CS213/293 Data Structure and Algorithms 2024

Lecture 16: Union-find for disjoint sets

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-10-29

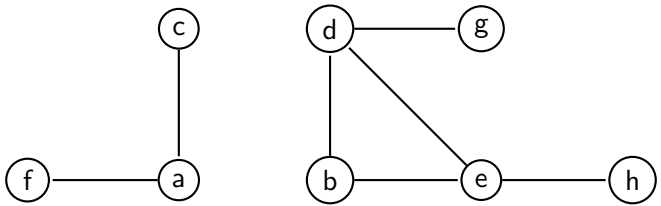
Topic 16.1

Disjoint sets

Example: connected components are disjoint sets

Example 16.1

Consider the following graph.



The connected components are disjoint sets of nodes $\{c, a, f\}$ and $\{d, g, b, e, h\}$.

Union-find data structure

The data structure that is used for the disjoint sets is called "union-find".

Interface of UnionFind

UnionFind supports four interface methods

- `UnionFind<T> s` : allocates empty disjoint sets object s
- `s.makeSet(x)` : creates a new set that contains x.
- `s.union(x,y)` : merges the sets that contain x and y and returns representative of union
- `s.findSet(x)` : returns representative element of the set containing x.

Repeated calls to `s.findSet(x)` returns the same element if no other calls are made in between.

Exercise 16.1

What other methods should be in the interface?

Commentary: Answer: The interface does not let you enumerate the sets. We cannot ask for the sizes. One can think of the extensions depending on the applications.

Connected components via UnionFind

Instead of BFS, we may use UnionFind to find the connected components.

Algorithm 16.1: ConnectedComponents(Graph $G = (V, E)$)

```
UnionFind s;
while v ∈ V do
    s.makeSet(v)
while {v, v'} ∈ E do
    if s.findSet(v) ≠ s.findSet(v') then
        s.union(v.v')
return s;
```

Exercise 16.2

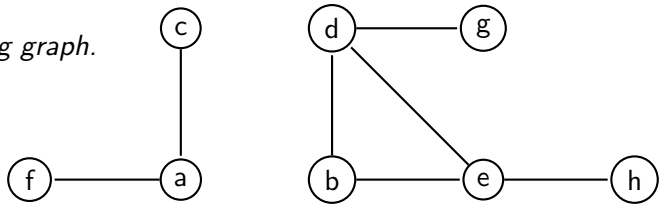
Given the returned s,

- *can we efficiently check the number of connected components?*
- *can we efficiently check if two vertices belong to same component or not?*
- *can we efficiently enumerate the vertices of a component?*

Example: run union-find for connected components

Example 16.2

Consider the following graph.



After processing all nodes: we have sets $\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{c\}$
After processing edge $\{d, b\}$: we have sets $\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$
After processing edge $\{a, f\}$: we have sets $\{a, f\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{g\}$ $\{h\}$
After processing edge $\{e, h\}$: we have sets $\{a, f\}$ $\{b, d\}$ $\{c\}$ $\{e, h\}$ $\{g\}$
After processing edge $\{d, g\}$: we have sets $\{a, f\}$ $\{b, d, g\}$ $\{c\}$ $\{e, h\}$
After processing edge $\{d, g\}$: we have sets $\{a, f\}$ $\{b, d, g\}$ $\{c\}$ $\{e, h\}$
After processing edge $\{a, c\}$: we have sets $\{a, f, c\}$ $\{b, d, g\}$ $\{e, h\}$
After processing edge $\{d, e\}$: we have sets $\{a, f, c\}$ $\{b, d, g, e, h\}$
After processing edge $\{b, e\}$: we have no change.

Recall: breadth-first based connected components

Algorithm 16.2: CC(Graph $G = (V, E)$)

for $v \in V$ **do**
    $v.component := 0$
$componentId := 1$;
while $r \in V$ *such that* $r.component == 0$ **do**
    BFSConnected($G, r, componentId$);
    $componentId := componentId + 1$;

In BFS, there was a step that needs to find next unvisited node after finishing a component.

Therefore, we needed to maintain a set of unvisited nodes.

Exercise 16.3

a. What is the needed interface for the set of unvisited nodes?
b. Can all the needed operations be done in unit cost?

Implementing UnionFind

If we can have efficient implementation of `findSet` and `union` then we may beat performance of BFS based connected components implementation.

What is the best can we do?

