# CS561 HW11

Ryan Scherbarth

November 2024

1. **Prove by induction that any tree with $n$ nodes has exactly $n-1$ edges. Don't forget to include the Base Case (BC), Inductive Hypothesis (IH), and Inductive Step (IS).**

**Base case:** $n = 1$
By definition, and undirected tree is defined as a collection of vertexes, $v$, and edges, $(u, v)$. An undirected edge is a path joining together two vertexes.

In the case where we have $n = v = 1$ vertexes, it is not possible for us to create a single edge, since by definition it must join two unique vertexes on an undirected graph. Therefore, we will have $n - 1 = 1 - 1 = 0$ edges, so our base case holds true.

**Inductive Hypothesis:**
For all $k < n$, the number of edges in a tree of size $k$ is exactly $k - 1$.

**Inductive Step:**
Let $T$ be a connected, acyclic graph with $n$ nodes. Due to the properties of this graph, proving that there exists at least 1 leaf node in any case will also show that there exists exactly $n - 1$ edges.

We will start by creating a new tree, $T_2$, which results from removing any single node and corresponding edge from $T$ s.t. $T_2$ is still a valid tree. Invoking our IH on $T_2$, we can say that this tree has exactly $n - 1 - 1 = n - 2$ edges.

Say we want to increase now to $n$ nodes again, in order to follow the rules of an acyclic graph it is only possible to add exactly 1 edge. We know this because adding 0 lines will result in the graph being unconnected, failing the second claim.

An edge is defined as a connection between two nodes (we cannot have an edge connected to itself in an undirected graph). We can then see that it will not be possible to add any number greater than 1 edge after increasing the number of nodes by 1 without introducing a cycle, which fails the second claim we've made about the graph.

Therefore, we have proven that when introducing a any new node, it is not possible to add less than 1 edge, and it is not possible to add any more than 1 edges. Thus, the proof holds.

2. **Prove Claim 1 from Single Source Shortest Paths Lecture (SSSP Lecture).**

**claim 1: if $dist(v) \neq \infty$, then $dist(v)$ is the total weight of the predecessor chain ending at $v$.**

We will prove claim one via induction on the defined variable $s = length(pred(v))$.

**Base case:** $s = 0$
When the length of $pred(v) = 0$, then we have no predecessors, so $s = v$. In this case, the claim holds true.

**Inductive Hypothesis:**
Assume that for any node $u$ with a predecessor chain of length $k$, s.t. $k < \ell$, the distance $dist(u)$ is the total weight of the predecessor chain from $s$ to $u$.

**Inductive Step:**
Let node $v$ have a predecessor chain of length $\ell$. We can use the inductive hypothesis to determine that $dist(pred(v))$ is the total weight of the predecessor chain from $s$ to $v$.

Let $w(prev(v), v)$ be the weight of the edge from $pred(v)$ to $v$. Then we can say
$dist(v) = dist(pred(v)) + w(pred(v), v)$.

By the inductive hypothesis, $dist(pred(v))$ is the sum of weights along the pred chain, adding $w(pred(v))$ gives the new total weight fo the path from $s$ to $v$, therefore, via proof by induction, claim 1 holds.

3. **Prove claim 2 from Single Source Shorted Paths Lecture (SSSP Lecture)**

**Claim 2: If the algorithm halts, then $dist(v) \leq w(s \to v)$ for any path $s \to v$.**

**Base case:** $n = 1$
If the path $s \to v$ contains no edges, meaning $v = s$, then $dist(v) = dist(s) = 0$. By definition, $w(s \to v) = 0$ since the weight function should be the summation of each edge. So $dist(v) \leq w(s \to v)$.

**Inductive Hypothesis:**
Assume that for any path $s \to u$ with $k$ edges, $dist(u) \leq w(s \to u)$, where $dist(u)$ gives the weight of the shortest distance from $s$ to $u$ when the program has halted.

**Inductive Step:**
Let the path $s \to v$ have $k + 1$ edges, and $u$ be the predecessor of $v$. Invoking the IH we can say $dist(u) \leq w(s \to u)$.

For each iteration, we relax an edge $(u, v)$, which means $dist(v)$ is the new minimum distance to that node.

By the IH, $dist(v) \leq w(s \to u) + w(u, v)$ for our node. since $w(s \to u) + w(s \to v) = w(s \to v)$, we can say that upon the algorithm halting, $dist(v) \leq w(s \to v)$.

4. **24-3 Arbitrage**

**Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 India rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 * 2 * 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent. Suppose that we are given $n$ currencies, $c_1, c_2, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i, j]$ units of currency $c_j$.**

(a) **Give an efficient algorithm to determine whether or not there exists a sequence of currencies $< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$ such that $R[i_1, i_2] * R[i_2, i_3] \ldots R[i_{k-1}, i_k] * R[i_k, i_1] > 1$. Analyze the running time of your algorithm.**

To determine if it is possible to make some amount of profit via any possible combination of exchanges we create a graph $G(V, E)$ where each vertex is a currency, and edge has the weight function $w(i, j) = -1 * \log(R[i, j])$. We are able to use this weight function so that we can use the property of summing weights of a cycle to detect multiplicative properties.

We will next use the Bellman-Ford starting from any currency to check for negative-weight cycles. if a negative cycle exists, it tells us there is a possible arbitrage opportunity over that cycle. The running time will be bounded by BF, with a time complexity of $O(n^3)$ for $n$ currencies.

(b) **Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.**

The Bellman-Ford algorithm guarantees that if there is a negative cycle possible in our graph, it will be made evident after no more than $n$ iterations. Once we've found such a path, we will start at that node and work backward. We can take a snapshot of the graph, and then use the predecessor array to walk backwards through the cycle, printing the sequence of nodes in the arbitrage loop. Bellman-Ford guarantees the shortest path up to the $n$th iteration, meaning this sequence can be at most $n$ paths.

5. **Saia Trucking is a very safety conscious - and algorithm loving - trucking company. Given a pair of cities, they always try to find the safest route between that pair. They are thus faced with the following problem.**
**There is a directed graph $G = (V, E)$, where the vertices represent cities and the edges represent roads. Each edge has a value associated with it that gives the probability of safe transport on that edge, i.e. the probability that there will be no accident when driving across that edge. The probability of safe transport along any path in the graph is the product of the probabilities of safe transport on each edge in that path.**
**The goal is to find a path from a given node $s$ to a given node $t$ that maximizes the probability of safe transport. Describe an efficient algorithm to solve this problem.**

We have a directed graph $g$, each edge being given a safety probability $p(u, v)$. We can use a log transformation and minimize the negative of the sum of logs, so we will set our new weight to be $w(u, v) = -\log(p(u, v))$. After the transformation we now have a shorted path problem which we can solve.

We can use Dijkstra's algorithm to solve for the shortest path from $s$ to $t$ so long as we have $w(u, v) \geq 0$. After running Dijstra's algorithm, the shortest path from $s$ to $t$ in our transformed graph corresponds to the path with the maximum probability of safe transport in the original graph.

Dijstra's algorithm works by using a queue, selecting each node iterating through starting at the smallest known distance, and marking visited nodes. We go through all of the vertices and edges in this manner. Our algorithm will ultimately be determined by this runtime, which is given by $O(|E| + |V| \log(|V|))$.