# 15295 Spring 2018 #4 Disjoint Sets -- Problem Discussion

February 7, 2018

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

## A. Destroying Array

The key here is to work backwards. Rather than destroying each element, we reverse the ordering array and simulate creating each element of the array. We use a union find structure to track which elements are part of the same segment, augmented with the sum of the values in that segment.

Each time we create an element, we check if its neighboring elements have already been created. If so, we union them together, and update the sum of their segment. We also track the running maximum value across all the segments, which we can update in O(1) time. At each step, we record this maximum, then reverse it at the very end as the result.

--Tom W

## B. Minimum spanning tree for each edge

Associate to each vertex a binary tree with a single node representing the vertex. During Kruskal's algorithm, when unioning two components A and B, associate to the new component a binary tree whose root represents the edge joining A and B and whose children are the trees associated with A and B. Once completed, we have the MST and a binary tree whose leaves represent the vertices of the graph, and the heaviest edge in the MST between any pair of vertices is represented by the LCA of the leaves.

For each edge in the MST, the size of the MST is the answer. For every other edge uv with weight w, find the heaviest edge in the MST between u and v, of weight w'; the MST containing uv is obtained by replacing this heaviest edge with uv, so the answer is (size of MST) - w' + w.

For finding the LCA, because we know all the necessary queries after obtaining the MST, Tarjan's offline LCA algorithm, which is also based on union-find, can be used. (The basic idea is to traverse the tree depth-first and keep track of the ancestors of visited nodes using union-find, checking at each step whether a query can be answered.)

--bney

Thanks bney. I must point out an easier solution, which contains the building blocks for the solutions to E. Envy described below.

We run Kruskal's algorithm on the graph. We use union by rank, and no path compression. Also, when we link two roots together by the union operation, the root node that got linked to a parent remembers the weight of the edge used to do that link.

Take a snapshot of the state of the world at some point in time. There is a current set of trees that have been built by the edges selected by Kruskal's algorithm. Each of these trees T has a counterpart tree T' in the union/find data structure that has the same nodes and same edges (weights) but different structure. These union/find trees have two important properties:

(1) The length of a path from a node in T' to the root is at most log n.
(2) Given two nodes (a,b) in T, let w(a,b) the weight of the most expensive edge on the path from a to b in T. Let w'(a,b) be the same except using T'. Then we have that w(a,b) = w'(a,b)

These properties can be proven by induction. Basically what is happening is this. Say Kruskal's is linking two trees T1 and T2 together by adding an edge e of weight w. Say a is in T1 and b is in T2. Then the most expensive edge on the path from a to b is w. Note that when the union/find algorithm links T1' and T2' together the edge used to link them is the most expensive edge in T1 and T2. Also, the path from a to b in the union/find tree MUST GO THROUGH that most expensive edge. So although the path from a to b in the union/find tree is much shorter than the path in the MST, that path must contain the most expensive edge on the path.

Now we have do queries of the form: what's the cost of the MST containing edge (i,j)? So at the end Kruskal's has produced one tree T, and we have the corresponding union/find tree T'. If edge (i,j) is in T, then our answer is the cost of the MST. If (i,j) is not in T then we can introduce (i,j) and remove the most costly edge on the cycle in T that it creates. That gives the MST containing (i,j).

And we can find that most costly replacement edge to (i,j) by using T'. We just find the least common ancestor of i and j, and take the maximum cost edge on the path between them. Since T' has depth at most log n, we can do this easily in O(log n) just by walking up the two paths (from i and from j) and figuring out the deepest point in the tree where they meet. As we've shown, that path contains the most expensive edge.

--Danny

**C. Swaps in Permutation**

First we determine which indices can be swapped into which other positions. We use a union find structure (with n sets) and union two indices if there is a swap between them. Then we sort digits of the permutation within each set. One way to to this is to create a heap for each set and insert all the digits at indices within that set into the heap. To create the final output, we loop over the indices, find which set it belongs to, and pop from the corresponding heap (if using a max-heap).

--Tom W

**D. Table Compression**

**E. Envy**

First, perform Kruskal's as normal, doing union by rank, but not path compression, and store the edge weight of the joined edge at the top of the tree (like in problem B).

