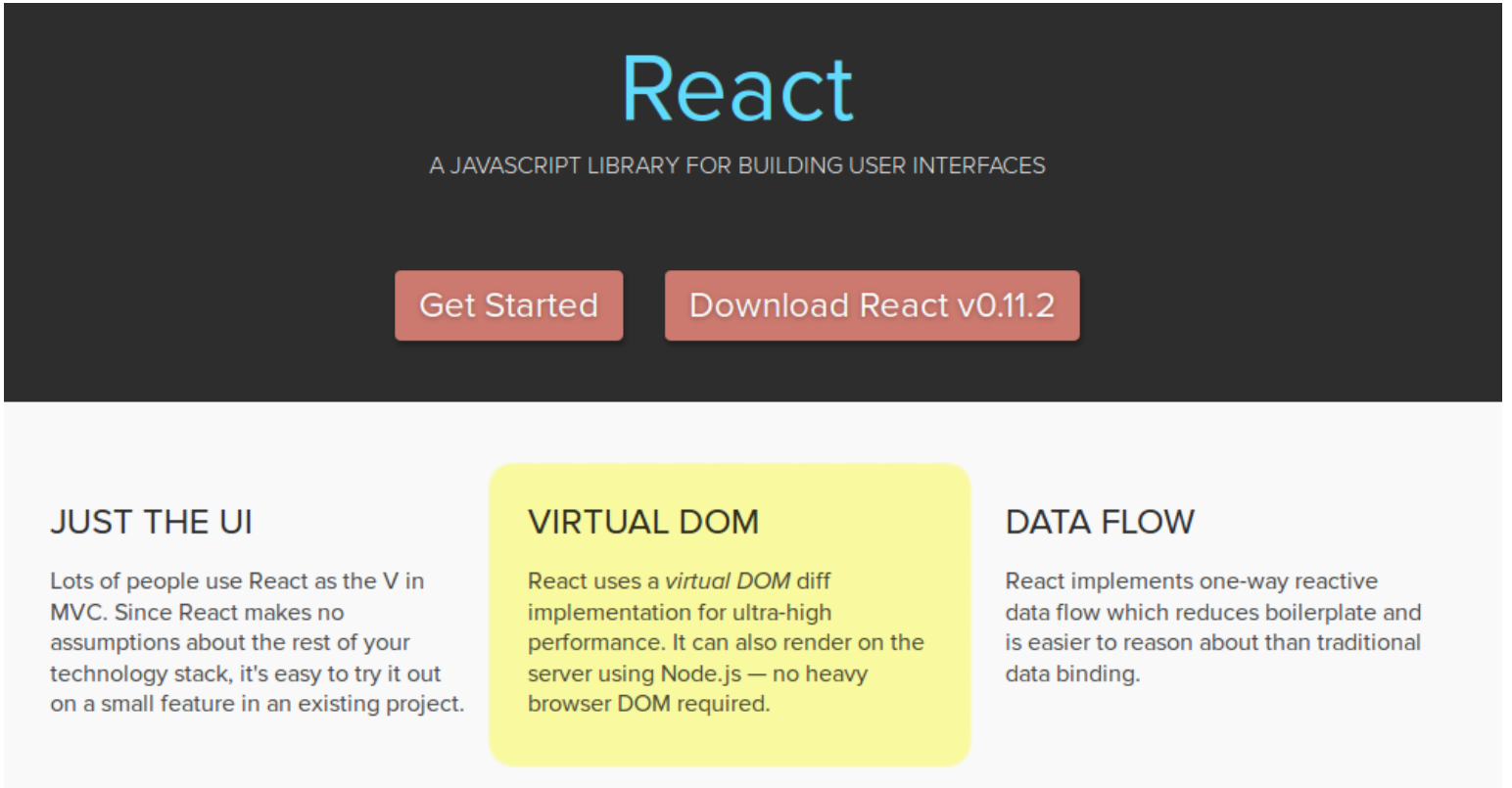
Life of a React Update

Robert Knight / Mendeley

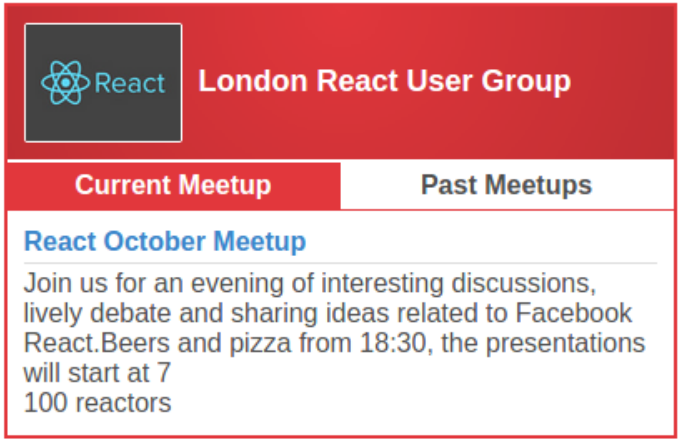


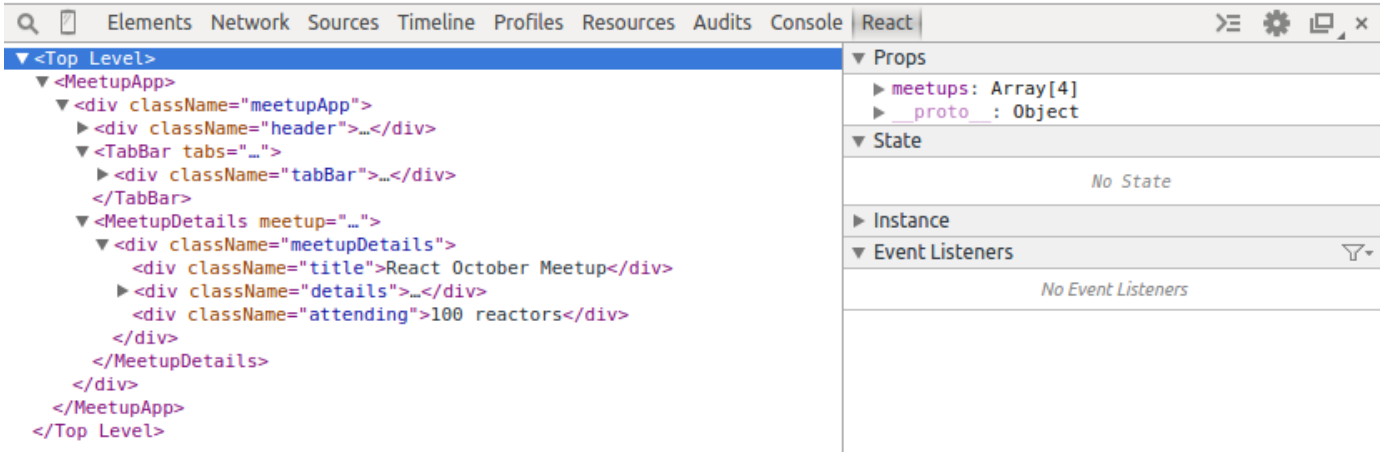
How does this part work?

Structure of a React UI

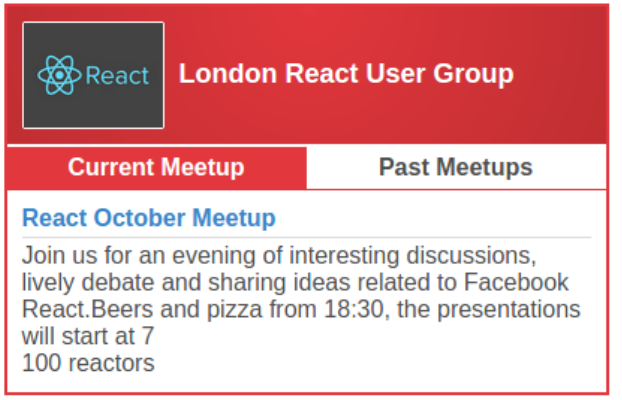
- Your app creates a tree of React components
- From which React creates a DOM tree

