Disjoint Sets – Union Find Move

Mon Mar 11 11:18:46 2019 +0100, rev. 6707dfe

Maintain a family of disjoint sets under repeated operations that perform unions of sets or move single elements from one set to another, using a tailor-made variant of a Union–Find data structure.

To solve this exercise, you need construct a new data structure, instead of just taking a well-known solution off the shelf, as we did in other exercises. In other words, this is a data structure *design* question, not an *application* question.

Description

Maintain a family of disjoint sets, initially the singletons

$$\{0\}, \{1\}, \ldots, \{n-1\}$$

under the following operations:

union The *union* operation takes two arguments s and t and creates the union of the two sets containing s and t. More precisely, if $s \in S$ and $t \in T$ with $S \neq T$ then S and T are removed from the family and replaced by the set $S \cup T$. (If $S = T$ then nothing happens.)

query The *query* operation takes two arguments s and t and determines if s and t belong to the same set. More precisely, if $s \in S$ and $t \in T$ then print `yes` if $S = T$ and print `no` if $S \neq T$.

move The *move* operation takes two arguments s and t and moves the element s to the set containing t, removing s from whatever set it may have belonged to before. More precisely, if $s \in S$ and $t \in T$ with $S \neq T$ then S and T are removed from the family and $S - \{s\}$ (if it is nonempty) and $T \cup \{s\}$ are added to the family. (If $S = T$ then nothing happens.)

Note that these operations maintain invariantly that the sets in the family are disjoint. The meaning of the union and query operations is exactly as in *Disjoint Sets I* and *Disjoint Sets II*.

Input

The input starts with two nonnegative integers n, m, on the first line. We have $0 \leq n \leq 10^6$ and $0 \leq m \leq 2 \cdot 10^6$. The integer n is the initial number of singletons. The following m lines are of the form "`u` *s t*" (for *union*) or "`q` *s t*" (for *query*) or "`m` *s t*" (for *move*); you can assume $0 \leq s < n$ and $0 \leq t < n$.

`4 9`	$\{0\}, \{1\}, \{2\}, \{3\}$
`q 0 1`	
`u 0 1`	$\{0,1\}, \{2\}, \{3\}$
`q 0 1`	
`u 1 2`	$\{0,1,2\}, \{3\}$
`q 1 2`	
`q 0 3`	
`m 2 3`	$\{0,1\}, \{2,3\}$
`q 0 1`	
`q 1 2`	

Figure 1: Sample input and interpretation.

```
no
yes
yes
no
yes
no
```

Figure 2: Sample output

Output

For each query, a single line with `yes` or `no` as described above.

Deliverable

Source code of solution. Solve the problem in Python or Java. Hand this in as `DSufm.java` or `DSufm.py`.

There is a set of instances on CodeJudge for you to test.

Report. Hand in a very brief report describing your data structure as a PDF. The solution should pass all tests on the Code Judge, and your report should say this and mention the resulting running times. **Please add the ID of the successful submission on Code Judge to your report.** Please include all the names and itu-email addresses of the group members as authors in the PDF of the report.

This is a difficult exercise because the standard Union–Find data structures you can find in libraries or text books will not solve it. (A moment's thought should convince you that the move operation can destroy the trees built in in the standard Union–Find data structure, typically when s is not a leaf in the data structure.) On the other hand, the instances are so large that an efficient solution is necessary.

A good idea is needed, and your report should explain it briefly and clearly. A figure is probably useful. (Draw it by hand and scan it, unless you are really familiar with some good drawing tools.)

Your solution must use at most logarithmic time (in n) per update, and total space linear in n.

Report: Disjoint Sets – union find move

by Alice Cooper, Bob Marley, and Charlie Brown[1]

We solved this exercise in Python 3 / Java.

Program `DSufm` passes all tests on the CodeJudge, with a maximum of running time of 4 seconds.

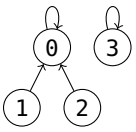
[1] Complete the report by filling in your real name and changing other parts wherever necessary. Remove the sidenotes in your final hand-in.

Our data structure is a modification of Weighted Max Flow[2] except that we also maintain a homeotoxic metasnail for every inverse tachyon flux capacitator. (As in [SW, sec. 1.5], we keep track of parents in `id` and component sizes for the roots in `sz`, but these are now flux capacitators.)

[2] The description, including the illustration, is utter nonsense and just here to indicate how many sentences you should roughly use. Replace the nonsense.

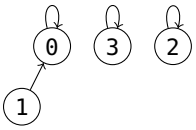
Queries and unions are almost the same as in the standard solution, except for the extra step of subverting the flux capacitator. For $move(s, t)$, a new copy of the node is inserted in the particle reference queue, and the nunstuck is popped.

For instance, just before the move operation in the sample input, the data structure looks like this:



element	0	1	2	3
id	0	0	3	
sz	3	–	–	1
flux	π			

After the move, element 2 is again just a reference to the inverse flux capacitator, and its old tachyon node remains untouched.



element	0	1	2	3
id	0	2	3	
sz	3	–	–	1
flux	2π			

We claim without proof [3] that the running time is logarithmic in n in the worst case per update and the data structure requires linear space in n space.

[3] If you want, you can give a proof. This is not mandatory.