1 Directed Graphs

A directed graph can be thought of as a graph of dependencies. For example, consider the following activities:

```
Breakfast                    Go to work

          Wake up

Shower                       Dress
```

Of course, you can't do all five of these activities at the same time; there are some dependencies. Some of them are:

- Wake Up → Breakfast → Go to Work.
- Wake Up → Go to Work.
- Wake Up → Shower → Dress → Go to Work.

In other words, you need to do A before you can do B, and so on. One of the things we might want to do is order this graph in a way that respects these dependencies.

1.1 Topological Ordering

Essentially, a directed graph can be thought of as a graph of dependencies. An edge $v \mapsto w$ means that v should come before w. We can use something known as topological ordering to better understand this relationship.

Definition 1.1: Topological Ordering

A **topological ordering** of a directed graph is an ordering of the vertices so that for each edge (v, w), v comes before w in the ordering.

Remark: There can be multiple different topological orderings for the same graph.

A question that we might have is: does every directed graph have a topological ordering?

- No. Consider the classic counterexample:

$$\text{Chicken} \mapsto \text{Egg}$$

$$\text{Egg} \mapsto \text{Chicken}$$

  In other words, there is a *cycle* where the chicken goes to the egg and the egg goes back to the chicken.

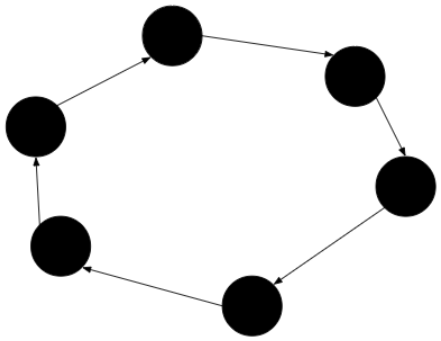
1.2 Cycles

Definition 1.2: Cycle

A cycle in a directed graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ so that there are edges:

$$(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$$

For example, here is a cycle:



1.2.1 Obstacle

Proposition. *If G is a directed graph with a cycle, then G has no topological ordering.*

So, in other words, if G is a directed graph with at least one cycle *anywhere*, then it has no topological ordering.

Proof. Suppose we have a cycle $v_1, \ldots, v_n$. Assume for the sake of contradiction taht we have an ordering. Then, there are n vertices so one of them has to come first; say that v_i came first in the ordering. But, this is a cycle so there is an edge from v_{i-1} to v_i. However, because v_i was first so it must come first in the order, in contradiction to the order property. □

1.3 Directed Acyclic Graph (DAG)

Suppose we want to focus on directed graphs with no cycles. This brings us to the following definition.

Definition 1.3

A **directed acylic graph** (DAG) is a directed graph which contains no cycles.

The previous result said that only DAGs can be topologically ordered. However, is the reverse true? That is, is it the case that every DAG has a topological ordering? **Yes**.

1.4 Existence of Orderings

Theorem 1.1

Let G be a (finite) DAG. Then, G has a topological ordering.

Proof. We consider the last vertex in the ordering; in other words, whatever vertex comes last in this ordering. This vertex must be a sink, or a vertex with no outgoing edges. So, once we find the sink, we can put the graph at the end of the topological graph, and then order the remaining vertices. Now, using the lemma below, we want to show that every DAG has a topological ordering. We will use induction on $|G|$. Omitting the base case, we find a sink v. Then, create a graph $G' = G \setminus \{v\}$. We can inductively order G', which is still a DAG. Then, add v to the end of the ordering. □

1.4.1 Sinks

Lemma 1.1

Every finite DAG contains at least one sink.

Proof. Start at a vertex $v = v_1$. Then, we can "follow the trail," or in other words follow the edges (v_1, v_2), (v_2, v_3), ... Eventually, we will either find:

- Some vertices repeat (which creates a cycle). This can't happen if this is a DAG, though.
- Or, we get stuck (which means we found a sink).

So, we are done. □

1.5 Algorithm

Suppose we want to design an algorithm that, given a DAG G, computes a topological ordering on G. We can use the proof to create a naive algorithm.

- Find a sink v. This is done by following a chain of vertices until we are stuck.
- Compute the ordering on $G - \{v\}$.
- Place v at the end.

This algorithm can be written like so:

```
TopologicalOrdering(G)
    If |G| = 0
        Return {}
    Let v in G
    While there is an edge (v, w)
        v = w
    Return (Ordering(G - v), v)
```

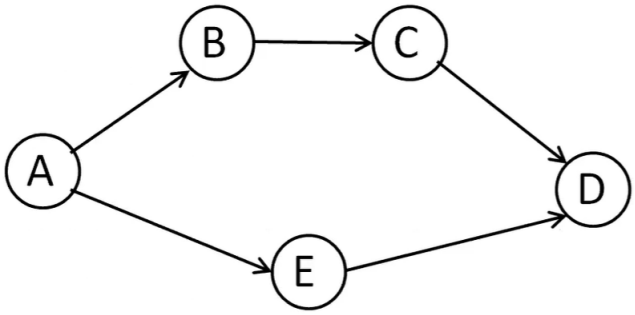
The runtime is $O(|V|^2)$. This is because we need $|V|$ time to find each sink and have $|V|$ sinks. This is suboptimal, however. Consider this optimal algorithm:

```
TopologicalOrdering(G)
    Run DFS(G) w/ Pre/Post Numbers
    Return Vertices in Reverse Postorder
```

This runs in $O(|V| + |E|)$.