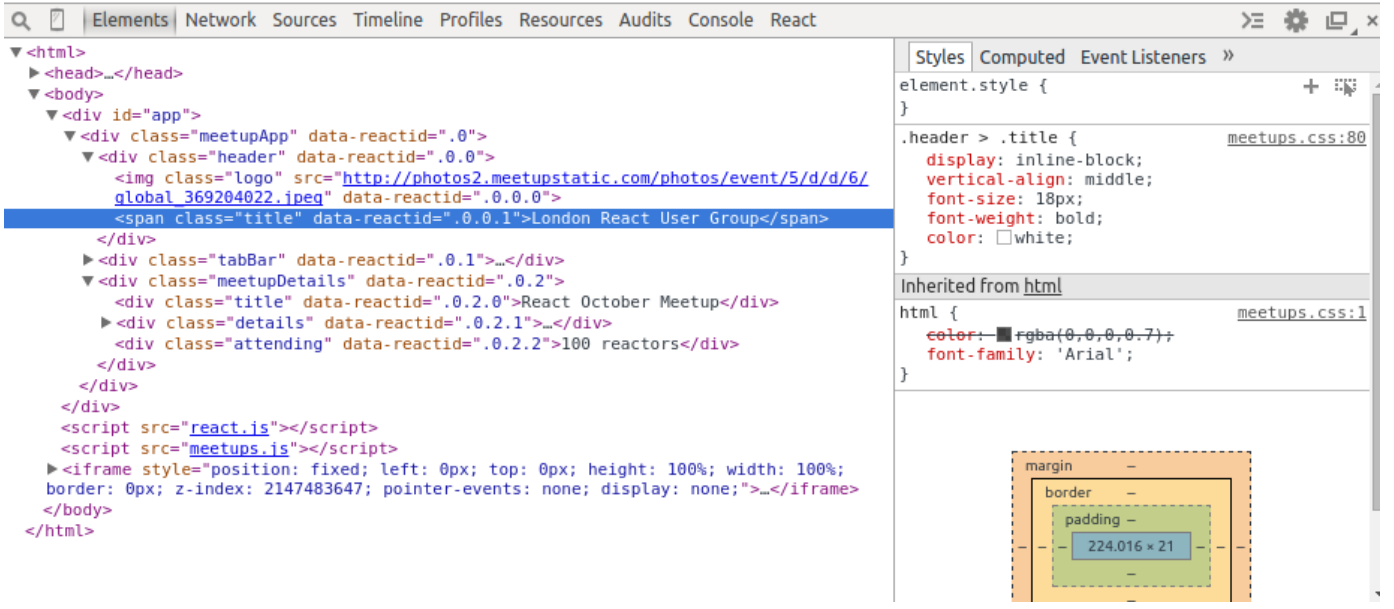
Component Tree





DOM Tree


React
London React User Group
Current Meetup
Past Meetups
React October Meetup
Join us for an evening of interesting discussions, lively debate and sharing ideas related to Facebook React.Beers and pizza from 18:30, the presentations will start at 7
100 reactors


Elements Network Sources Timeline Profiles Resources Audits Console React
<html>
<head>…</head>
<body>

<img class="logo" src="http://photos2.meetupstatic.com/photos/event/5/d/d/6/global_369204022.jpeg" data-reactid=".0.0.0">
London React User Group

