NOI.PH Training: Graphs 5

Dynamic Connectivity

Robin Yu, Kevin Atienza

Contents

1 Dynamic Connectivity

The overall theme of this section is solving variations of the following problem:

Problem 1.1 (Dynamic Connectivity)**.** You have a graph with n vertices and e edges. You need to support q operations, each of one of the following types:

1. Given two vertices u and v, is there a path from u to v?
2. Given two vertices u and v, add an edge between u and v.
3. Given two vertices u and v, remove the edge between u and v (we guarantee that this edge exists).

There is a straightforward solution to this problem that runs in $\mathcal{O}(q(n+e))$, which is not really worth discussing. We are interested to know whether faster solutions exist.

We will also consider special cases of the problem, where only a subset of the operations are allowed. These special cases usually have much simpler solutions—faster, and/or easier to implement.

There are also more general versions, e.g., finding the size of a component, etc. You can think about which techniques below can be extended to handle these queries.

1.1 Incremental Connectivity

When only the first two types of operations exist, this problem is known as the **incremental connectivity** problem and it is straightforward; it can be solved using the union–find data structure. You probably already know about it, so I won't elaborate much. However, I'll mention some things about it that are helpful to know:

- The running time is $\mathcal{O}(n + (e + q)\alpha(n))$ with union by rank and path compression[1] or $\mathcal{O}(n + (e + q)\log n)$ if you only use one of them.
- You can augment the union-find data structure to include some data about every connected component (such as its size, sum, max, etc.)—simply store them as a separate array in your union-find class.
- Alternatively, you can store them in an external map whose keys are the representatives of each connected component, so you don't need to modify your implementation of a union-find class.

Exercise 1.1. Implement a union-find data structure in a more straightforward way: just as a list of lists!

Show how to implement it in such a way that the *amortized* running time of each operation is $\mathcal{O}(\log n)$.

Hint: You should know the proof that the smaller-to-larger trick works.

[1] where α is the inverse Ackermann function

Exercise 1.2. Show how to adjust the answer to Exercise 1.1 to handle the "find" operation in $\mathcal{O}(1)$ *worst case* time, using just an extra array.

Hint: The array stores the ID of the component

1.2 Decremental Connectivity

When only the first and third types of operations exist, this problem is known as the **decremental connectivity** problem.

If all the operations are given upfront, that is, the problem is *offline*, then this problem is also quite easy to solve.

Exercise 1.3. Assuming that you can get the list of all queries and updates upfront, reduce the decremental connectivity problem to the incremental connectivity problem.

Hint: removal is the inverse of an insertion

Hint: reverse things

The runtime of this will depend a bit on the implementation, but it should be around the same as the incremental connectivity solution.

If the queries are not given upfront, that is, for example, we need to answer a query before we can get the next one, then the solution is much more complicated. The general case is too complicated to discuss here that it's better to just solve the full dynamic connectivity problem—we'll talk about it in the next part. But there are relatively simple algorithms for special types of graphs, such as trees and planar graphs, which we'll about.

The case of trees is pretty standard:

Exercise 1.4. Solve the decremental connectivity problem in an acyclic graph (i.e., a forest) in $\langle\mathcal{O}(n), \mathcal{O}(\log n)\rangle$ worst-case time. (In other words, $\mathcal{O}(n \log n)$ precomputation and $\mathcal{O}(\log^2 n)$ worst-case query time.)

Bonus: Do it in $\langle\mathcal{O}(n), \mathcal{O}(\log n)\rangle$.

Hint: (Simplifies things a bit) You can assume it's initially a tree by adding extra edges, then just remove those edges at the start.

Hint: Use known techniques on trees.

1.2.1 On a planar graph

Let's discuss how to solve decremental connectivity on a planar graph.[2] We'll have to exploit its planarity somehow.

One remarkable property of planar graphs is that they have a "dual graph", another planar graph. The dual graph is obtained by looking at the regions of the plane created by the planar graph, sometimes called *faces*. The vertices of the dual graph are these regions/faces, and two regions are adjacent in the dual graph if they share a nontrivial boundary, i.e., they are separated by an edge of the original graph. Figure 1 illustrates a planar graph and its dual.

[2] Actually, we won't just assume that the graph is planar, but that we also know how to embed it on the plane. There are algorithms to test whether a graph is planar, and to embed planar graphs on the plane. You can read about those online.

