Week 6

Data Structures for Disjoint Sets

1. Introduction
2. Operations
3. Application: connected components
4. Simple data structure
5. Advanced data structure
6. Final remarks
General remarks

- We consider our last example for datastructures, supporting disjoint sets.
- Again we learn how to use and how to build them.

Reading from CLRS for week 6

- Chapter 21 (not Section 21.4).
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.
The problem

Maintaining a collection

\[ S = \{ S_1, S_2, \ldots, S_k \} \]

of disjoint sets.

Each set \( S_i \) is \textit{represented} by an element \( x \in S_i \).

The collection can change over time; thus these represent \textit{dynamic} sets.

They are implemented by \textit{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 \texttt{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 \texttt{Union}-operations are performed, the return-value of \texttt{Find-Set} are stable.
- And furthermore \texttt{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 \texttt{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 \texttt{Find-Set}(x) == \texttt{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 \texttt{UNION}(x, y) and \texttt{FIND-SET}(x) are in fact not the elements themselves, but pointers (“iterators”) \textbf{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?

- \texttt{MAKE-SET}(x) still has as input an element \(x\) itself — there is no pointer to it yet.
- So actually \texttt{MAKE-SET}(x) needs to \textbf{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 \texttt{UNION}(x, y) or \texttt{FIND-SET}(x).
Review of graphs

We have already seen “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 ($\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$) 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 ($\text{FIND-SET}(u) == \text{FIND-SET}(v)$) return TRUE;
2. else return FALSE;
Connected components illustrated
Connected components via DFS

Also via BFS and DFS we can determine the connected components of a graph:

- For DFS we need in the outer loop (running through all vertices $u$ of $G$) a counter, call it $ccc$ (for “connected component counter”), initialised with 0, and incremented with every call of the recursive procedure $DFS$–$visit$.
- In that way we can count the number of connected components.
- And if we want to know for a vertex in which component it is in, then we need another array $cc$ of integers with length $|V|$ (the number of vertices), and before calling $DFS$–$visit(u)$ we set $cc[u]$ to $ccc$.

This is a linear-time algorithm (linear in the number of vertices and edges).
DFS on the example graph

Let’s consider the example, using the following order on the vertices (with induced order on the edges):

1 — 2
|
6 — 7

3 — 4 — 5

6

8

Running DFS, for each node we get the following values of discovery time, finishing time, and connected component number, together with the shown spanning forest:

(1, 8, 1) — (2, 7, 1) — (9, 14, 2) — (11, 12, 2) — (15, 16, 3)

(3, 6, 1) — (4, 5, 1) — (10, 13, 2)
Connected components via BFS

The form of BFS presented in the lecture only explores the connected component of the given start-vertex \( s \). For example using start-vertex 2 on the previous graph, we get the (rooted) BFS-tree

```
     2
    / \   \
   1   6
     /    \
    7
```

(note that this is a spanning tree (only) for the connected component of 2).

- To get all connected components, first we need to add an outer loop which runs through all vertices, and enters BFS for the vertices which haven’t been discovered yet.
- Then using a connected-component-counter and an array for storing the index of the connected component of a vertex as before, we get the same functionality (regarding connected components) as with DFS.
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.

![Diagram of cell structure](image)
Example

A linked list representation of the sets

\[ \{ a, f \}, \{ b \}, \{ g, c, e \}, \{ d \}. \]
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 \texttt{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 \texttt{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 \( \text{rep} \) 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

\[ \text{UNION}(c, b) \]

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 \texttt{MAKE-SET} operations;

and

\[ m \]: the total number of \texttt{MAKE-SET}, \texttt{UNION}, and \texttt{FIND-SET} operations.

\textbf{Note:}

1. We must have \( m \geq n \).

2. After \( n-1 \) \texttt{UNION} operations, we have only one set remaining.
A nasty example

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of objects updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-SET($x_1$)</td>
<td>1</td>
</tr>
<tr>
<td>MAKE-SET($x_2$)</td>
<td>1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>MAKE-SET($x_n$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_1$, $x_2$)</td>
<td>1</td>
</tr>
<tr>
<td>UNION($x_2$, $x_3$)</td>
<td>2</td>
</tr>
<tr>
<td>UNION($x_3$, $x_4$)</td>
<td>3</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>UNION($x_{n-1}$, $x_n$)</td>
<td>$n-1$</td>
</tr>
<tr>
<td>Total</td>
<td>$n + \frac{n(n-1)}{2} = \Theta(m^2)$</td>
</tr>
</tbody>
</table>

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

\[ \text{UNION}(x, y) \]

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 \( m \) \text{MAKE-SET}, \text{UNION} and \text{FIND-SET} operations, \( n \) of which are \text{MAKE-SET} operations, takes

\[ O(m + n \log n) \]

time.
Proof of Theorem

There are $O(m)$ Make-Set and Find-Set operations, each costing $O(1)$ time, so these contribute $O(m)$ 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 $O(n \log n)$.

Therefore the cost of a sequence of $m$ operations with $n$ Make-Set operations is $O(m + n \log n)$. □
Disjoint-set forests

**Idea:** Each set is represented by a rooted tree.

```
  i
 /  \
|   |
f  g  h
|   |
|   |
c  d e
|   |
   b
```

**Remarks:**

- The nodes of these trees 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 points to itself.
- One could have used the ordinary nil-pointer as well.
Cost of basic operations

**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:

\[ x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_{n-1} \rightarrow x_n \]

The cost of each successive \texttt{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 \textit{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

\[ \text{UNION}(x, y) \]

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

\[ \text{FIND-SET}(x) \]

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

\[ \text{UNION}(f, d) \]

\[ \text{UNION}(f, g) \]
Example of path compression

**Find-Set**(a):
Pseudocode

**MAKE-SET**(x)

1. \( p[x] = x; \)
2. \( size[x] = 1; \)

**UNION**(x, y)

1. **LINK**(\( \text{FIND-SET}(x), \text{FIND-SET}(y) \));

**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 \( (x \neq p[x]) \) \( p[x] = \text{FIND-SET}(p[x]); \)
2. return \( p[x]; \)
Runtime analysis

**Theorem:** Using both heuristics of union by size (or rank) and path compression together, the worst-case runtime for disjoint forests is for all practical purposes $O(4m)$ for $m$ disjoint-set operations on $n$ elements (i.e., $m$ operations including $n$ \texttt{Make-Set} operations).
Placement **MAKE-SET**

In order to gain access to the pointers (or iterators) into the data structure, we said that $\text{MAKE-SET}(x)$ 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** $\text{MAKE-SET}$, 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 $\text{MAKE-SET}$ is then just to set the content of this node.