…

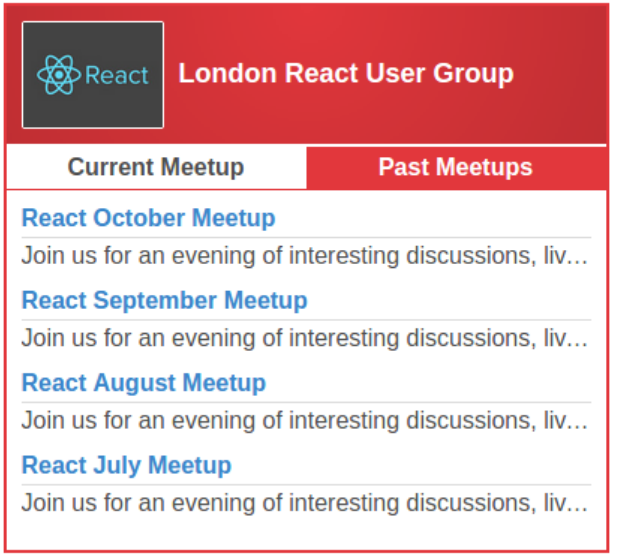
React October Meetup
…
100 reactors

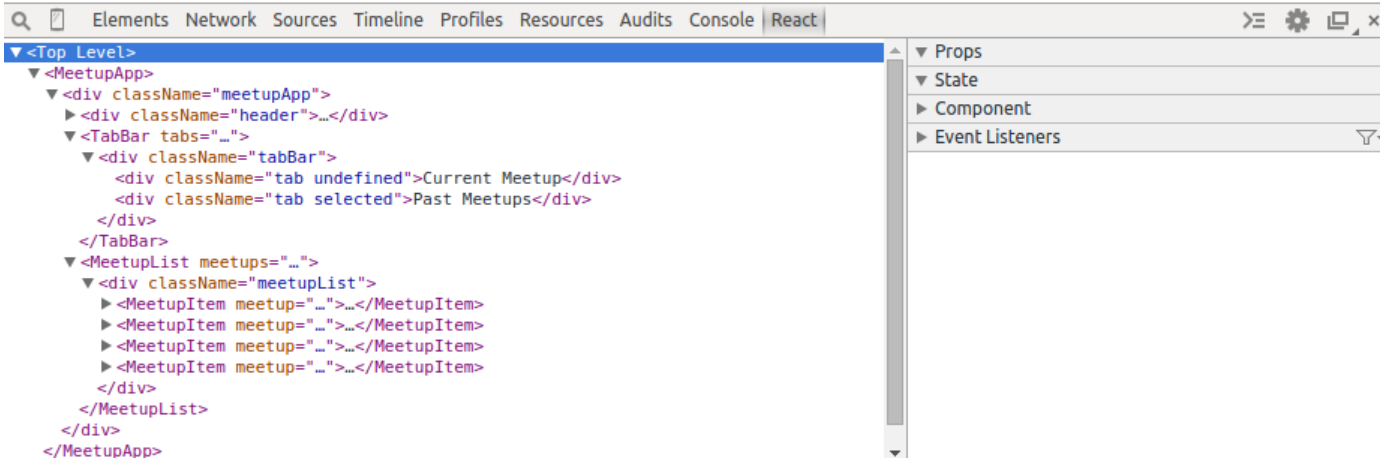
<script src="react.js"></script>
<script src="meetups.js"></script>
<iframe style="position: fixed; left: 0px; top: 0px; height: 100%; width: 100%; border: 0px; z-index: 2147483647; pointer-events: none; display: none;">…</iframe>
</body>
</html>
Styles Computed Event Listeners
element.style {
}
.header > .title { meetups.css:80
display: inline-block;
vertical-align: middle;
font-size: 18px;
font-weight: bold;
color: white;
}
Inherited from html
html { meetups.css:1
color: rgba(0,0,0,0.7);
font-family: 'Arial';
}
margin
border
padding
224.016 × 21

Reconciliation

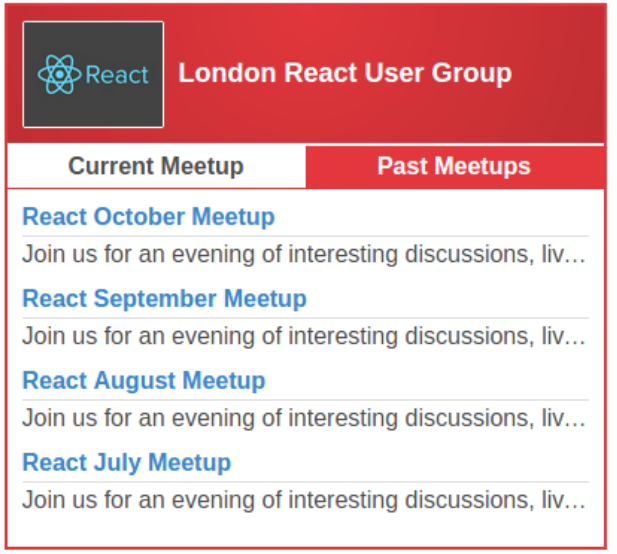
- User action or event happens. **setState()** or **setProps()** is called on one or more components
- **render()** function is called on the dirty components
- This produces a new React component tree
- Process of updating the DOM tree to match the new component tree is called **reconciliation**

