Week 5

Data Structures for Disjoint Sets

1. Introduction
2. Operations
3. Simple data structure
4. Advanced data structure
5. Final remarks

Sets again

Last week we have implemented *dynamic sets* using *binary search trees*.

- The essence of dynamic sets is that we have just one set, which is growing and shrinking, and where we want to check elementship.
- Additionally we want also to determine extreme elements (minimum and maximum), and get from one element to the next resp. previous one.

Now we have several sets, and the basic operations are

- determining for an object in which of the sets it is
- computing the union (absorbing the old sets).

However this is not done for general set union, but only for *disjoint set union* — this is an important special case, where we have very fast algorithms.

Data Structures for Disjoint Sets

Read:

- CLRS Chapter 21 (not Section 21.4).

The problem

Maintaining a collection

$$\mathcal{S} = \{ S_1, S_2, \ldots, S_k \}$$

of disjoint sets.

Each set S_i is *represented* by an element $x \in S_i$.

The collection can change over time; thus these represent *dynamic* sets.

They are implemented by *disjoint-set data structures*.

Basic operations

MAKE-SET(x) creates a new set whose only element is x. Its representative is of course x.
(*Assumption*: x does not already appear in any of the existing sets.)

UNION(x, y) combines the set S_x containing x and the set S_y containing y, forming a single new set S.
The representative of this new set S is usually chosen to be either the representative of S_x or the representative of S_y.
(*Side effect*: S_x, S_y no longer exist by themselves.)

FIND-SET(x) returns (a pointer to) the representative of the set containing x.

Which representatives?

Is it possible that FIND-SET(x) on one occasion returns z, and on another occasion a different z' ?

- No — it is guaranteed that representatives stay the same if the sets concerned are not touched.
- Thus as long as no UNION-operations are performed, the return-value of FIND-SET are stable.
- And furthermore UNION(x', y')-operations only affect the return-values of calls for x in either the old $S_{x'}$ or $S_{y'}$.

Often actually the precise return-value of FIND-SET(x) is not of relevance, but it is only used to determine whether two different x, x' are in the same set — this is the case if and only if FIND-SET(x) == FIND-SET(x') holds.

Elements versus pointers

One further important clarification is needed:

Disjoint-sets data structures are not designed for searching!

So the inputs for $\textsc{Union}(x, y)$ and $\textsc{Find-Set}(x)$ are in fact

not the elements themselves, but
pointers ("iterators") **into the data structure**.

Thus we don't need to search for x and y in the data structure, but the input is already their place in it. However, how to obtain these "handles" for the elements?

- $\textsc{Make-Set}(x)$ still has as input an element x itself — there is no pointer to it yet.
- So actually $\textsc{Make-Set}(x)$ needs to **return** the pointer ("handle") to the place (node) in the data structure.
- This pointer has to be stored, and used instead of x when using $\textsc{Union}(x, y)$ or $\textsc{Find-Set}(x)$.

Preview on graphs

Next week we will be exclusively concerned with "graphs". Here now we consider a nice application to graphs.

- A graph consists of *vertices* (arbitrary objects), and *edges*.
- An edge in an *undirected graph* (the default, and just called *graph*) connects two vertices v, w.
- As a mathematical object an edge is just a 2-element set $\{ v, w \}$ (note that sets have no order, and thus the edge is undirected).

For example the following is a graph with 8 vertices and 6 edges:

This graph has *three connected components*.

Connected components

A natural application of disjoint-set data structures is for computing the *connected components* of a graph.

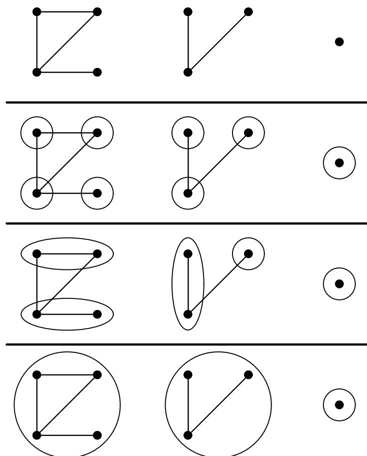
Input: An undirected graph G.
Output: The connected components of G.

```
CONNECTED-COMPONENTS(G)
1  for each vertex v of G MAKE-SET(v);
2  for each edge { u, v } of G
3      if (FIND-SET(u) ≠ FIND-SET(v)) UNION(u, v);
```

After computation of the connected components, we can determine whether two vertices u, v are in the same component (that is, are connected by some path) or not:

```
SAME-COMPONENT(u, v)
1  if (FIND-SET(u) == FIND-SET(v)) return TRUE;
2  else return FALSE;
```

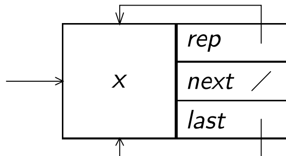
Connected components illustrated



Linked-list representation

Idea:

