Colors – solution

Author: Costin Oncescu

We begin with some general observations about the problem.

Proposition 1. *The values of a never increase.*

Proof. The operation $a_u = \min(a_u, a_v)$ can only make a_u smaller, never larger. □

Proposition 2. *If initially $a_u < b_u$ for any node u, then there is no solution.*

Proof. This follows directly from Proposition 1. □

Proposition 3. *Any constructive algorithm may never make $a_u < b_u$.*

Proof. If it any point we operate on a_u and make it smaller than b_u, then we find ourselves in the conditions of Proposition 2. Obviously this has no importance if there is no solution to begin with, but we cannot know that beforehand. □

Proposition 4. *When propagating color c from node u to node v, we may only pass through nodes w having $a_w \geq c$ and $b_w \leq c$.*

Proof. If $a_w < c$, then we simply cannot assign color c to node w because the min operation would select a_w, not c. If $b_w > c$, then by assigning color c to node w we would be violating Proposition 3. □

This is the crux of the solution: propagating a color c from nodes u that have it ($a_u = c$) to nodes v that need it ($b_v = c$).

Definition 1. *A node v can be satisfied if there exists a node u with $a_u = b_v$ and a path $u \to v$ such that all nodes w on the path have $a_w \geq b_v$ and $b_w \leq b_v$. Node u is said to be a source node for v.*

Note that nodes v having $a_v = b_v$ are trivially satisfied. The path contains only node v itself and no operations are necessary.

Definition 2. *A color c can be satisfied if every node v having $b_v = c$ can be satisfied.*

Proposition 5. *Coloring a can be changed into b if and only if every node can be satisfied.*

Proof. The negative half is easy: If there exists a node v that cannot be satisfied, then either (a) we will not be able to change a_v into b_v or (b) we can only change it by making $a_w = b_v < b_w$ somewhere along the way, thus violating Proposition 3.

To prove the positive, constructive half, we remark that propagating colors changes the graph. By making the value of a_w smaller for an arbitrary node w while satisfying a node v, we are making it harder to obey the condition $a_w \geq b_{v'}$ later when we are attempting to satisfy another node v'.

Fortunately, the fix is simple. We consider colors in decreasing order, from the largest value in b to the smallest. Suppose at some moment we are propagating the value c. This may affect some nodes w having $a_w > c$ by making $a_w = c$ ("the change"). However, this will not be a problem when propagating a future color $d < c$. There are three possible cases:

1. Node w was accessible before the change, meaning $a_w \geq d$ and $b_w \leq d$. After the change, $a_w = c > d$ and b_w is unchanged, so node w is still accessible after the change.
2. Node w was inaccessible before the change because $a_w < d$. Since the change further decreased a_w to c, node w is still inaccessible after the change.
3. Node w was inaccessible before the change because $b_w > d$. Since the change did not alter b_w, only a_w, node w is still inaccessible after the change.

□

Next we discuss how to implement this for the various graph types given in the statement.

Complete graph

In a complete graph the path $u \to v$ is simply the edge (u, v). It is never necessary to visit intermediate nodes because changing colors can only make the problem harder, never easier.

Thus node v can be satisfied if there exists a node u with $a_u = b_v$. Globally, the problem admits a solution if:

1. $a_u \geq b_u$ for all u;

2. $b \in a$ (we can view a and b as sets by considering their distinct elements).

The time complexity is $O(N^2)$ because we still need to read past the edges of the graph in order to get to the next test case. Deciding the satisfiability itself takes $O(N)$ time.

Chain (1-dimensional array)

When all the nodes lie on a chain, we will view the graph as a pair of arrays a and b and paths as ranges in those sequences. We can satisfy an index i if

(a) There exists an index $l \leq i$ such that $a_k \geq a_i$ and $b_k \leq b_i$ for all $l \leq k < i$ (informally, we propagate the color from the left), **or**

(b) There exists an index $r \geq i$ such that $a_k \geq a_i$ and $b_k \leq b_i$ for all $r \geq k > i$ (informally, we propagate the color from the right).