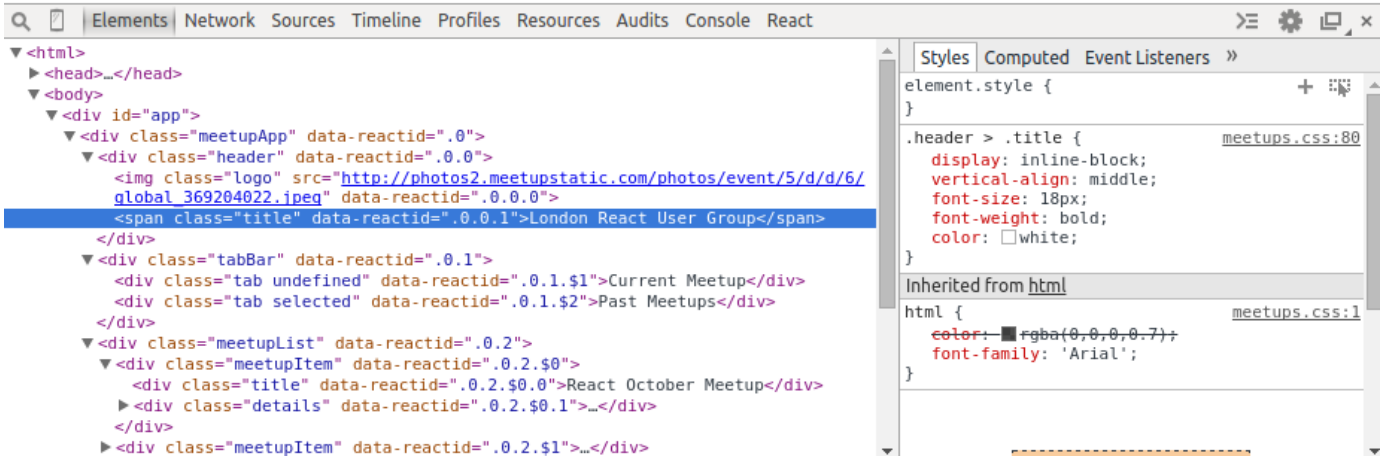
New Component Tree





New DOM Tree


React
London React User Group
Current Meetup
Past Meetups
React October Meetup
Join us for an evening of interesting discussions, liv…
React September Meetup
Join us for an evening of interesting discussions, liv…
React August Meetup
Join us for an evening of interesting discussions, liv…
React July Meetup
Join us for an evening of interesting discussions, liv…


Elements Network Sources Timeline Profiles Resources Audits Console React
<html>
<head>…</head>
<body>

<img class="logo" src="http://photos2.meetupstatic.com/photos/event/5/d/d/6/global_369204022.jpeg" data-reactid=".0.0.0">
London React User Group

Current Meetup
Past Meetups

React October Meetup
…

…
Styles Computed Event Listeners »
element.style {
}
.header > .title {
meetups.css:80
display: inline-block;
vertical-align: middle;
font-size: 18px;
font-weight: bold;
color: white;
}
Inherited from html
html {
meetups.css:1
color: rgba(0,0,0,0.7);
font-family: 'Arial';
}

How can we do this reconciliation efficiently?

1. Recreate the entire DOM

- Guaranteed to give the right result
- But slow if we have an even vaguely complex UI
- Whatever optimizations we do, we have to get the same result ***as if*** we recreated the whole DOM

2. "Diff" the old and new trees and apply changes

- Compare the old and new trees
- Produce a list of edits required to update the current DOM to match the new React component tree

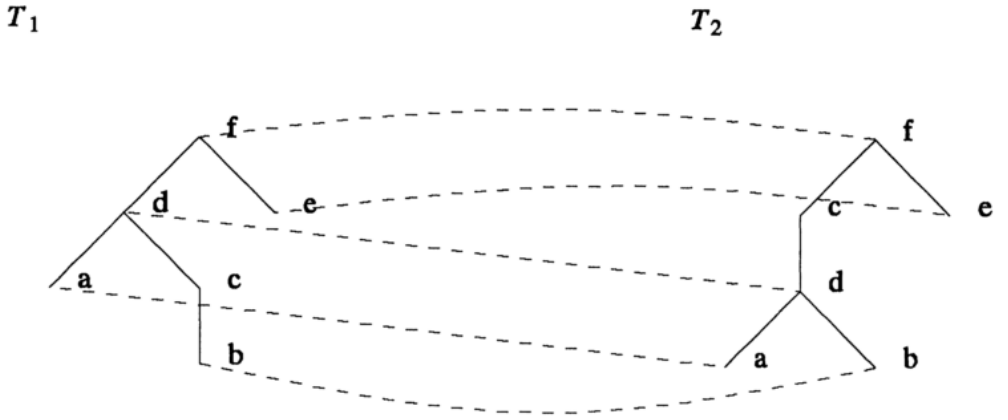


FIG. 4

Can we use a generic tree diff/patch algorithm?