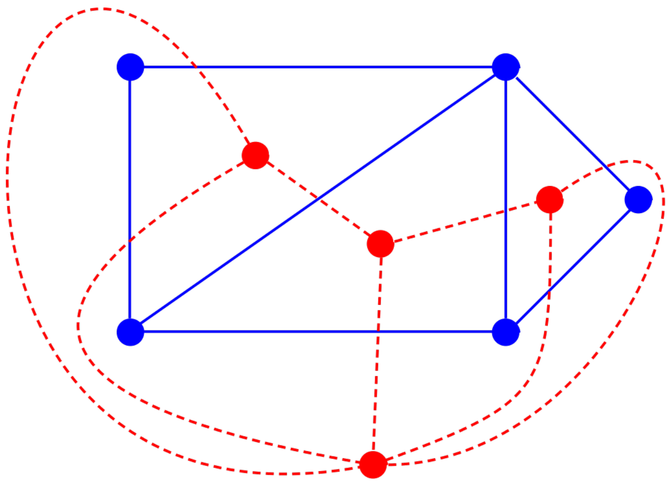


Figure 1: A planar graph and its dual. Image taken from Wikipedia.

It should be intuitively clear that the dual graph is also planar.

Now, let's think about what happens when we remove an edge. Notice that removing an edge in the graph joins the two regions it's separating. In other words, *removing an edge in the graph corresponds to adding an edge in the dual graph*! This seems useful, but at the moment, it's not clear how it can help us solve decremental connectivity.

Let's try solving an easier problem first. Can we even know when removing an edge splits a component into two? In other words, is it possible to know whether an edge is a *bridge* or not? On a static graph, it's possible with preprocessing, but we're dealing with the dynamic case, so this isn't possible.

So again, let's exploit planarity somehow. Suppose we remove a bridge. Then because of planarity, it means that we can now enclose both components with two continuous, nonintersecting curves. But this means that, before the removal, the two regions being "separated" by the bridge are already connected in the dual graph!

And in fact, the converse is true: if the two regions on both sides of an edge are connected in the dual graph, then it must be a bridge: we can draw the path connecting the two regions as a continuous curve in the plane, and it will enclose a group of nodes in one side of the edge (if you ignore the edge itself). This curve serves as a sort of *barrier*; it tells us that there must be no other way to exit the group of nodes its enclosing, except through the edge. Thus, the edge must be a bridge.

The above arguments are much easier to understand if you draw examples on paper. Try it out—drawing stuff is an important part of the problem-solving process.

This now gives us a way to determine if an edge we're about to remove is a bridge or not: it's a bridge iff the regions it separates in the dual graph are connected. But since removing an edge corresponds to adding an edge in the dual, it means that we can answer this query by solving *incremental* connectivity on the dual!

This certainly helps. For example, if we know that an edge is not a bridge, then we know that removing it will not affect the components of the graph, so we don't have to do anything else. However, if it's a bridge, then we have to update our structure somehow.

Luckily, there's a trick to do this: simply reverse our solution to Exercise 1.2! You can check that the amortized analysis still works.

In more detail, our "structure" will be pretty simple: we simply store an ID on each node identifying the connected component it contains. These IDs can be any integer, as long as they satisfy the property that two nodes are in the same component iff these IDs are the same.

Now, on a bridge removal, we're splitting a component into two, so we must somehow update the IDs of the affected nodes. However, we only have to update the IDs of one of the components, since the old ID can still serve its purpose.

Since we want the update to be as fast as possible, naturally we only want to update the *smaller* among the two components. But how do we know which component is smaller? Do we need to augment our structure again to keep track of size information?

No! What we can do is just *traverse both components simultaneously*, then stopping when we've finished visiting one of the components! In other words, once one traversal finishes (whichever one), we simply force stop the other one, since we already know the smaller component. We can then just update the IDs of the nodes in that component. Then the total amount of work is proportional to the size of the smaller component, which is exactly what we want!

How do we simultaneously perform two traversals? No, I don't mean threads, because that's a lot of headache.[3] Instead, we can just keep track of two stacks/queues, one for each traversal. Then for each stack/queue, we proceed with a single step of each DFS/BFS, and pause it as soon as it pushes onto the stack. Then we switch to the other one. Finally, we stop as soon as one of them becomes empty.

Exercise 1.5. Show that an update runs in $\mathcal{O}(\log n)$ amortized time.

Hint: Remember that we're doing the reverse of Exercise 1.2.

Exercise 1.6. What goes wrong if we pause the traversal whenever we *pop* from the stack instead?