We explain how to handle the left side. One approach that is easy to formulate uses range minimum/maximum queries. We store pointers from each i to the closest $l \leq i$ having $a_l = b_i$. Then we can satisfy index i from the left if

1. $\min(a_l, a_{l+1}, \ldots, a_i) \geq b_i$ (or we simply won't be able to propagate color b_i) and
2. $\max(b_l, b_{l+1}, \ldots, b_i) \leq b_i$ (or propagating color b_i will make some indices unsatisfiable).

The running time is $O(N \log N)$ with a practical implementation of range minimum queries. This can be improved to $O(N)$ using sorted stacks.

Star graph

As before, we assume that $a_u \geq b_u$ for all nodes and that all values in b also appear in a. Then the root r is satisfiable because we can propagate b_r along a direct edge if needed. Furthermore, there are only three ways to satisfy a leaf v.

1. If $b_v = a_v$ then nothing needs to be done.
2. If $b_v = a_r$ then we propagate b_v from the root.
3. If $b_v = a_u$ for some v then the path $u \to v$ passes through r and v is satisfiable if $a_r \geq b_v$ and $b_r \leq b_v$.

In theory, case (3) can mean that a_r and b_r must have the maximum and minimum values in b. Checking this condition explicitly is not necessary and can be tricky in practice. For example, the nodes having the minimum value in b may already be satisfied (case 1 above).

Small tree

Trees have $M = N - 1$ edges. When the sum of N^2 is small, an $O(MN)$ approach works. Please see the section "Small graph" below.

Permutation tree

If b is a permutation of a, then for every node v there exists exactly one possible source node u and a single path $u \to v$. For u to be a source node, we must check that:

1. $\min\limits_{w \in u \to v} a_w \geq b_v$ and
2. $\max\limits_{w \in u \to v} b_w \leq b_v$.

Thus, the solution reduces to path minimum and maximum queries. We discuss the minimum case. One approach is to choose an arbitrary root r and define

- $A(u, k)$ as the 2^k-th closest ancestor of u for $k \geq 0$;
- $B(u, k)$ as the minimum value of a over the closest 2^k ancestors of u, including u itself.

Since $k \leq \log N$, we need $O(N \log N)$ space to store A and B. We can also compute them in $O(N \log N)$, specifically

- $A(u, k + 1) = A(A(u, k), k)$
- $B(u, k + 1) = \min(B(u, k), B(A(u, k), k))$

We can then compute the lowest common ancestor l for every pair (u, v) and compute the path minimum by considering the paths (u, l) and (v, l). In turn, the answer for each path can be computed by considering two overlapping chains whose size is a power of 2 and which cover the path completely.

The time and space complexity is $O(N \log N)$.

Small graph

For small graphs, an $O(MN)$ approach is sufficient. Therefore, we can afford to run up to N depth-first searches, one from each node v. Each search runs in $O(M + N)$ and visits only nodes w having $a_w \geq b_v$ and $b_w \leq b_v$. Node v is satisfiable if and only if the search encounters any nodes with $a_w = b_v$.

General graph

When $M \gg N$, we reconsider the problem in terms of dynamic connectivity. Let $G = (V, E)$ be the initial graph. Let $c \in b$ be a color. Let $G_c = (V_c, E_c)$ be the graph induced by the set of valid nodes while trying to satisfy color c. Specifically,

- $V_c = \{u \in V \mid a_u \geq c \text{ and } b_u \leq c\}$
- $E_c = \{(u, v) \in E \mid u, v \in V_c\}$

Suppose we construct a disjoint-set forest for G_c. Then color c is satisfiable if for every node v having $b_v = c$ there exists a node u having $a_u = c$ in the same connected component as c.

Now let us consider the next color in decreasing order, $d < c$. In similar fashion we wish to obtain $G_d = (V_d, E_d)$, build its disjoint-set forest and decide the satisfiability of d. How can we achieve this? Simply rebuilding the forest from scratch takes $O(M\alpha(N))$, yielding a slow running time of $O(MN\alpha(N))$ for all the colors.

To improve upon this, let us consider what changes between V_c and V_d:

- Nodes having $a_u = d$ are added to the graph.
- Nodes having $b_u = c$ are removed from the graph.

