CSE 421 HW2

Rohan Mukherjee

April 9, 2024

1. We first prove the base case. The only two possible trees with vertices labeled 1 and 0 with 2 vertices are, where the sum of the vertices is even, are:

   $$0 \text{ —— } 0$$

   $$1 \text{ —— } 1$$

   In the first case, we can take F to be the empty set, and in the second we can take $F = E$.

   Suppose the statement is true for all trees with n vertices, and let T be an arbitrary tree of $n+1$ vertices. Find a leaf v of the tree. If the label of v is 0, removing v from our tree will result in another tree, who's label-sum is also even since we removed 0 from it. Thus we can find a subset of the edges F so that every vertex with label 0 is adjacent to an even number of edges and every vertex with label 1 an odd number of edges. Now, the vertex v with label 0 has 0 edges adjacent to it, which is an even number, so we are done by taking the above F.

   If v instead had label 1, we have two cases. Since v has degree precisely 1 let v' be its parent (the node that it is adjacent to). Flip v''s label, so that if v' was labeled 1 it would now be labeled 0 and vice versa. Now remove v from the tree, and let the remaining tree $T' = T \setminus v$. We see that T' still has an even label sum, since if v' was 1 we would flip it to 0, removing 1, then remove v, removing 1 again and still have an even number, and similarly if v' was 0 then we would remove v, removing 1, and change v' to 1, adding 1, still resulting in an even number. Now let F be the subset where every vertex with label 1 has an odd number of adjacent edges and every vertex with label 0 has an even number. By adding the edge connecting v with v', we first see that v satisfies the property we need. By our construction, we flipped the parity of v', so we get the opposite parity of adjacent edges needed. After this we added precisely 1 to this parity, flipping it back, so we have the right parity for v' as well, which completes the proof.

2. We first prove the following lemmata:

Lemma 1. *If G's edges can be partitioned into disjoint edge-cycles then every vertex in G has even degree.*

Proof. Suppose that $G = (V, E)$'s edges can be partitioned into disjoint-edge cycles, say as

$$E = \coprod_{i=1}^{k} E_i.$$

Clearly, every vertex in a cycle has degree precisely equal to 2. After removing the edges in E_i from E, the vertices that the edges in E_i touch go down by precisely 2, and the other vertices are unchanged. Since the E_i are disjoint, and cover all of E, after removing all the E_i, all vertices have degree 0. Since removing each E_i reduces the degree of the vertices in the cycle by 2, we see that every vertex in G has even degree. □

and,

Lemma 2. *If every vertex in G has degree ≥ 2, then G contains a cycle.*

Proof. Suppose that G has at least one edge, and thus let v_1 be a vertex of degree > 0. We can construct a cycle as follows: let v_2 be an edge adjacent to v_1, and since $\deg v_2 \geq 2$, we can find v_3 that is adjacent to v_2 that is not v_1. Now, since v_3 has degree ≥ 2, it either has an edge trailing back to one of v_1, v_2, or there is a v_4 that is not v_1 or v_2 that is adjacent to v_3. We can continue this process, and since there are only finitely many vertices, we must eventually reach a vertex that we have already visited, which yields a cycle.

□

The above two lemmata pave the way for a concise polynomial-time algorithm to solve our problem:

```
Function EdgePartition(G):
    if G has a vertex with odd degree then
        Output "Impossible";
        return;
    end
    if every vertex in G has degree 0 then
        return;
    end
    Initialize E ← ∅;
    Let v be a vertex in G with degree ≥ 2;
    while true do
        for each edge (v, u) do
            if u is adjacent to a seen vertex w then
                Add (u, w) to E;
                Remove all vertices of E added before w;
                break;
            end
            Let w be a vertex adjacent to v;
            Mark w as seen;
            Add (v, w) to E;
            v ← w;
        end
    end
    Remove all edges in E from G;
    return E ⨿ EdgePartition(G);
```

The above algorithm works by finding a cycle (if one exists), removing it from the graph, and calling itself recursively. The algorithm is correct first beacuse clearly each of the E_i returned are disjoint, and second beacuse since in a cycle each vertex has precisely degree 2, so when we remove the edges from G, we remove precisely 2 from the degree of each vertex in the cycle, so the degree of each vertex in the remaining graph is still even. This means that unless G is empty after removing the final cycle, we can find another one, hence we have certainly partitioned the edges into disjoint sets.