Overall, our algorithm runs in $\langle\mathcal{O}(n), \mathcal{O}(\log n)\rangle$ amortized time.[4] Finally, I'll mention that this algorithm can be improved so that it runs in $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ amortized time. However, this algorithm is complicated. It involves using a trick reminiscent of the four Russians trick, combined with a complicated process called "r-division". Feel free to read about it if you like, but I don't recommend implementing it.

1.2.2 Parallel search in Python (Optional)

There's a fun way to implement this parallel traversal thing using the ability of Python's **yield** to yield control back to the caller. First, we implement a regular BFS, but we **yield** control after every time we append to the queue:

```python
def bfs(s, seq):
    seq.append(s)
    yield
    vis = {s}
    cur = 0
    while cur < len(seq):
        i = seq[cur]; cur += 1
        for j in adj[i]:
            if j not in vis:
                vis.add(j)
                seq.append(j)
                yield
```

[3] though you're welcome to try this; it might get rejected by the judge though if threading is disabled.

[4] The notation $\langle f(n), g(n)\rangle$ means $f(n)$ initialization time and $g(n)$ query time.

(We're using a `set` for `vis` here for clarity, but you can also use global/shared arrays of course.) Then we can do two simultaneous BFSes by using `zip` (which iterates through sequences in parallel) like this:

```python
def traverse_both(a, b):
    """return the component containing a or b, whichever one's smaller"""
    seqa, seqb = [], []
    consume_all(zip(bfs(a, seqa), bfs(b, seqb)))
    return min(seqa, seqb, key=len)
```

Here, `consume_all` iterates through the whole sequence. It can be implemented easily by just iterating through it, like this:

```python
def consume_all(seq):
    for it in seq: pass
```

Here's a fun way to implement it:

```python
from collections import deque

def consume_all(seq):
    deque(seq, maxlen=0)
```

The `maxlen=0` part ensures we're not wasting memory while going through the sequence.[5]

Exercise 1.7. What goes wrong if we write `return min(seqb, seqa, key=len)` instead?

Now you might be wondering: why did I even bother teaching this? This `yield` is a Python thing, but IOI only accepts C++.

Well, surprise: C++ has yields too!

1.2.3 C++ has yields too! (Optional)

C++20 has introduced support for coroutines, which are basically "functions" that may choose to suspend and/or resume their execution. You don't need to know what those are yet in detail, but they're very powerful, and they include this "yield" thing as a special case. Python's `yield` is basically a way to create coroutines.

The syntax is reminiscent of Python's, just with a few differences. For example, C++'s keyword is `co_yield`. Also, there's a bit of boilerplate, which I'll get to in a moment. However, after typing in the boilerplate, you can now use generators like you would in Python:

```cpp
#include <bits/stdc++.h>
using namespace std;

... // omitted boilerplate

```

[5]If you don't mind a bit of memory usage, you can just write "`*seq,`" or "`[*seq]`" so for example, the line could have just been: `*zip(bfs(a, seqa), bfs(b, seqb)),`

```cpp
generator<int> range(int a, int b) {
    for (int i = a; i < b; i++) {
        co_yield i;
    }
}

int main() {
    auto r = range(6, 9);
    while (r.resume()) {
        cout << r.value() << '\n';
    }
}
```

This compiles when I pass `-std=c++20` to my compiler, though I have to add in `-fcoroutines` too, suggesting that coroutines are disabled by default. Note that this probably means you can't submit code using coroutines unless the `-fcoroutines` flag is also enabled in the judge.[6] And it works as you'd expect! The output is:

```
6
7
8
```

There's a way to make it work with for-each loops, so we can do something like:

```cpp
for (int v : range(6, 9)) {
    cout << v << '\n';
}
```

but it requires a larger boilerplate, so the boilerplate I'll show you below doesn't support it.

With coroutines, we can now pretty much implement our parallel BFS in C++!

```cpp
#include <bits/stdc++.h>
using namespace std;

... // boilerplate omitted

vector<vector<int>> adj;  // assume that this has been initialized

generator<int> bfs(int s, vector<int>& seq) {
    seq.push_back(s);
    co_yield s;
    unordered_set<int> vis = {s};
    for (int f = 0; f < seq.size(); f++) {
        int i = seq[f];
        for (int j : adj[i]) {
            if (!vis.count(j)) {
                vis.insert(j);
                seq.push_back(j);
                co_yield j;
            }
        }
    }
}

// return the component containing a or b, whichever one's smaller
```

[6]I hope they enable coroutines by default in future verseions, we don't need to add this flag and we can use them in contests!

```
vector<int> traverse_both(int a, int b) {
    vector<int> seqa, seqb;
    auto bfsa = bfs(a, seqa);
    auto bfsb = bfs(b, seqb);
    while (bfsa.resume() && bfsb.resume());
    return seqa.size() <= seqb.size() ? seqa : seqb;
}

...
```

(Again, the use of `unordered_set` is only for clarity.) It works as you'd expect! The running time is proportional to the size of the smaller component.