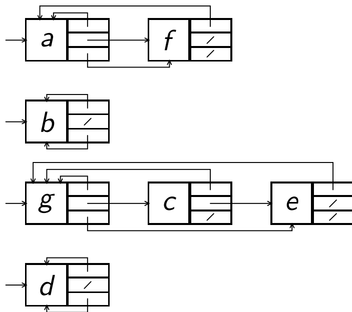
- Each element is represented by a pointer to a cell.
- We then use a linked list for each set.
- Each cell has a *next* pointer to the next cell in the list, as well as a *rep* pointer to the representative element at the head of the list.
- Each cell also has a *last* pointer to the last element in the list; however, we shall only expect that this be correctly defined for the representative cell.



Example

A linked list representation of the sets

$$\{a, f\}, \quad \{b\}, \quad \{g, c, e\}, \quad \{d\}.$$



Some remarks on the list-structures

Above we used just one node-type, while in CLRS two node-types are used:

1. One type for the head of each list, one for ordinary nodes.
2. In this way one can save (potentially) some space, since the special information in the head-node doesn't need to appear in each ordinary node.
3. Especially when adding further information on the sets (i.e., to the head-nodes) this could become relevant.
4. However our implementation is simpler.

Further remarks on potential space savings

- And actually the CLRS-implementation might use more space, since when creating the singleton sets by Make-Set, two nodes have to be created, and these nodes are just carried around later.
- It is only that the ordinary nodes don't need to contain the `last`-pointer (and potential further information).
- So "later", when the number of sets shrinks (due to unions performed), the space gains could be realised.
- However in practice quite likely this does not show up, since one has to delete the superfluous head-nodes and release the memory occupied by them, which requires some effort.

Cost of basic operations

MAKE-SET(x): Constant time.

FIND-SET(x): Constant time. (Note that this relies on x being a *pointer* to the node containing x.)

UNION(x, y): A naïve implementation appends x's list onto the end of y's list. (Note that this is opposite to CLRS, where y is appended to x.)
This makes use of the *last* pointer.

Problem: You have to update the `rep` pointer in every cell in x's list to point to the head of y's list.
The cost of this is thus:

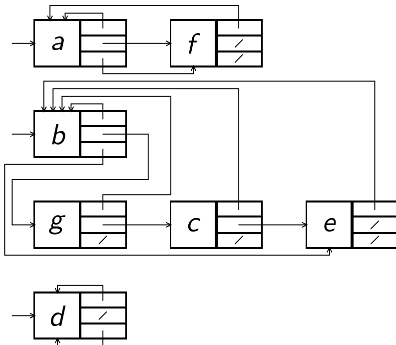
$$\Theta(\text{length of } x\text{'s list}).$$

Example

The operation

$$\textsc{Union}(c, b)$$

applied in the previous example would result in the following configuration.



Convention for runtime analysis

We shall express the runtime in terms of:

n : the number of MAKE-SET operations;
and

m : the total number of MAKE-SET, UNION, and FIND-SET operations.

Note:

1. We must have $m \geq n$.
2. After $n-1$ UNION operations, we have only one set remaining.

A nasty example

Consider the following sequence of $m = 2n - 1$ disjoint-set operations.

Operation	Number of objects updated
$\textsc{Make-Set}(x_1)$	1
$\textsc{Make-Set}(x_2)$	1
$\vdots$	$\vdots$
$\textsc{Make-Set}(x_n)$	1
$\textsc{Union}(x_1, x_2)$	1
$\textsc{Union}(x_2, x_3)$	2
$\textsc{Union}(x_3, x_4)$	3
$\vdots$	$\vdots$
$\textsc{Union}(x_{n-1}, x_n)$	$n - 1$
Total	$n + \frac{n(n-1)}{2} = \Theta(m^2)$

Thus the *amortised* (i.e., average) cost of each operation is $\Theta(m)$.

A weighted-union heuristic

Idea: Record the length of each list.

Then when executing

$$\textsc{Union}(x, y)$$

we append the shorter list onto the longer one (breaking ties arbitrarily).

Theorem: Using the linked-list representation of disjoint sets with this weighted-union heuristic, a sequence of m Make-Set, Union and Find-Set operations, n of which are Make-Set operations, takes

$$O(m + n \log n)$$

time.

Proof of Theorem

There are $\mathrm{O}(m)$ MAKE-SET and FIND-SET operations, each costing $\mathrm{O}(1)$ time, so these contribute $\mathrm{O}(m)$ time to the cost of executing the sequence.

For the UNION operations, we note that a cell's links are updated only when it is in the smaller of the two sets being UNIONed.