Let $T(m)$ be the running time of the above algorithm on a graph with n vertices and m edges. The first check can be completed in $O(n)$ time, and if it doesn't return, can save a vertex of positive degree. The second check can be completed in $O(n)$ time. The while

loop termiantes in at most n steps, since the longest cycle in a graph with n vertices is precisely n (since each vertex can only be visited once). The if statement can be completed in $O(n)$ time (in the worst case the w is apoproximately the $(n-3)$rd element seen). Since each cycle has at least 3 edges, we see that

$$T(m) \leq T(m-3) + O(n) \implies T(m) = O(nm).$$

3. We use a slight variation on BFS.

```
Function BFS(discovered, s):
    Initialize an empty queue Q;
    Q.enqueue(s);
    discovered[s] ← true;
    Initialize total ← 0;
    while Q is not empty do
        u ← Q.dequeue();
        total ← total + c_u;
        for each neighbor v of u do
            if v is undiscovered and v is not dead then
                discovered[v] ← true;
                Q.enqueue(v);
            end
        end
    end
    return dist, prev;
Function LCV(G):
    Initialize an empty dictionary discovered;
    Initialize lcv ← 0;
    for each vertex v in G do
        if v is undiscovered and v is not dead then
            lcv ← max(lcv, BFS(discovered, v));
        end
    end
    return lcv;
```

Since BFS will travel to a node iff there is a path from one node to another, we can see that the above algorithm gives us a collectible value since it only adds the value of each node once, and it never visits dead nodes. This variation of BFS will also output the aggregate sum of the nodes that are reachable from the starting node that don't cross dead vertices (since in the for loop we exclude dead vertices). The function LCV then outputs the maximum of these values. It is thus clear that the LCV is $\geq$ the output of the above algorithm. The LCV is less beacuse it is equivalent to the sum of some list $v_1, \ldots, v_k$ whose sum is equal to the LCV. Running BFS on, say, v_1 would either output this sum if $v_1, \ldots, v_k$ are all the elements in the connected component (excluding the dead vertices), or

something higher, which completes the proof of correctness. Finally, the above algorithm is just a BFS, possibly excluding some dead vertices, so the running time can only go down–in any case it is certainly bounded by $O(n + m)$.

4. We want to minimize the product of the weights: given a path $s = v_1, \ldots, v_n = t$, the weight of this path is just

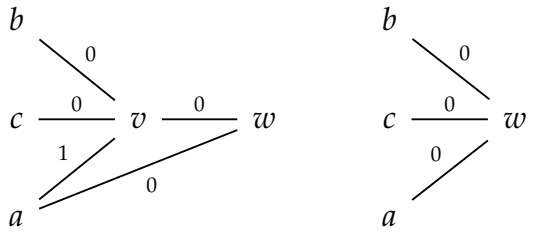
$$\prod_{i=1}^{n-1} w_{(v_i, v_{i+1})}$$

Since $\log(x)$ is an increasing function, this is fully equivalent to minimizing the sum of the logs of the weights:

$$\sum_{i=1}^{n-1} \log\left(w_{(v_i, v_{i+1})}\right)$$

Our algorithm will thus have 3 steps:

(1) Collapse each weight-1 edge. We must define the subroutine collapse(v, w) that takes in two vertices, with the assumption that there is a weight-0 edge between them, removes one of the vertices (say, the one with lower lexicorigraphical ordering) v, and collapses the edge into a vertex vw where all the old edges going to v now point to w with the same weight, where if w was already connected to one of those vertices pointing to v then we break ties by picking the smaller-weight edge. For example,



We define the subroutine collapse(G) by calling collapse on each of its edges. This can be accomplished using an adjacency matrix in $O(n + m)$ time, since given a weight 0 edge (v, w), all m_v of v's edges will have to be transferred to w, and breaking ties can be accomplished in $O(1)$ time since we can just access entry (v, w) in the adjacency matrix and set it to the minimum of the it's current weight and the new proposed weight. Doing this on all n vertices yields a running time of $\sum_{i=1}^{n} m_i = m$, and since we need to access each vertex at least once to collapse its edges, we see that the running time is $O(n + m)$.

We claim that the collapsing procedure preserves the length of the shortest path. We shall show this by showing that given a path of the original vertices, there is a path of the new vertices with the same weight, and vice versa. Suppose that the path is $v_1, \ldots, v_n$. If there is a weight 1 edge between v_i and v_{i+1}, then the collapse procedure will collapse (WLOG)

v_i into v_{i+1}, whence we have a new path $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ of the same length since we just divided by 1. Given a path of the collapsed vertices $w_1, \ldots, w_k$, we can unravel this path by remembering which vertices we collapsed into which, and lift this to a path $v_1, \ldots, v_n$ (where $n \geq k$ possibly), of the same length, just multiplied by a high power of 1.

(2) After collapsing edges, we know we will be left with just weight-2 edges, so we can replace the weight of each edge with the log of it's weight. This can clearly be done in $O(n + m)$ time, and ensures that the BFS level of each vertex is the log of the shortest path to it.

(3) Finally, we can run BFS on the graph starting at s, and return $2^{\text{BFSLevel}(t)}$ as the answer. Since BFS will return the shortest path, by our logic in previous two steps we can see that we will return the shortest path in the original graph with the original weights and weight function. Since BFS runs in $O(n + m)$ time, we see that our entire procedure runs in $O(n + m)$ time, and we are done.