

1.

First, note that UNION operations and MAKE operations each cost $O(1)$, so the only difficulty in this problem is bounding the total amount of time required for the FIND operations.

Let us look at a particular node v . After one FIND operation which traverses the parent pointer of v (that is, a FIND of either v itself or some descendant of v), the parent pointer of v is reset to point to the root. Since all UNION operations are done before all FIND operations, the root remains the same during all FIND operations. We account for the total time required by all FIND operations by assuming that each pointer access costs one dollar, and that every node has a dollar stashed in it when it is created, which the FIND operation can use when it accesses the node. The first time v is accessed by a FIND operation, we use its dollar to pay for traversing v 's parent pointer and resetting the parent pointer to point to the root. On all subsequent FIND operations which access v , v is a child of the root and we have to pay one dollar for the access "out of pocket". Thus every access to a node which is NOT a child of the root can be paid for with the dollars stashed at the nodes themselves, and each FIND operation only has to pay "out of pocket" for the accesses to the one node traversed which is a child of the root.

Let n be the number of nodes, and let m be the number of operations of all sorts (MAKE, UNION, FIND) performed on the data structure. The total cost (and hence the time required for all pointer operations) during the FIND operations is at most $O(n)$ for the n dollars stored at the nodes, and $O(m)$ for the pointer operations on nodes which are children of the root. Since $n \leq m$, this is $O(m)$.

2.

Claim: If the edge weights in graph G are unique, the minimum spanning tree is unique.

Proof: If the edge weights are unique, this means that no two edges have the same weight. Assume for the purpose of contradiction that there are two minimum spanning trees of equal weight, T_1 and T_2 . Since T_1 differs from T_2 , there must be some minimum-weight edge in one which is not in the other; assume without loss of generality that this minimum-weight edge e is in T_1 but not in T_2 . Consider adding e to T_2 . It forms a cycle in T_2 . Since all edges in either tree with weight less than the weight $w(e)$ of e are in both T_1 and T_2 , it cannot be the case that the cycle containing e in $T_2 + e$ consists of only edges with weight less than $w(e)$, since then the cycle would exist in T_1 as well. So we can remove from the cycle an edge of weight greater than $w(e)$, contradicting the assumption that T_2 was also a minimum spanning tree.

3.

Recall that that a Hamiltonian cycle is a cycle that visits each vertex exactly once; what we have been calling a tour. The Hamiltonian cycle problem takes an undirected, unweighted graph as input and answers YES if the graph contains a Hamiltonian cycle, and NO otherwise. The very similar Hamiltonian path problem takes an undirected, unweighted graph as input and returns YES if it contains a path which visits each vertex exactly once, and NO otherwise.

To show that Hamiltonian cycle is NP-complete, we need to prove 1) that it is NP-hard and 2) that it is poly-time checkable.

To show that Hamiltonian cycle is NP-hard, we argue that if we could solve Hamiltonian cycle in polynomial time we could solve the known NP-complete problem Hamiltonian path in polynomial time. So assume that we can solve Hamiltonian cycle in polynomial time. Then we could use the following algorithm for Hamiltonian path. It takes a graph $G = (V, E)$ as input.

```

HamiltonianPath( $G$ )
 $n = |V|$ 
For  $i = 1$  to  $n$ 
    For  $j = 1$  to  $i$ 
        if  $(i, j) \notin E$ 
             $G' = G + (i, j)$ 

```

```

        if HamiltonianCycle( $G'$ ) = YES return(YES)
return(NO)

```

The algorithm tries adding each edge (i, j) to the graph G in turn. If the new edge creates a Hamiltonian cycle, then without the edge, the remainder of the cycle forms a Hamiltonian path. If G contains a Hamiltonian path, then when i and j are the two endpoints of the path, adding edge (i, j) will produce a Hamiltonian cycle and it will be found. If no edge (i, j) produces a Hamiltonian cycle, we know that G contains no Hamiltonian path. Thus, the algorithm above outputs YES when G contains a Hamiltonian path and NO otherwise. It runs in polynomial time if HamiltonianCycle runs in polynomial time, since it calls HamiltonianCycle at most n^2 times. This establishes that Hamiltonian Cycle is NP-hard.

To show that Hamiltonian Cycle is polynomial-time checkable, we have to argue that there is some "proof" which we can check in polynomial time that G does indeed contain a Hamiltonian Cycle. The obvious "proof" one could provide is the description of a Hamiltonian cycle in G , for instance as a list of vertices. In polynomial time we could check that each vertex in the list is connected to its neighbor by an edge, and that the last and first vertices in the list are connected together by an edge.

4

The easiest reduction is from the known NP-complete problem Clique. Recall that the input to Clique is an undirected, unweighted graph $G = (V, E)$ and an integer k , and we should output YES if there is a set $C \subseteq V$ of vertices such that every vertex in C is connected to every other, and with $|C| \geq k$. Such a set C is known as a clique.

We want to show that if we can solve Independent Set in polynomial time, then we could also solve Clique in polynomial time. We could use the following algorithm. Given G , construct the *complement graph* G' . The vertex set V' of G' is the same as the vertex set V of G . For each vertex pair (u, v) , if $(u, v) \in E$ in G , then $(u, v) \notin E'$ in G' , and if $(u, v) \notin E$ in G , then $(u, v) \in E'$ in G' . That is, edges in G are NOT edges in G' , and visa versa.

Now notice that if G contains a clique C , then no vertex in C is connected to any other vertex in C in G' . So C is an independent set in G' . Thus if G contains a clique of size $\geq k$, G' contains an independent set of size $\geq k$, and if G does not contain a clique of size $\geq k$, then G' does not contain an independent set of size $\geq k$. So our algorithm is just to construct G' , and then output YES if and only if IndependentSet(G') outputs YES. This proves that Independent Set is NP-hard.

We also need to prove that Independent Set is polynomial-time checkable by describing a "proof" which we can check in polynomial time. If we are given an independent set C of size $\geq k$, we can check in polynomial time that no vertex in C is attached to any other.