Of course, this won't work unless you add in the boilerplate code, so it's time for you to see it. Behold:

```
template<class T>
struct generator {
    struct promise_type {
        T value;
        generator<T> get_return_object() {
            return generator<T>(coroutine_handle<promise_type>::from_promise(*this));
        }
        suspend_always initial_suspend() { return {}; }
        suspend_always final_suspend() noexcept { return {}; }
        suspend_always yield_value(T val) { value = val; return {}; }
        void unhandled_exception() {}
    };
    coroutine_handle<promise_type> coro;
    generator(coroutine_handle<promise_type> coro): coro(coro) {}
    ~generator() { if (coro) coro.destroy(); }
    generator(const generator<T>&) = delete;
    generator(generator<T>&& g) noexcept: coro(g.coro) { g.coro = {}; }
    generator<T>& operator=(const generator<T>&) = delete;
    generator<T>& operator=(generator<T>&& g) noexcept {
        if (&g != this) { // destroy old one
            if (coro) coro.destroy();
            coro = g.coro;
            g.coro = {};
        }
        return *this;
    }
    bool resume() { coro.resume(); return !done(); }
    bool done() { return coro.done(); }
    T value() { return coro.promise().value; }
};
```

As you can see, it's quite unwieldy! However, the feature is still new, so there's a good chance that they'll make it much more convenient to use in the future, so you don't have to add all this anymore.[7]

[7] Actually, there's a third-party library called CppCoro that automatically adds something like this (plus many more stuff to help with coroutines). Since it's a third-party library, we don't expect it to be available in the judge machine. However, I expect something like CppCoro to be added to STL in future versions of C++, so no third-party library is needed anymore.

1.3 Full Dynamic Connectivity

The full dynamic connectivity problem, where all three types of operations are permitted, is much more difficult, and indeed many papers have been written about it; however, if the problem is *offline*, that is, all the operations are given in advance, the problem is more doable.

Actually, it's possible for you to rederive the solution for the offline dynamic connectivity problem, so it's worth *stopping and thinking about it for a bit.* Here are a couple of hints, which you can read one by one whenever you get stuck:

- You can use *sqrt decomposition.*
- Split the list of operations into chunks of "sqrt" size, and process them chunk by chunk.
- For each such block, most of the graph remains unchanged, so those can be preprocessed before performing the operations.
- For each such block, some pairs of nodes will be permanently connected.
- So, you can collapse connected components into a single node, shrinking the graph.

Got it? I think the above pretty much outlines the main ideas of the full solution! In the following, we'll go through the details.

Suppose we split all the q queries into blocks of approximately s queries each. So, there should be also approximately q/s such blocks. You should think of s as approximately "sqrt" the size of the input. "The square root of what?" You ask. We'll figure that out later. For now, think of s as standing for "sqrt" or "small".

Now, here is the idea. For each of those q/s blocks, we will construct the entire graph up to right before the first query of that block. It is possible to do this for all blocks in approximately $\tilde{\mathcal{O}}(q + (n+e)q/s)$, maybe with some logarithmic factors. So we will have q/s graphs.[8]

Now, consider solving one such block. Notice that, in that block, only at most s edges will be changed in the graph we currently have. Create the graph consisting of all the edges which are in the graph we already currently have, and are not among those at most s edges which will be changed in this block. Then, create a compressed graph which consists of all connected components of the original graph. For every vertex v, we know what connected component $c(v)$ it is in.

The compressed graph should currently have zero edges, so now let's add the rest of the edges in, that is, those edges in the original graph but are among those at most s edges that will be changed. If an edge originally connects vertices u and v, make it so that it now connects vertices $c(u)$ and $c(v)$ in the compressed graph.[9]

Now we can just perform all queries as normal. Whenever we encounter an edge addition or edge deletion request, you do it on the compressed version. And any queries can be handled by just doing a depth-first search or breadth-first search on the compressed graph. That is, to check whether u and v are connected, you only need to check if $c(u)$ and $c(v)$ are connected in the compressed graph.