Interestingly, each node (along with its incident edges) is added and removed from the graph exactly once. The key is to build the forest of d from the forest of c, or some other forest we have previously built, to save time. Thus, we have reduced the problem to *offline dynamic connectivity*, where we maintain a forest throughout the entire algorithm and perform M edge additions and M edge removals on it. This can be done theoretically in $O(\log N)$ per operation, but the implementation is impractical here. We present two different approaches, achieving $O(\log^2 N)$ and $O(\alpha(N)\sqrt{M})$ per operation respectively.

Disjoint-set forests with undo support

Consider an edge (u, v) with its initial values a_u, a_v, b_u, b_v. Suppose that, at some point during the algorithm, we propagate a value c across the edge. What can we say about c?

First, $c \leq a_u$ and $c \leq a_v$ because we started with a_u and a_v and values never increase. Second, $c \geq b_u$ and $c \geq b_v$, otherwise we would violate Proposition 3. Thus, we can introduce two notations t_1 and t_2 and say that

$$t_1 \triangleq \max(b_u, b_v) \leq c \leq \min(a_u, a_v) \triangleq t_2$$

The letter t is not accidental. We can think of colors as moments of time and say that edge (u, v) is "in existence" between times t_1 and t_2 inclusively. We do this for all edges. Now, in order to satisfy a color c, we wish to address the question: what edges are in existence at time c? Then we move to the next color, update the edge list and its corresponding disjoint-set forest, and repeat the question.

For this purpose, we construct a segment tree over the N time moments with all the intervals $[t_1, t_2]$. For every interval we also store the originating edge (u, v). Then we traverse the tree in depth-first order. When entering a node, we add all the edges stored in that node to the disjoint-set forest. We use stacks to keep the history of the forest data, specifically each node's rank and parent. This allows us, when exiting a node, to remove the edges from the disjoint-set forest and revert to the state before entering the node.

Finally, leaves in the segment tree correspond to single moments of time t, and at those leaves we query the disjoint-set forest to decide if the color t is satisfiable.

The use of stacks makes it impractical to use path compression in our disjoint-set forest. We still perform unions by rank, which achieves $O(\log N)$ time per operation.

Thus, there are M edges in the segment tree, each potentially occurring in $O(\log N)$ nodes, and to process each occurrence we perform $O(\log N)$ operation on the disjoint-set forest. The overall complexity is $O(M \log^2 N)$.

Square root decomposition

Suppose we intend to build from scratch the disjoint-set forest for a color c. However, if the number of nodes having $a_u = c$ or $b_u = c$ is small, then we have expended $O(M\alpha(N))$ effort for little benefit.

Instead, let us build a smaller forest, but one that we can keep using for a longer time. Specifically, find a color $d \leq c$ such that there are $O(\sqrt{M})$ edges appearing and disappearing in all the transitions from E_c to E_d. We call the interval $[d, c]$ a block. Next, build a disjoint-set forest F using all the edges in $E_c \cap \cdots \cap E_d$, namely edges between nodes u having $a_u \geq c$ and $b_u \leq d$. F is relevant to all the satisfiability checks for colors d through c. Make a copy of F so we can reuse it multiple times.

In order to answer the satisfiability question for a color $e \in [d, c]$, we augment F with all the relevant edges, specifically all those between nodes u having $a_u \geq e$ and $b_u \leq e$. Due to our choice of d, there are $O(\sqrt{M})$ edges to add and $O(\sqrt{M})$ nodes in whose connectivity we are interested, so each color can be verified in $O(\alpha(N)\sqrt{M})$ time.

Once we are done, discard F and move on to the next block, beginning with the next color after d.

The running time is made up of:

1. Block-level effort. There are $O(\sqrt{M})$ blocks and it takes $O(M\alpha(N))$ to rebuild the forest in each block.
2. Color-level effort. There are N colors and for each color we check connectivity in $O(\alpha(N)\sqrt{M})$.

Thus, the overall running time is $O(\alpha(N)M\sqrt{M})$.

This approach is not theoretically sound, because there may exist a color (even multiple colors) with $O(M)$ incident edges. However, it behaves well in practice.