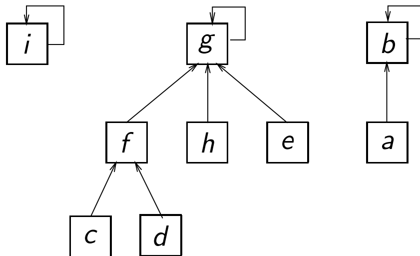
This can happen at most $\lceil \log n \rceil$ times (as the set containing a given element must at least double in size when that element is involved in a UNION operation which updates its links).

The total time spent in updating the n objects with the UNION operations is thus $\mathrm{O}(n \log n)$.

Therefore the cost of a sequence of m operations with n MAKE-SET operations is $\mathrm{O}(m + n \log n)$. □

Disjoint-set forests

Idea: Each set is represented by a rooted tree.



Remarks:

- The nodes of these tree only need the parent-pointer, and no pointers to children, since these trees are always traversed from the leaves towards the root.
- The root of each tree is recognisable by the fact, that its parent pointer point to itself.
- One could have used the ordinary nil-pointer as well.

Cost of basic operations

Introduction
Operations
Simple data structure
Advanced data structure
Final remarks

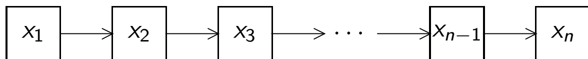
MAKE-SET(x): Constant time.

FIND-SET(x): This requires following the pointers to the root of x's tree. (The path followed is called the *find-path*.) The cost is thus proportional to the height of the tree.

UNION(x, y): The naïve strategy makes the root of x's tree point to the root of y's tree. The cost is thus proportional to the depths of x and y, that is, the lengths of their find-paths.

The nasty example revisited

The example sequence of operations produces forests consisting of one degenerated tree (just one linear chain of nodes) plus the singleton trees of yet untouched objects. Finally it all results in a single degenerated tree:



The cost of each successive UNION operation is proportional to the find-paths of the objects, which gets longer and longer.

Hence the basic disjoint-set forests implementation is no faster than the linked list implementation. (That is, regarding the *worst-case analysis* — in practice there might be substantial differences, also depending on the circumstances.)

Two heuristics

Union by size: At each root vertex, maintain a record of the size (i.e., number of nodes) of its tree.
Then when executing

$$\textsc{Union}(x, y)$$

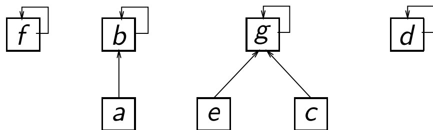
make the smaller tree point to larger one.
The point of this heuristics is to reduce the height of trees.
(Note that CLRS uses *rank* (i.e., depth) rather than size.)

Path Compression: When executing

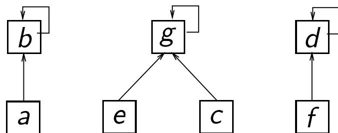
$$\textsc{Find-Set}(x)$$

make each vertex on the find-path point to the root.
Also this heuristics reduces the height, exploiting that our (rooted) tries can use arbitrary numbers of children at each node (here the root).

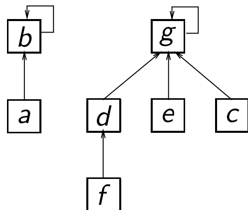
Example of union by size



UNION(f, d)

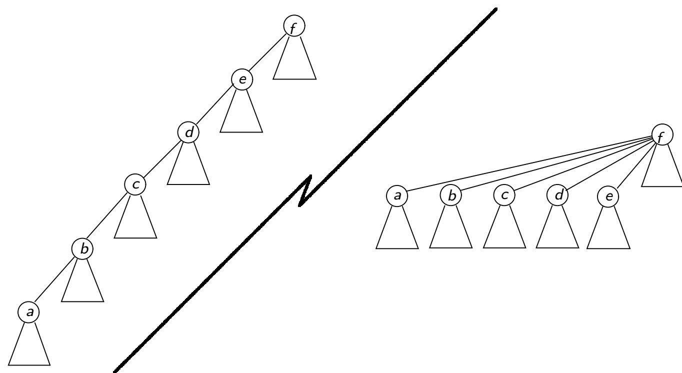


UNION(f, g)



Example of path compression

FIND-SET(a):



Pseudocode

```
MAKE-SET(x)
1  p[x] = x;
2  size[x] = 1;

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y));

LINK(x, y)
1  if (size[x] > size[y])
2        p[y] = x;
3        size[x] = size[x] + size[y];
4  else p[x] = y;
5        size[y] = size[x] + size[y];

FIND-SET(x)
1  if (x ≠ p[x]) p[x] = FIND-SET(p[x]);
2  return p[x];
```

Runtime analysis

Ackermann's function is defined by:

$$
\begin{array}{rcll}
A(1,j) & = & 2^j & \text{for } j \geq 1 \\
A(i,1) & = & A(i-1,2) & \text{for } i \geq 2 \\
A(i,j) & = & A(i-1, A(i,j-1)) & \text{for } i,j \geq 2
\end{array}
$$

This is an extremely-fast-growing function.