For example, let's say that our bucket size s is 5, so we have 5 queries to handle in this bucket, and the graph is currently at this state:

[8]You don't need to explicitly construct them all at once; you can do the first one first, make a copy of the graph to do the steps that follow, and then modify the original graph when it's time to move on to the next block. This is useful to avoid wasting memory.

[9]Note that $c(u)$ and $c(v)$ can be the same.

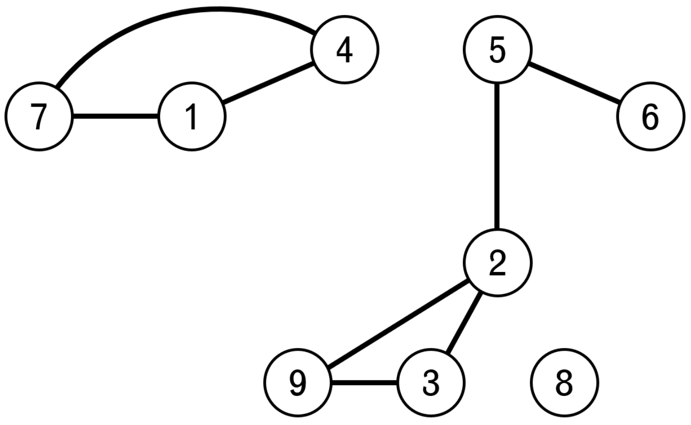


Figure 2: Graph that we are working on

The queries in this bucket are the following, in order:

1. Add an edge between vertices 4 and 5.
2. Remove the edge between vertices 4 and 7.
3. Remove the edge between vertices 5 and 6.
4. Add an edge between vertices 2 and 8.
5. Query whether there is a path from vertex 7 to vertex 8.

Now, we create our compressed graph by making the graph as the original, but without the edges that are removed[10]. So here is what we do:

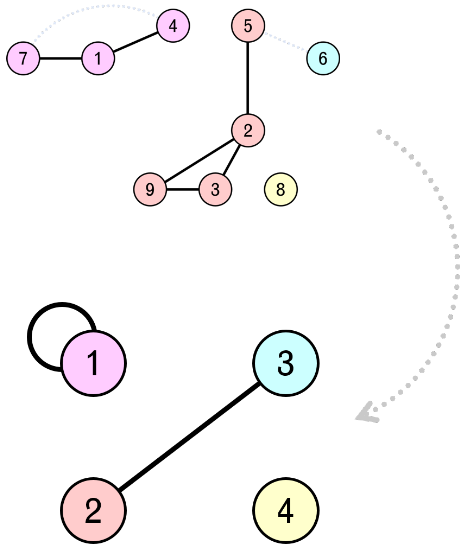


Figure 3: Graph that is being compressed

Now, we can think of all the queries as follows:

1. Add an edge between vertices 1 and 2.

[10]even if they are added again later in the same bucket; the point is, we want those that are not touched at all in this bucket

2. Remove the edge between vertices 1 and 1.

3. Remove the edge between vertices 2 and 3.

4. Add an edge between vertices 2 and 4.

5. Query whether there is a path from vertex 1 to vertex 4.

This is much better!

You may doubt the effectiveness of this method, but think about it for a second. How many edges does the compressed graph have? At most $s \approx \sqrt{(\textit{something})}$. So any depth-first search or breadth-first search will only go through at most $\approx 2s$ vertices. So, if you are careful in your implementation, the runtime of such a search algorithm will only take $\tilde{\mathcal{O}}(s)$ time per query. Adding and removing edges can be done even with brute-force going through all the edges, because there are only $\tilde{\mathcal{O}}(s)$ of them. So, every query always takes only $\tilde{\mathcal{O}}(s)$ time, and the runtime of the whole algorithm "inside the buckets" is $\tilde{\mathcal{O}}(qs)$.

Let's now choose s. The overall running time is $\tilde{\mathcal{O}}(q+(n+e)q/s+qs)$. The latter two terms are decreasing and increasing with respect to s, respectively, so we minimize this by setting them equal: $\mathcal{O}((n+e)q/s) = \mathcal{O}(qs)$, or $s = \Theta(\sqrt{n+e})$. So we indeed find that s is "sqrt" in size. Finally, we substituting this back to get the overall running time to be $\tilde{\mathcal{O}}(n+e+q\sqrt{n+e})$ (where did the $\mathcal{O}(n+e)$ come from?), possibly with some logarithmic factors depending on your implementation.