Now, we need to figure out a way to answer the queries. To do this, we will simulate what would have happened if we had performed Kruskal's with the edges we want to use sorted first. If we can figure out a way to detach an old edge of the same weight as the new edge we are adding while keeping the MST acyclic, we are done (note that here we cite a theorem that the sorted list of edge weights in any MST is constant for a given graph).

Modify your find operation to not traverse edges of weight greater than the current weight of the original spanning tree (traverse all edges of the new ST). Then, if the find from each node that will be connected finds different nodes, we can detach the node of lower rank from its parent (provided the parent-edge has the same weight as the edge we are adding), and attach it to the node of higher rank. This simulates what would have happened had we actually run Kruskal's with that edge. If at every point, we can remove an edge of equal weight and add in our desired edge, we have successfully transformed the spanning tree. If both endpoints of the edge end up at the same node, then this edge would have produced a cycle during Kruskal's and the set of edges cannot be part of the minimum spanning tree.

One trick for implementing this is to use round numbering in your nodes. Then, if the node's round is equal to the current round, traverse the virtual edge. Otherwise, traverse the real edge if the weight is not larger. Then, when re-attaching a node, set its round to the current round.

--Corwin

The first half of my solution is the same as Corwin's. But mine does not actually change the tree at all when processing a query. Let me start from the beginning and explain it.

Think about what happens when you run Kruskal's algorithm. So the edges are sorted by weight. Consider a contiguous, maximal, subsequence of edges in the sorted list that have the same weight. This set of edges are is S = {e(i), e(i+1), … e(j)}. Let their weights be w(i)...w(j) respectively. We have:

$$w(i-1) < w(i) = w(i+1) = w(i+2)= \dots = w(j) < w(j+1)$$

After processing e(i-1) there exists a collection C of disjoint sets of vertices produced so far by the algorithm. Consider a subset T of S. We'd like to know if all the edges of T can be in an MST.

Claim: Let G(C,T) be a graph where the vertices are the disjoint sets of C, and the edges are those induced by T on those sets of C. The edges of T can be used in an MST iff the graph G(C,T) is acyclic.

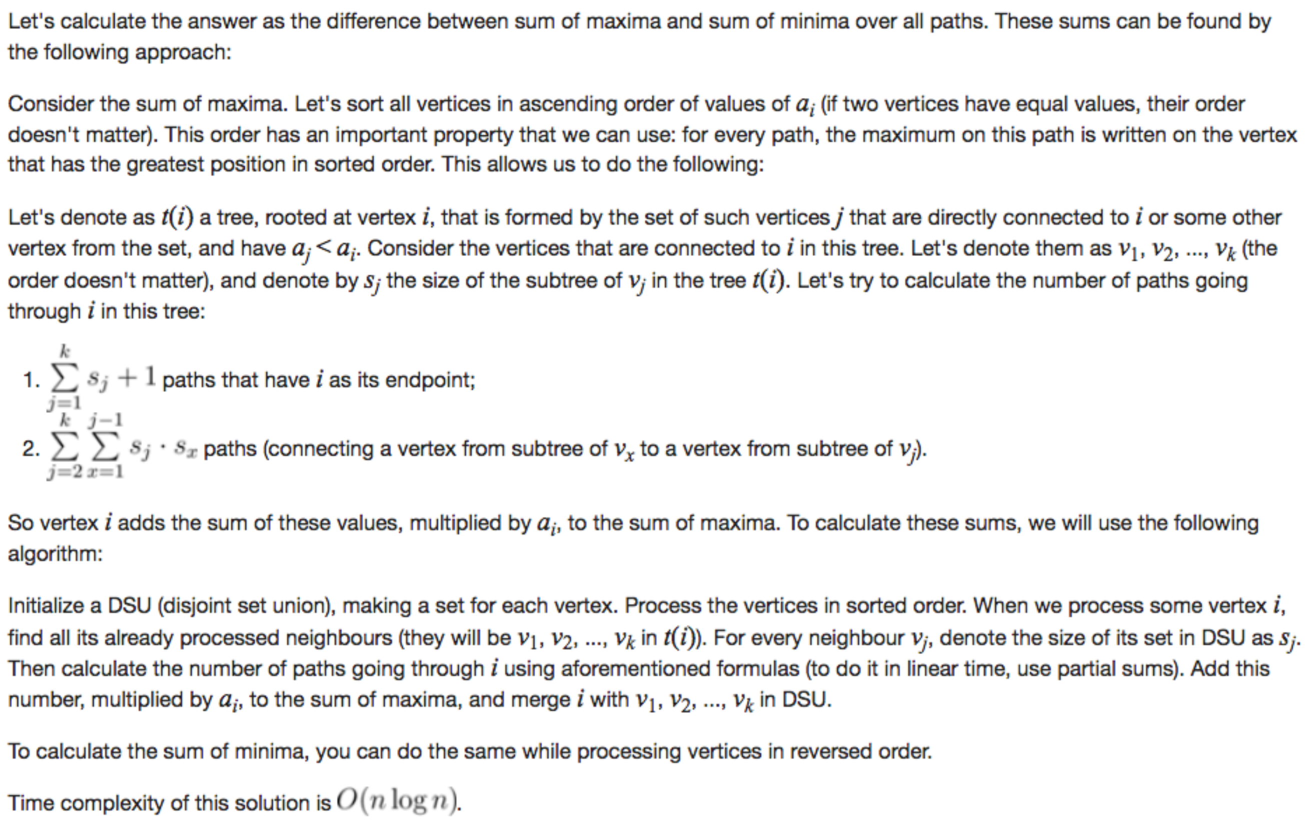
The proof is simply that if we reorder the edges of S with those of T being first, then Kruskal's algorithm will use all the edges of T iff that graph G(C,T) is acyclic.