$A(i,j)$	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	2^1	2^2	2^3	2^4
$i=2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i=3$	2^{2^2}	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}16$

Introduction

Operations

Simple data structure

Advanced data structure

Final remarks

Analysis (continued)

Now define the following "inverse" function:

$$\alpha(m, n) \;=\; \min\{\, i \geq 1 \;:\; A(i, \lfloor m/n \rfloor) > \log n \,\}.$$

This is an extremely-slowly growing function.

In fact, for all practical purposes, $\alpha(m, n) \leq 4$:

$\alpha(m, n) > 4$

only if $A(4, \lfloor m/n \rfloor) \leq \log n$

only if n is larger than the number of atoms in the universe.

Theorem: Using both heuristics of union by size (or rank) and path compression the worst-case runtime for disjoint forests is

$$\mathrm{O}(m\alpha(m, n))$$

for m disjoint-set operations on n elements (i.e., m operations including n MAKE-SET operations).

Placement MAKE-SET

In order to gain access to the pointers (or iterators) into the data structure, we said that MAKE-SET(x) returns a pointer to the node containing x.

- So users of the disjoint-sets data structures don't have to care about the construction of the nodes.
- However we didn't say anything about *destruction* — this needs to be handled, either by garbage collection, or, necessary for larger graphs, directly.
- Especially for larger graphs the users knows best when and how to construct (and destruct) nodes, so a second form, namely a **placement** MAKE-SET, should be provided.
- This placement form has no return value, but a pointer to the node is already provided as input, assuming the node has already been created, and the task of MAKE-SET is then just to set the content of this node.

Counting connected components

We said that computing the connected components of a graph is a very natural application of disjoint-sets data structures:

1. The vertices of the graph are just the elements.
2. And the edges just “visualise” the calls to UNION.

One natural thing one wishes to know is how many connected components are there. This can be also asked for the disjoint-sets data structure in general — how many (disjoint) sets are there (at the time when the question is asked)?

- Currently we needed to run through all elements, compute their representatives, store them, and determine which are different. This needed run-time $O(n \log n)$.
- However, much more efficient is to keep a counter N of the number of (disjoint) sets: At the beginning $N := n$, while by each UNION-operation N is decremented by 1.

Graph generators

Since graphs provide natural benchmarks for disjoint-sets data structures (and the precise measurement of actual implementations is important!), we need to generate graphs — large graphs, fast, and in great variety.

- Perhaps the simplest generator is for *complete graphs* with n vertices, denoted by K_n, where all possible edges are present. Thus K_n has $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ many edges.
- Another simple generator is for *path graphs* with n vertices and $n - 1$ edges, denoted by P^n.

Using K_n as as benchmark amounts to running through all possible pairs of different elements for (potential) union. And using P_n amounts to considering the unions for $1, 2$, then $2, 3$, and so on, until $n - 1, n$.

Random graphs

Such generators are useful for extreme cases. But don't we have some more "random" models? There are two main models (using n as the number of vertices):

- The **binomial random graph** $\mathfrak{G}(n, p)$, where $0 \leq p \leq 1$ is the *edge probability*, has every possible edge of K_n with probability p.
  So for $p = 1$ we have K_n, while for $p = 0$ we have the graph with n vertices and no edge. The average number of edges is $p \cdot \binom{n}{2} = p \cdot \frac{n(n-1)}{2}$.
- The **uniform random graph** $\mathfrak{G}(n, m)$, where $0 \leq m \leq \binom{n}{2}$, has (exactly) m edges, and considers every selection of m edges from all possible edges as equally likely.

A useful approximation of $\mathfrak{G}(n, m)$, which is easy and fast to compute, just chooses m edges from K_n randomly and independently of each other.

Connectedness of random graphs

How large must p resp. m be, so that with high probability the graph $\mathfrak{G}(n, p)$ resp. $\mathfrak{G}(n, m)$ is connected?

(A graph is called *connected* if it has only one connected components, that is, iff from each vertex you can reach every other vertex via some edge-path.)

- The *threshold* is at an average (asymptotic) vertex degree of $\log(n)$, where the "degree" of a vertex is the number of vertices to which it is (directly) connected via an edge.
- Above that threshold with high probability the random graph is connected.
- So for example choosing $p > \frac{\log(n)}{n}$ and $m > \frac{1}{2} n \cdot \log(n)$ results with high probability in connected graphs.
- And this is asymptotically sharp.

"Above that threshold" means just very slightly above, like adding the inverse Ackermann function (however then the guarantee needs very big n for high probabilities).