Header styling inspired by CS 70: https://www.eecs70.org/

# 1 Graphs

## 1.1 DFS

- Explore unvisited nodes with some tiebreaking algorithm, and add unvisited ones to a stack. Classifies edges, so for an edge $(u, v)$ (i.e. $u \to v$ ):
  - Forward edge: edge explored in DFS tree (same as a tree edge mathematically): $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.
  - Back edge: an child to an ancestor in DFS tree: $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$.
  - Cross Edge: neither a child or an ancestor (unrelated nodes) $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$

    Which one of these requires that the edge $(u, v)$ exists in the graph?
- Runtime: $O(n + m) = O(|V| + |E|)$

## 1.2 DAGs

- Directed acyclic graphs: directed graphs with no cycles basically.
- There are $\binom{n}{2}$ edges, beause every pair of vertices may only be chosen once.
- DAGs are special because they can always can be **topologically sorted:** arrange the vertices left to right in such a way that outgoing edges only go to the right.
- Topological sort: sort based on decreasing postorder (from left to right). The smallest postorder must be a sink (all edges lead to $v$ ), and largest postorder must be a source (all edges lead away from $v$).

  Do I need to study the proof of this?

## 1.3 SCCs

- Two vertices are strongly connected if there exists a path from $v \to u$ and also a path from $u \to v$. This requires the graph to have a cycle (by definition)
- The *meta graph* (graph of SCCs) must always be a DAG. This is because had there been a cycle between two SCCs, this implies that the two SCCs should be combined together to form a new SCC.
- Kosaraju's algorithm: reverse all edges in graph, then run DFS on the graph in decreasing postorder. Each end of the explore call will find a new SCC every time.

## 1.4 Path finding algorithms

- Given a graph $G = (V, E)$ and edge weights $e_i$, what is the shortest path from a node $u$ to another $v$?
- BFS: A way to find the shortest path between $u$ and $v$ given that all edge weights are 1. $O(m + n) = O(|V| + |E|)$

- Dijkstra's: A way to find the shortest $u \to v$ given that all edge weights are positive. $O(n \log n + m) = O(|V| \log |V| + |E|)$. This heavily depends on the implementation of Dijkstra's, this runtime is only possible given we use a fibonacci heap.
- Bellman-Ford: A way to find the shortest $u \to v$ path with no restrictions on edge weights. $O(nm) = O(|V||E|)$. The only restriction is that we can't have negative weight cycles, since then the notion of a shortest path is broken.

# 2 Greedy Algorithms

- An algorithm that during runtime, makes the most naively optimal choice every time (i.e. greedy).
- We generally use induction to prove that greedy is always optimal.
- Class scheduling: given a set of classes, what is the maximum we can fit in a room? We solve this by adding based on earliest end time.

## 2.1 Huffman Coding

- Given a text with letter frequencies, what's the minimum encoding we can encode with such that a unique message is recoverable?
- Generate a tree based on frequencies: the most frequent letter should use the least bits, and so on. Greedy solution here is in fact optimal!
- The cost function can be calculated as:
$$\text{cost} = \sum_i f_i \cdot d(i)$$
where $d(i)$ represents the depth at which we find character $i$.
- Specifically, it creates a priority queue based on frequency and merges least frequent nodes, adds them together and puts them back in the queue.
- Runtime: $O(n \log n)$ when utilizing a binary heap.

## 2.2 Horn-SAT

- Satisfiability problem: given a set of clauses $x_i \vee x_j \cdots \vee x_k$ and a list of implications (stuff) $\implies x_k$, we want to know whether this is satisfiable.
- Greedy solution: for every clause of the form (stuff) $\implies x_k$, set $x_k$ to true. Do this for every clause. Then, check if all pure negative clauses ($x_i \vee x_j \cdots$) are satisfied. If yes, then return true, otherwise the instance is not satisfiable.
- The reason this outputs an optimal solution is the fact that in order for the instance to be satisfiable, we do the "minimum" we can possibly do to satisfy the clauses, or in other words setting $x_k$ to true for every implication we find. Because at every step we're doing the *minimum possible*, this ends up being the best optimal solution.

## 2.3 MSTs

A minimum spanning tree is basically a tree of a graph $G = (V, E)$ that contains all vertices $v \in V$, and has minimum sum of edge weights. There could be multiple MSTs, given that the edge weights sum up to the same thing (for instance, consider a complete graph $G$ whose edge weights are all 1. There are two main MST-finding algorithms that we need to know: Prim's and Kruskal's algorithm:

- Kruskal's Algorithm: start with an empty set, and always add the minimum weight edge $e$ such that adding that edge doesn't create a cycle. Repeat this until all vertices are covered, and we have an MST.
- Prim's Algorithm: start with a set $S = \{v\}$ and consider the set $T = V \setminus S$, of all the edges from $S$ to $T$, consecutively add the minimum weight edge.

The most important thing about MSTs is the **cut property:** Given that we have a set of edges $X$ that's already in the MST, then the lightest weight edge that connects $X$ to $V \setminus X$ is part of the MST as well. This implies that we can create a meta algorithm:

- Start with an empty set, and pick $S \subseteq V$ such that $X$ has no edges from $X$ to $V \setminus X$.
- Add the lighest weight edge from $X$ to $V \setminus X$.

Any algorithm that follows this meta algorithm is guaranteed to find an MST. The thing that algorithms do differently is how they pick $S$, and then correspondingly how they pick $e$. For Prim's, it's fairly obvious how this is chosen, and for Kruskal's, the condition that the added edge $e$ doesn't form a cycle is specifically what guarantees that the edge crosses from $X$ to $S \setminus X$.