EGOI 2024 Editorial - Make Them Meet

Problem author: Hazem Issa.

The problem

You are given a graph with N vertices and M edges. There are two tokens in some of its vertices, but you do not know in which ones. You can apply multiple operations. In one operation, you assign an integer between 0 to N to each vertex – its color. Then each token is moved to an adjacent vertex with the same color as its current vertex. If there is no such adjacent vertex, the token stays in the same vertex. If there are multiple such adjacent vertices, the token could go to any of them. Your goal is to make sure the two tokens *meet* at least once: they either are in the same vertex at some point, or pass the same edge in the opposite directions at some point. Note that they do not have to be in the same vertex after all operations are done. Your goal is to make them meet in as few operations as possible, more specifically you get a full score if you use at most 600 operations, while $N \leq 100$.

Overall approach

How to interpret "if there are multiple such adjacent vertices, the token could go to any of them"? This is not an interactive problem, you simply print a sequence of operations, and then the checker will verify that your output is correct, meaning that the two tokens meet no matter where they start and which choices they make when there are multiple options.

This means that the checker is probably implemented in the following way: it maintains a set of pairs of positions (a, b) where the two tokens could currently be if they have not yet met, and updates this set after every operation. In the end this set needs to be empty. This concept will also be helpful for designing our own solution below.

Test group 1: a star

In this test group the graph is a star where everything is connected to vertex 0 and there are no other edges.

It turns out that coloring everything with the same color is a really powerful operation for a star: when we do it, if the tokens were in two leaves of the star, they will all gather to the center and meet. The token from the center on the other hand will go to an arbitrary leaf. So after this operation the set of possible pairs will be $\{(0, i) : i \geq 1\}$.

Repeating the same operation again does not yield anything, as we will still have $\{(0, i) : i \geq 1\}$ after it. However, we can do the following operation instead: color vertices 0 and 1 with one color, and all other vertices with their own colors. Then we get the following state: $\{(1, i) : i \geq 2\}$.

Now the two tokens are guaranteed to be in the leaves, so we can color everything with the same color again and we will make them meet. So for a star just 3 operations are enough.

These operations will also be useful building blocks for the subsequent test groups.

Test group 3: a path

In this test group the graph is a path where vertices i and $i + 1$ are connected. Note that we discuss this test group before test group 2.

Consider the following two operations. Operation *even* colors vertices 0 and 1 with the same color, vertices 2 and 3 with same color (but different from 0 and 1), vertices 4 and 5 with same color, etc. Operation *odd* does almost the same, but coloring vertex 0 with its own color, and then vertices 1 and 2 with the same color, vertices 3 and 4 with the same color (but different from 1 and 2), etc.

We will now alternate those operations: even, then odd, then even again, and so on.

Since every color is used at most twice in those operations, there is never uncertainty as to where a token moves. Let us follow what happens to a token that starts in vertex 0. On the first even operation it moves to vertex 1, then on the second odd operation it moves to vertex 2, then on the third even operation it moves to vertex 3, and so on, until it reaches vertex $N-1$, at which point it will stay still for one operation, and then start moving backward to vertex $N-2$, then $N-3$, and so on until it reaches vertex 0 again, where it will stay still for one operation, and then start moving forward again.

A token that starts in any other vertex will follow the same path: first move forward or backward towards one of the ends of the path, then move in the other direction to the other end of the path, and so on.

Note that after exactly N steps a token in vertex a always moves to vertex $N - 1 - a$. Therefore tokens in two different vertices a and b will swap their ordering on the path, and therefore will have to meet. So this approach solves this test group in N operations.

This pattern will also be very useful to solve the general case.

Test group 2: a clique

In this test group any two vertices are connected with an edge.

It turns out that we can apply the solution from test group 3 without any changes (maybe some of you got the points for this test group without expecting it?). Since the clique also contains the path, and the solution for the path uses the same color only for pairs of vertices connected by a path edge, the tokens would move in exactly the same way.

There are many other approaches that work for a clique, but reusing the solution for the path can save you some time.

Test group 4: a tree

In this test group the graph is a tree.

Both a star and a path are special cases of a tree. So we likely need to come up with a solution that reduces to the above solution for a star when the input is a star or a tree which has vertices of high degree, and to the above solution for a path when the input is a path or similar to it.

Here is a way to achieve this. Consider any leaf of the tree, and root the tree using it. Let a be this leaf, and b be its only neighbor. Let us number all levels of the tree (distance from root) starting from 0, so that vertex a has level 0 and vertex b has level 1. Then consider the following version of even and odd operations:

Operation *odd* colors all vertices on odd levels using the same color as their parent, while for vertices on even levels we pick distinct colors. Operation *even* colors all vertices on even levels using the same color as their parent, while for vertices on odd levels we pick distinct colors. For the root of the tree a (level 0), there is no parent so we need to define its color separately: operation even colors it the same color as b. So the edge between a and b is activated in both operations.

We now alternate these operations starting with operation odd, then operation even, then operation odd again, and so on.