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1245–1262, December 1989

SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS*

KAIZHONG ZHANG† AND DENNIS SHASHA‡

Abstract. Ordered labeled trees are trees in which the left-to-right order among siblings is significant. The distance between two ordered trees is considered to be the weighted number of edit operations (insert, delete, and modify) to transform one tree to another. The problem of approximate tree matching is also considered. Specifically, algorithms are designed to answer the following kinds of questions:

1. What is the distance between two trees?
2. What is the minimum distance between T_1 and T_2 when zero or more subtrees can be removed from T_2?
3. Let the pruning of a tree at node n mean removing all the descendants of node n. The analogous question for prunings as for subtrees is answered.

A dynamic programming algorithm is presented to solve the three questions in sequential time $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$ and space $O(|T_1| \times |T_2|)$ compared with $O(|T_1| \times |T_2| \times (depth(T_1))^2 \times (depth(T_2))^2)$ for the best previous published algorithm due to Tai [*J. Assoc. Comput. Mach.*, 26 (1979), pp. 422–433]. Further, the algorithm presented here can be parallelized to give time $O(|T_1| + |T_2|)$.

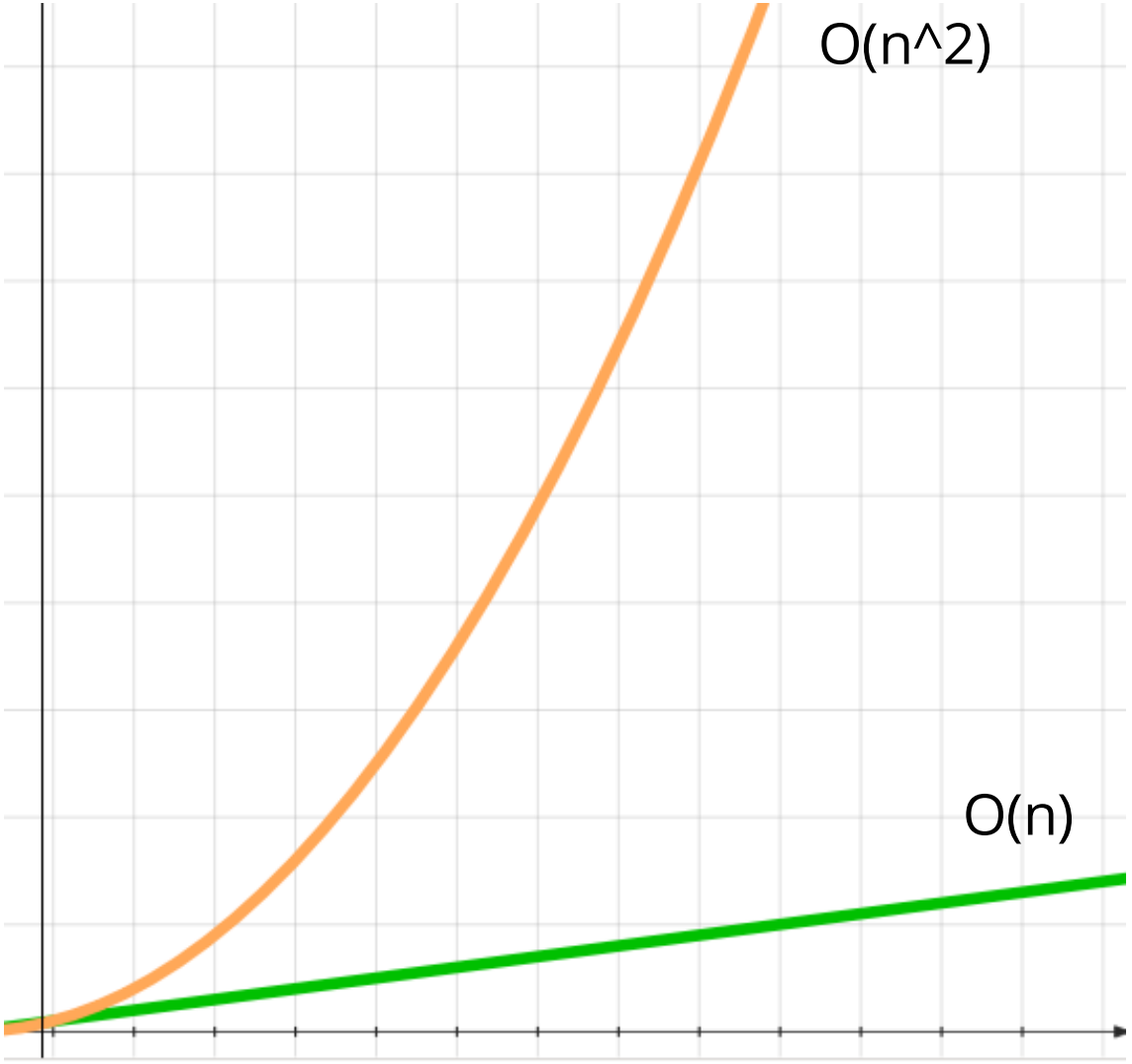
Key words. trees, editing distance, parallel algorithm, dynamic programming, pattern recognition