We are interested in amortised cost of the two operations, since in a practical use they are called several times.

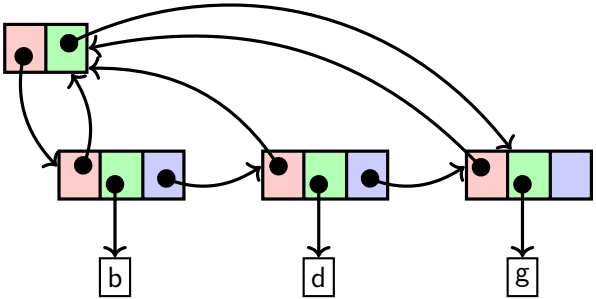
We will design an almost linear data-structure for unionFind in terms of the number of the calls to the operations.

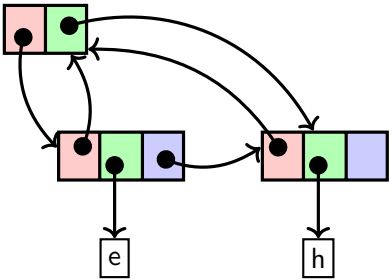
Topic 16.2

Union-find via Linked list

Disjoint sets representation via linked list

We need a node with two pointers to serve as the header of the set. It points to the head and tail.





Implementing findSet and makeSet

Exercise 16.4

a. Give implementation of makeSet?

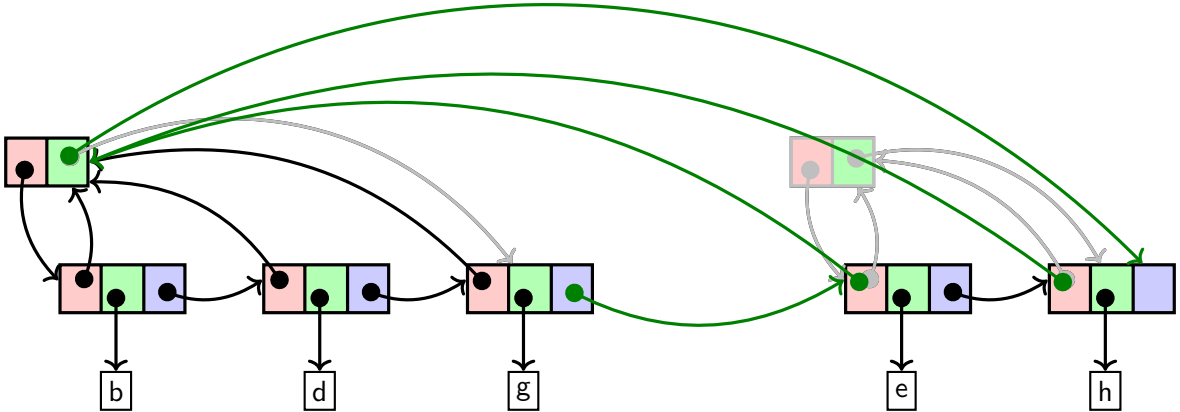
b. Give implementation of findSet?

Commentary: a. makeSet allocates a header node and a linked list, and links the nodes as suggested in the previous slide. b. findSet(node) { return node-¿header-¿head; }

Implementing union for UnionFind

Exercise 16.5

On calling union(node(d), node(e)), the gray edges are removed and green edges are added.



Exercise 16.6

Write code for the above transformation.

Running time of union

Theorem 16.1

For n elements, there is a sequence of n union calls such that total time of the calls is $O(n^2)$.

Proof.

Consider the following sequence of calls after creating nodes x_1,..., x_n.

$union(x_2, x_1)$, $union(x_3, x_2)$, $union(x_4, x_3)$, ..., $union(x_n, x_{n-1})$

At ith call, the union will update pointers towards headers in i nodes.

Therefore, the total run time is $\sum_{n=1}^{n-1} O(i)$, which is $O(n^2)$. □

Can we do better?... Yes.

We will consider three ideas to make the running time better.