What happens to a token that starts somewhere in this tree? When it starts on an even level and not at a, operation odd will move it down one level, then operation even will move it down one more level, and so on until we reach a leaf, at which point the next operation will do nothing, and then it will start going up one level every step until it reaches a after operation odd, then it will go back to b after operation even, then it will go back to a after operation odd, so it becomes trapped going back and forth on the edge between a and b after less than $2N$ steps.

When a token starts on an even level, operation odd will move it up one level, then operation even will move it up further, so it will get trapped on the edge between a and b even faster.

The most tricky case is a token that starts at a. On the first operation odd, it will go down to b. On the second operation even, it has two choices: it can either go down, in which case it follows the same path as any other even-level-starting token and eventually gets trapped on the edge between a and b, or go back up to a.

So in the general case, a token starting at a can go between a and b some (even) number of times, and then go down to the rest of the tree, then go back up, and be stuck between a and b forever after.

Since any other token will reach the edge between a and b after less than $2N$ steps, if the token starting at a still did not leave this edge by then, they will meet. And if it leaves this edge before $2N$ steps complete, it will be back there in less than $4N$ steps and the tokens will meet then. So the tokens are guaranteed to meet in less than $4N$ steps here.

In fact, a more accurate analysis can prove that they will meet in less than $2N$ steps, because if the token starting at a does not meet the other token on its way down the tree, it means that the additional time it takes this token to return to the root is taken using a path disjoint from the path of the other token, and therefore since the total number of edges in two disjoint paths is at most $N - 1$, the token starting at a will return to the root in at most $2N$ steps if it does not meet the other token on the way down. But even $4N$ was enough for the constraints of the problem.

Test group 5: a general graph

In this test group the graph is an arbitrary connected graph.

The first idea is, can we use the same trick we used for the path and the clique, just find any spanning tree in the graph and use the tree solution on it? Unfortunately, things are not so simple. Since our solution for the tree sometimes colored two vertices with the same color that are not connected by an edge, if the graph has an edge between such vertices, then additional movements will be possible (within a level, or between level 0 and level 2), which will break the predictable movement pattern.

So our goal should be to find a spanning tree that is rooted using one of its leaves, and such that there are no additional edges between the children of one parent, and no edges from the root to level 2. The first property is easy to satisfy, for example a depth-first search tree will never contain horizontal edges, but it is harder to make sure there are no edges from the root to level 2. In fact, this is clearly impossible when the graph is a clique, but for that case we have already described a solution above.

Let us run the depth-first search on the graph from any vertex. If the resulting depth-first search tree is a path, we will use the path solution above, as we did for the clique. Otherwise the depth-first search tree has at least one branching point, let us find such branching point that there are no branching points below it, in other words that all its children are paths. Let us denote this branching point as q, its parent as p, and its children as r_i. By the depth-first search tree properties, there are no edges between r_i.

First, suppose there is at least one r_i that does not have an edge to p, without loss of generality we can assume it is r_0. Let us temporarily remove the rest of the path beyond r_0 from the graph (so r_0 becomes a leaf in the depth-first search tree attached to q) and run a new depth-first search starting from r_0, ordering the edges in such a way that the depth-first search first goes to q, and from q it first traverses other r_i and p. This way there will be no new children of q in this tree, and therefore the condition above will be satisfied: the root r_0 is not connected to level 2.

Note that we have removed the rest of the path beyond r_0 temporarily, let us add it back while maintaining the properties of our solution. It will form an additional path starting from the root, let us label it with the same levels (r_0 has level 0, the next vertex on the path has level 1, and so on). We update operation even to also color the vertices of level 1 and 2 on the path with the same color, vertices of level 3 and 4 on the path with the same color, and so on. We update operation odd to also color the vertices of level 2 and 3 on the path with the same color, vertices of level 4 and 5 with the same color, and so on. We would also like to cover the vertices of level 0 and 1 of the path with the same color, but that would interfere with the existing tree construction. Therefore we introduce a new operation *bridge* that has just two vertices colored with the same color: vertices of level 0 and 1 on the path.

We now repeat the three operations in sequence: odd, even, bridge, odd, even, bridge, . . . A token in a vertex that is not on the additional path behaves as before: it either goes down, then back up, and then keeps going between r_0 and q forever, or goes directly up, and then keeps going between r_0 and q forever. A token that is on the additional path works similar to before, but not exactly the same: it keeps traversing the additional path back and forth, and every time it passes r_0, it can either continue going back and forth on the path, or go down the rest of the tree. If it does the latter, then it will eventually return from the tree and keep going between r_0 and q forever.

Two tokens will still meet after at most N iterations of our repeating sequence, but since the sequence repeats three operations instead of two as before, this is guaranteed to make them meet after at most $3N$ operations.

We have one case left to consider in the solution above: when all r_i have an edge to p. In this case we need to come up with one more small trick: we temporarily remove the path after r_0 as before, then start a depth-first search from r_0 as before, make it consider the edge to q first as before, and then make it consider the edges from q to r_i before other edges from q. Since all r_i are connected to p, this will traverse vertex p while traversing the children of r_i, and therefore the set of children of q in the new tree will be a subset of r_i only, which means that level 0 is once again not connected to level 2, and we can proceed with the rest of the above solution.

There are many other approaches that work in this problem. Some of them might be easier to come up with, but harder to implement or to prove.