**Algorithm 2.3 (Topological Sort III)**

1. Initialize the queue by traversing the graph and inserting each $v$ whose list of incoming edges is empty.
2. Pick a vertex $u$ off the queue and make $u$ the next vertex in the total order.
3. Delete $u$ and all outgoing edges $(u, v)$. For each such $v$, if its list of incoming edges becomes empty, put $v$ on the queue. Go to step 2.

Step 1 takes time $O(n)$. Step 2 takes constant time, thus $O(n)$ time over all iterations. Step 3 takes time $O(m)$ over all iterations, since each edge can be deleted at most once. The overall time is $O(m + n)$.

Later we will see a different approach involving depth first search.