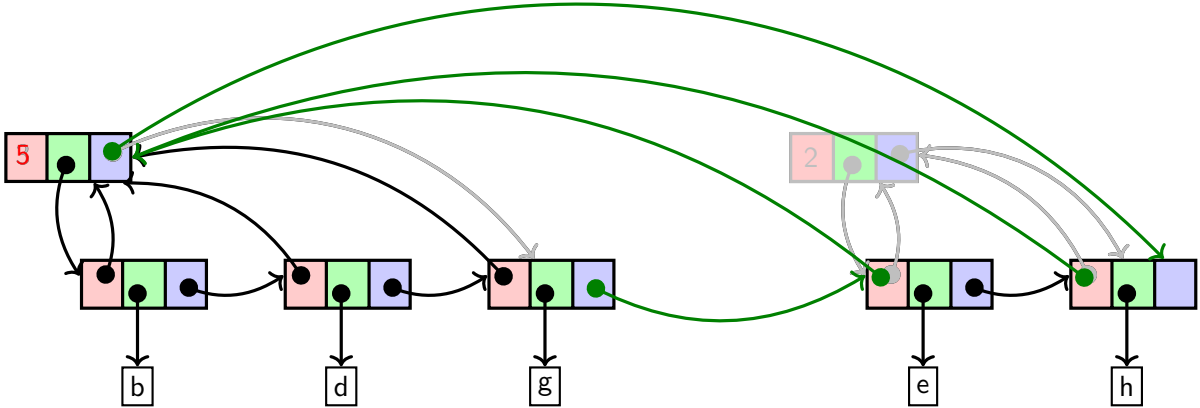
Topic 16.3

Idea 1: weighted-union heuristics

weighted-union heuristics

Update only the shorter list.

Add third field in the header node to store the length of the array. Exchange the parameters in $union(x, y)$ if the set containing y has longer length.



Running time with weighted-union

Theorem 16.2

Each time a header pointer is updated in a node x, x joins a set with at least double length.

Proof.

Let us suppose x is in a set of size k and y is is in a set of size k'.

During call $union(x, y)$, the header pointer of x is updated if $k' \geq k$.

After the union, total size of the set will be $k + k' \geq 2k$. □

Running time with weighted-union(2)

Theorem 16.3

Let us suppose there are n elements in all disjoint sets. The total running time of any sequence of n unions is $O(n \log n)$

Proof.

For each node the header pointer cannot be updated more than $\log n$ times.

Therefore, the total run time is $O(n \log n)$. □

Can we do better?.... **Yes**

Topic 16.4

Idea 2: UnionFind via forest

Each set is a tree

To avoid long traversals along the linked lists, we may represent sets via trees.

The root of tree represents the set.

Each node has only two fields: parent and size.

Forest implementation of UnionFind

Algorithm 16.3: MakeSet(x)

```
x.parent = x;
x.size = 1;
```

Algorithm 16.4: FindSet(x)

```
if x.parent ≠ x then return FindSet( x.parent ) ;
return x.parent;
```

Exercise 16.7

Is FindSet tail-recursive?

Forest implementation of UnionFind(2)

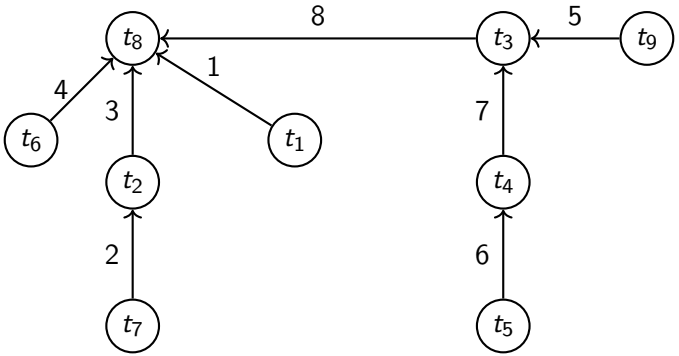
Algorithm 16.5: UNION(x, y)

x := FINDSET(x);
y := FINDSET(y);
if *x.size* $<$ *y.size* **then** SWAP(x,y) ;
y.parent := x;
x.size = x.size + y.size

Example: unionFind for equality reasoning via forest

Example 16.3

Consider: $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



Running time of UnionFind via forest

Theorem 16.4

The running time is still $O(n \log n)$.

Can we do better?.... **Yes**

Exercise 16.8

Show that if the height of a tree is h, then it has at least 2^{h-1} nodes.

Commentary: Assume all trees with height h have nodes 2^{h-1} Consider $union(x, y)$, let y be the smaller tree with height h. Then, the number of nodes is 2^{h-1}. The tree T_x containing x will have at least 2^{h-1} nodes. Let us suppose the height of $T_x \leq h + 1$. If the height of tree is $h + 1$ after the union, then the number nodes is more than 2^h. If height of $T_x \geq h + 1$. There is no height change and only more nodes are added.