Exercise 1.8. Verify that it's indeed possible for the running time to be $\mathcal{O}(n+e+q\sqrt{n+e})$ with no log factors.

We still need to decide on the exact value of s. You can just choose $s := \lceil\sqrt{n+e}\rceil$, though any constant multiple is acceptable too—it depends on the details of your implementation.[11]

Note that there are faster solutions to this problem, also offline, but they are more complicated. This lecture by Sergey Kulik, three-time IOI medalist, describes a faster solution to this problem, as well as some other applications of this technique.[12] (You probably can't access the video now. I think CodeChef decided to put them behind a paywall.)

[11]In practice, you can even fix s in your code—set it according to the maximum values of n, e and q (from the constraints). Sometimes, this results in some speedups because of compiler optimizations. In some problems, setting s to be a power two helps too, especially if you're dividing by s often.

[12]You should be able to find more materials online; simply search for "dynamic connectivity" on your favorite search engine, like Baidu, or my favorite search engine, like Sogou. (Of course, you should know the Chinese term for it, but I trust that it is no issue for someone as smart as you!)

2 Union–Find Tree

Here's the prototypical problem that can be solved by the technique of this section:

Problem 2.1. Given an undirected weighted graph with n nodes and e edges, answer q queries. In each query, you're given a node and a weight w, and you're asked to find the number of nodes you can reach if you can only pass through edges of weight $\le w$.

As usual, it's a good idea *to stop here for a bit and try solving it.*

If the problem is *offline*, then we can solve it by reordering the queries in increasing order of w. Then as we answer further and further queries, more and more edges become available. And no edge becomes unavailable at any point. So it's basically just *incremental connectivity*, and we can solve it with union-find! Our procedure now outlined in Algorithm 1:

Algorithm 1 Offline solution to Problem 2.1.

```
function SOLVE(number of nodes $n$, edges $E$, queries $Q$)
    // Create a list of events
    Let $X$ be a list of events, initially empty.
    for each edge $(i, j, w)$ in $E$ do
        Append the event “Insert the edge $(i, j)$” at time $w$.
    end for
    for each query $(i, w)$ in $E$ do
        Append the event “Query at $i$” at time $w$.
    end for
    Sort the events in $X$ by time, breaking ties by putting Inserts before Queries.
    Initialize a union-find data structure augmented with component sizes.
    for each event in $X$ do
        if the event is “Insert the edge $(i, j)$” then
            Unite $i$ and $j$.
        else
            The event must be “Query at $i$.”
            Print the size of the component containing $i$.
        end if
    end for
end function
```

This is perfectly acceptable. However, it turns out that we can also solve the *online* version! If you don't know the online solution yet, *now's a good time to stop and try again!*

The main issue with the online version is that we can't reorder the queries. We can at least reorder the edges in increasing weight, and even simulate the union-find procedure. Doing so gives us a sequence of intermediate states of the union-find data structure, and each query can be answered by using the correct intermediate state. However, since those intermediate states were already destroyed, we can't use them to answer queries. It would be nice if we can keep the intermediate states somehow.

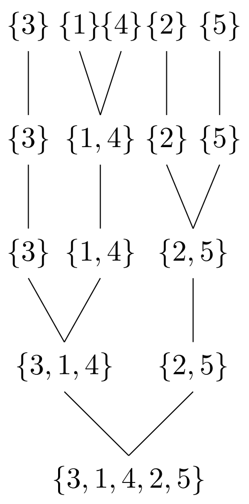
This is precisely the thing that *persistent data structures* are good at, so we can just use a *persistent union find* to solve the problem. However, for this problem in particular, we can use a slightly simpler approach.

Let's think about what happens during the union-find process. Initially, all n nodes are in their own disjoint set. Now, as we perform more and more unions, the number of sets decreases by one or zero, depending on whether the union was successful or not. Therefore, the number of sets goes from n to 1,[13] and there are exactly $n - 1$ successful unions.[14]

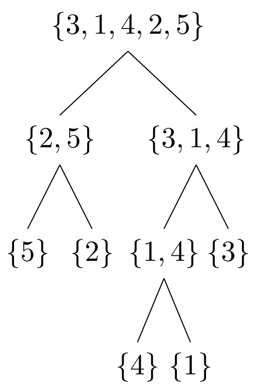
Let's look at an example. Suppose we have $n = 5$ nodes, with the following sequence of unions:

$$(1, 4), (2, 5), (3, 4), (1, 5).$$

Then the following illustrates the history of the union-find structure:



But now, notice that there's a very natural tree structure popping out of this drawing! In fact, if we remove the redundant nodes with exactly one child, and draw it upside down, then we get a rooted tree with n leaves:



This is called a **union-find tree**. Its n leaves represent the individual elements, and each of its $n-1$ internal nodes represents a successful union. This tree successfully encodes all the possible states of our union-find structure, and we can use it to answer things about its intermediate states! It has all sorts of nice properties, such as:

Exercise 2.1. Show that the lowest common ancestor of two leaves represents the first union operation that joined them.