So here’s my solution to this problem. Run Kruskal’s algorithm using union by rank and no path compression (see problem B). Each node of the tree keeps the weight of the edge that caused that union to occur. (Same as Corwin’s)

We process a query as follows. Group the edges in the query in bundles of those with the same weight. Let T be a set of equals in the query. We have a modified find that only traverses edges with weight less than the edges of T. Using this find we find the canonical element (elements of the collection C) of all the endpoints of the edges of T. We build the graph G(C,T) described above and see if it’s acyclic. If all groups of edges in the query satisfy this test then the subset can be used, otherwise not.

--Danny

## F. Imbalance Value of a Tree



Let's calculate the answer as the difference between sum of maxima and sum of minima over all paths. These sums can be found by the following approach:

Consider the sum of maxima. Let's sort all vertices in ascending order of values of $a_i$ (if two vertices have equal values, their order doesn't matter). This order has an important property that we can use: for every path, the maximum on this path is written on the vertex that has the greatest position in sorted order. This allows us to do the following:

Let's denote as $t(i)$ a tree, rooted at vertex $i$, that is formed by the set of such vertices $j$ that are directly connected to $i$ or some other vertex from the set, and have $a_j < a_i$. Consider the vertices that are connected to $i$ in this tree. Let's denote them as $v_1$, $v_2$, ..., $v_k$ (the order doesn't matter), and denote by $s_j$ the size of the subtree of $v_j$ in the tree $t(i)$. Let's try to calculate the number of paths going through $i$ in this tree:

1. $\sum_{j=1}^{k} s_j + 1$ paths that have $i$ as its endpoint;
2. $\sum_{j=2}^{k} \sum_{x=1}^{j-1} s_j \cdot s_x$ paths (connecting a vertex from subtree of $v_x$ to a vertex from subtree of $v_j$).

So vertex $i$ adds the sum of these values, multiplied by $a_i$, to the sum of maxima. To calculate these sums, we will use the following algorithm:

Initialize a DSU (disjoint set union), making a set for each vertex. Process the vertices in sorted order. When we process some vertex $i$, find all its already processed neighbours (they will be $v_1$, $v_2$, ..., $v_k$ in $t(i)$). For every neighbour $v_j$, denote the size of its set in DSU as $s_j$. Then calculate the number of paths going through $i$ using aforementioned formulas (to do it in linear time, use partial sums). Add this number, multiplied by $a_i$, to the sum of maxima, and merge $i$ with $v_1$, $v_2$, ..., $v_k$ in DSU.

To calculate the sum of minima, you can do the same while processing vertices in reversed order.

Time complexity of this solution is $O(n \log n)$.