Topic 16.5

Idea 3: Path compression

Path compression

Let us directly link the a node to its representation, each time we visit a node during the run of FindSet.

Algorithm 16.6: FindSet(x)

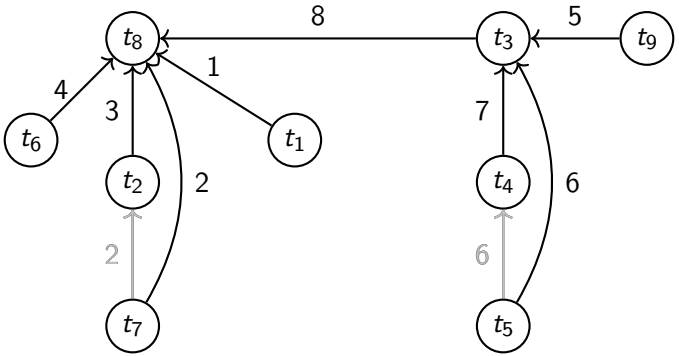
if $x.parent \neq x$ **then** x.parent := FindSet(x.parent) ;
return x.parent;

Exercise 16.9
Is FindSet *tail-recursive?*

Example: unionFind for equality reasoning with path compression

Example 16.4

Consider: $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



Running time of UnionFind with all three ideas

The running time is $O(n\alpha(n))$, where α is a very slow growing function.

For any practical $n < 10^{80}$, $\alpha(n) \leq 4$.

The final data-structure is almost linear.

The proof of the above complexity is involved. Please read the text book for the proof.

Commentary: Watch the following interview by Tarzan who proved the bound. https://www.youtube.com/watch?v=Hhk8ANKWGJA

Topic 16.6

Proof of work

Proof of work

Should we be content if an algorithm says that the formula is unsatisfiable?

We must demand "why?".

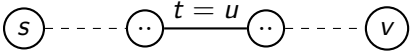
Can we generate a proof of unsatisfiablity using UnionFind data-structure?

Proof generation in UnionFind (without path compression)

Proof generation from UnionFind data structure for an unsatisfiable input.
The proof is constructed bottom up.

1. There must be a dis-equality $s \neq v$ that was violated.
   We need to find the proof for $s = v$.

2. Find the latest edge in the path between s and v. Let us say it is due to input literal $t = u$.



   Recursively, find the proof of $s = t$ and $u = v$.

   We stitch the proofs as follows

$$\frac{\dfrac{\dots}{s = t} \quad t = u \quad \dfrac{\dots}{u = v}}{s = v}$$

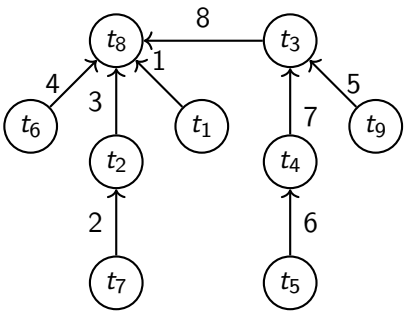
For improved algorithm: R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. RTA'05, LNCS 3467

Commentary: We may need to apply symmetry rule to get the equality in right order.

Example: union-find proof generation

Example 16.5

Consider: $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



$$\dfrac{\dfrac{\dfrac{}{t_7 = t_1}}{t_1 = t_7} \quad t_7 = t_5 \quad t_5 = t_4}{t_1 \neq t_4 \qquad t_1 = t_4}$$
$$\bot$$

1. $t_1 \neq t_4$ is violated.
2. 8 is the latest edge in the path between t_1 and t_4
3. 8 is due to $t_7 = t_5$
4. Look for proof of $t_1 = t_7$ and $t_5 = t_4$
5. 3 is the latest edge between t_1 and t_7, which is due to $t_7 = t_1$.
6. Similarly, $t_5 = t_4$ is edge 6

Topic 16.7

Tutorial Problems

Exercise: undo problem

Exercise 16.10

Give an algorithm to undo the last union operation assuming there is path compression or not?

Exercise: tight bounds

Exercise 16.11

Show that complexity bounds in theorem 16.3 and theorem 16.4 are tight?

Exercise: printSet

Exercise 16.12

Give an implementation of printSet(x) function in UnionFind (with path compression) that prints the set containing x. You may add one field in each node and must not alter the asymptotic running times of the other operations.

End of Lecture 16