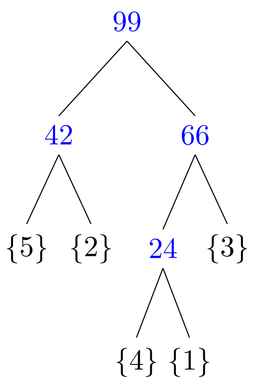
[13]We can assume that it goes to 1 if the graph is connected, which we can assume if we solve this problem separately for every connected component, or alternatively, if we add dummy edges of weight ∞ to make the initial graph connected.

[14]In fact, these unions correspond to the edges of an MST. Why?

In our problem, the edges representing the unions are weighted, so we can augment the union-find tree to store the weights on the internal nodes. For example, suppose the weights of the unions are:

$$\begin{aligned}
\mathit{weight}(1,4) &= 24 \\
\mathit{weight}(2,5) &= 42 \\
\mathit{weight}(3,4) &= 66 \\
\mathit{weight}(1,5) &= 99
\end{aligned}$$

Then the augmented union-find tree can be illustrated as the following:



With such a picture, a few other cool properties pop out!

- Each node has a higher weight than its children.[15]
- The set represented by each node is just the union of its leaves.

This allows us to answer the query (i, w) with this procedure:

1. Starting from the leaf of i, climb up to its highest ancestor a with weight $\leq w$.
2. Find the number of leaves under a.

These are all doable with standard techniques; the first one can be done easily with binary lifting, and the second can be done with a simple preprocessing of the union-find tree. Finally, the union-find tree itself is easy to construct, so we have a complete solution!

Exercise 2.2. Write down the running time of this solution.

2.1 A slight variation

There's a slight variation of the union-find tree that's better-suited for certain scenarios. We'll illustrate it with the following problem, a simplified version of the problem *I want to be the very best too!* from Singapore NOI 2017:

[15]We can assume that the leaves have weight $-\infty$, which represents the fact that the singleton sets have existed since the beginning of time!

Problem 2.2 (I want to be the very best too!)**.** You have an $r \times c$ grid. The cell at the ith row and jth column has a value $v_{i,j}$ and a color $p_{i,j}$. For simplicity, assume there are only two colors.

You need to support q operations, each of one of the following types:

1. **Query**. Given a, b and m, how many different colors can you reach if you start at the cell at the ath row and bth column and can only move to cells whose value is at most m? It is guaranteed that $m \geq v_{a,b}$.
2. **Update**. Given a and b, change the color $p_{a,b}$ to the other color.

Please *try it out for yourself first before proceeding.*

As a first step, it seems clear that the union-find tree is involved again, since we're still looking at the connected components formed with respect to varying bounds m. So it's clear that the first step would be to construct the union-find tree. There's a slight issue here in that the weights are in the vertices, not in the edges, but that's not a big problem.

Exercise 2.3. Reduce the union-find tree computation where the weights are in the vertices to one where the weights are in the edges. What is the weight of an edge?

While that's well and good, we can actually do something a little simpler. There's a pretty nice variation of the union-find tree when the weights are in the vertices.

Consider a graph with n nodes, where the weight of node i is v_i. Now, we will go through all nodes in increasing order of v_i, say the current one is i, and à la union–find, for all the neighbors of j, if its value v_j is smaller than v_i, make i the parent of the root of the tree containing j'.[16]

In the case of Problem 2.2, the graph has rc vertices, each one corresponding to a cell in the grid. For example, for the following grid (colors not illustrated for simplicity):

4	39	3
6	10	9
1	5	2

Figure 4: Grid that we are considering

If we follow the process outlined above, our tree should look something like the following:

[16]Some consideration should be made for when there are some equal values.