1.5.1 Example: DAG

Consider the following DAG which we will call G:



We start at A since this is where the graph "begins."

- Using Algorithm 1:
  - First, we enumerate through the graph G until we find a sink. In this case:
    $$A \mapsto B \mapsto C \mapsto D$$
  - We *remove* D from the graph and add it to the ordering.
    ```
    D
    ```
  - Now, we enumerate through the graph $G - \{D\}$. In this case:
    $$A \mapsto B \mapsto C$$
  - We *remove* C from the graph and add it to the ordering.
    ```
    C -> D
    ```
  - Now, we enumerate through the graph $G - \{C, D\}$. In this case:
    $$A \mapsto B$$
  - We *remove* B from the graph and add it to the ordering.
    ```
    B -> C -> D
    ```
  - Now, we enumerate through the graph $G - \{B, C, D\}$. In this case:
    $$A \mapsto E$$
  - We *remove* E from the graph and add it to the ordering.
    ```
    E -> B -> C -> D
    ```
  - As we only have A left, we can add it to the ordering.
    ```
    A -> E -> B -> C -> D
    ```

    So, we are done.
- Using Algorithm 2:
  - First, we enumerate through the graph G until we find a sink. In this case:
    $$A \mapsto B \mapsto C \mapsto D$$
  - We remove D from the graph and add it to the ordering.
    ```
    D
    ```
  - Now, we remember that we already made it to C and that C is a sink in $G - \{D\}$. So, we remove C and add it to the ordering:
    ```
    C -> D
    ```

– We remember that we already made it to B and that B is a sink in $G - \{C, D\}$. So, we remove B and add it to the ordering:

```
B -> C -> D
```

– Now that we're back at A, we note that A has another edge. So, we go to that edge until we find a sink. In this case:

$$A \mapsto E$$

– We remove E from the graph and add it to the ordering.

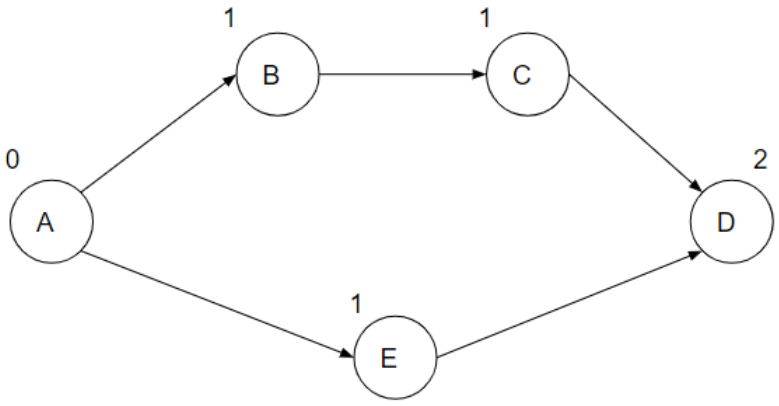
```
E -> B -> C -> D
```

– Now, A is the only vertex left in $G - \{B, C, D, E\}$. So, we add it to the ordering:

```
A -> E -> B -> C -> D
```

Here, we notice that algorithm 2 resembles depth-first seearch!

Another way to find a topological ordering for this graph is to consider the in-degrees of each vertex; i.e. the number of edges connecting **to** it. If we label each of G's vertices with its in-degrees, we will have:



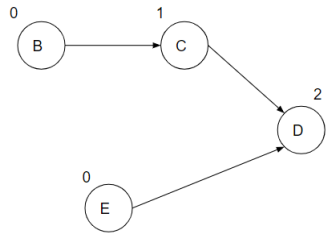
So, this makeshift algorithm is essentially:

```
TopologicalOrdering(G)
    ordering = []
    while G is not empty
        v = vertex in G with lowest in-degree (i.e. in-degree 0)
        ordering.append(v)
        Remove v from G
        Update in-degrees of existing vertices
    return ordering
```

- First, note that A has the lowest in-degree. So, remove A from the graph and add it to the ordering. The ordering now looks like:

```
[A]
```

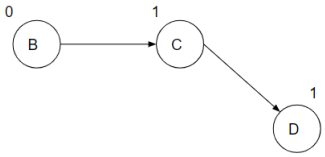
- The graph now looks like:



Here, B and E both have the lowest in-degree. So, we can remove one or the other. For the sake of consistency, remove E from the graph and add it to the ordering. The ordering now looks like:

```
[A, E]
```

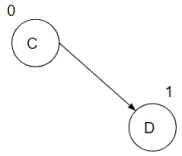
- The graph now looks like:



Here, note that B is the only node with the lowest in-degree. So, we remove it and add it to the ordering. The ordering now looks like:

```
[A, E, B]
```

- The graph now looks like:



Here, C has the lowest in-degree so we remove it and add it to the ordering. The ordering now looks like:

```
[A, E, B, C]
```

- By now, it's trivial to see that D has in-degree 0, so we add it.

```
[A, E, B, C, D]
```

And we are done.

1.6 Topological Sort

This is a particularly useful algorithm.

- Many graph algorithms are relatively easy to find the answer for v if you've already found the answer for everything downstream of v.
  - We can topologically sort G.
  - Then, solve for v in reverse topological order.

1.7 Connectivity in Digraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component. Well, this no longer works for digraphs.