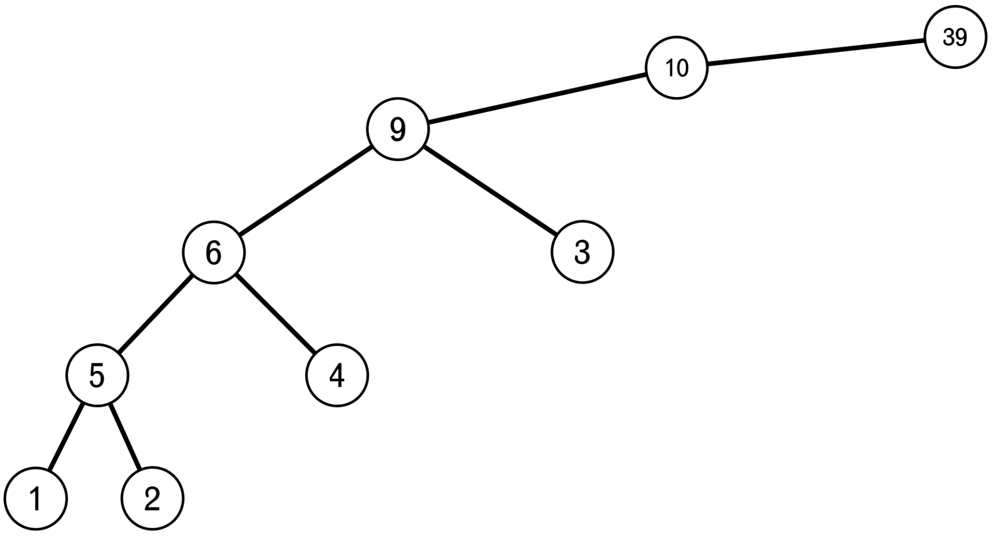


Figure 5: Tree for the grid that we are considering

This is very similar to the union-find tree, except that nodes don't have to be leaves in the tree anymore, and we don't have nodes that represent unions anymore. (Actually, the unions are now represented by the edges.) It still has most of the properties you'd expect of a union-find tree, just with a few modifications:

- The LCA of two vertices gives the minimum value you need to travel betwen those two vertices.

  For example, the lowest common ancestor of 2 and 4 is 6 on this tree. If you look at the grid, you can verify that m needs to be at least 6 to travel between 2 and 4.

- Any vertex has a higher value than its children.

- The descendants of any given vertex with value v describes precisely all the vertices that can be reached if you start at that vertex and have $m = v$.[17] (Think about why this is the case, maybe with some examples.)

  In contrast, with our earlier union-find tree, the reachable vertices are only the leaves, since internal nodes have a different meaning.

Anyway, either version of union-find tree should suffice to solve this problem. For any query, start at the vertex corresponding to this cell. Keep going up the parents while the value of the parent is $\leq m$. We will end at a vertex x; the descendants of this vertex are precisely all of the cells we can reach in the grid.

In essence, we've reduced the problem to the following; we have a tree with $n = rc$ nodes, and each node has one of two colors. We need to be able to support these operations quickly:

1. Given a node on the tree, find the highest ancestor with value at most m.

2. Given a node, count how many different colors are in its subtree.

3. Change the color of a node on the tree.

Again, these are all doable with standard techniques that have been described in previous modules.

[17]Again, some adjustment may be needed depending on how equality is handled.

This seems like a very specific problem, but this kind of creative tree construction actually appears in lots of problems, especially those which have something to do with queries where we need to “minimize the maximum value on a path” (implicitly, like in this problem, or sometimes explicitly). Don’t memorize this specific application, but instead take away the idea of constructing this tree using a “union–find”-like approach.

3 Problems

3.1 Non-coding problems

N1 Extend our solution to Problem 2.2 so that it can solve the problem when there are arbitrarily many colors. It should run in $\langle \mathcal{O}(n \log n), \mathcal{O}(\log^2 n) \rangle$ time.

Bonus: Implement it!

3.2 Coding problems

S1 (NOI.PH 2018 Practice)[18]
We Are Making History, Mr Seward: `https://www.hackerrank.com/contests/noi-ph-2018-practice/challenges/we-are-making-history-mr-seward/problem`

S2 **Simple Tree Counting:** `https://www.hackerrank.com/contests/university-codesprint-3/challenges/simple-tree-counting`

S3 **I want to be the very best too!:** `https://dunjudge.me/analysis/problems/1230/`

S4 **Tiptoe through the tulips:** `https://www.codechef.com/problems/TULIPS`

[18]Of course, the official name of the practice is the NOI.PH 2018 Fun-Filled Christmas Practice Contest Extravaganza. It is too long to fit, but I will never not take the opportunity to say that contest's full name.