AMS(MOS) subject classifications. 68P05, 68Q25, 68Q20, 68R10

1. Motivation

They exist, but too slow (worse than O(n^2)) and complex

Reconciliation Time



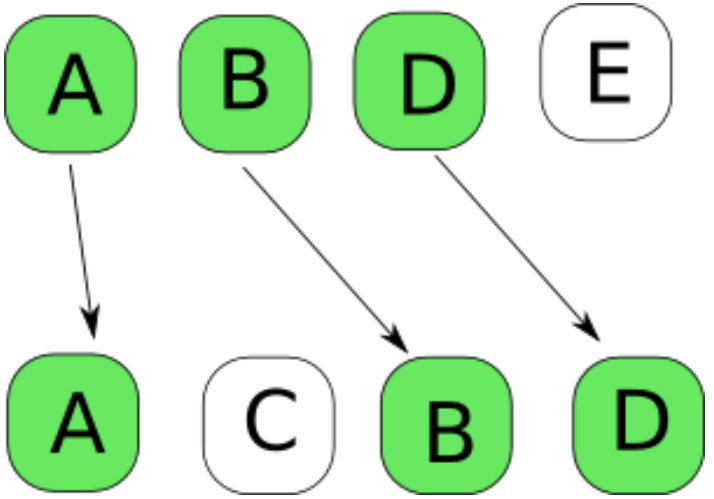
Reduce to O(n) by using heuristics

1. Hierarchical structure

- UIs have a hierarchical structure - components are unlikely to move from one part of the tree to a completely separate part
- Go down the tree from the top level-by-level, only looking for changes within each level

2. Items have unique IDs

- We've now reduced the problem to diff-ing each level of the tree.
- What is the cost of finding the changes between two lists?
- Approach used by "diff" is to find the **longest common subsequence** of the two lists. Standard algorithm is O(n^2)



- In most situations with React however, we can usually come up with unique keys for each item in a list:
  - Tweet ID
  - Bug number
  - Product code
  - Post date
- Giving every item in the list a unique **key** makes matching up items in the old and new lists much cheaper - O(n)
  - For every item in the new list - what was its position in the old list?
  - For every item in the old list - is it still in the new list?

```
// find out which items were added or moved
for (key in newChildren) {
   var newChild = newChildren[key];
   var oldChild = oldChildren[child.key];
   if (!oldChild) {
    // new element
   } else if (newChild.index != child.index) {
    // moved element
   } else {
    // nothing changed
   }
}

// find out which items were removed
for (key in oldChildren) {
   var newChild = newChildren[key];
   if (!newChild) {
    // element was removed
   }
}
```

Component Types

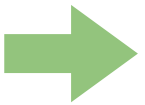
- We know we're dealing with a tree of components
- Different kinds of components probably don't have much in common
- If the component type of an item changes, React won't both looking for differences and will re-render the whole component

Reconciling a React.DOM.* component

- As we go down the tree, we eventually get to React.DOM.* components which map to the visible elements on screen
- If this is a new component, render it with setInnerHTML()
- Otherwise, we need to update the DOM node attributes and style properties to match the new props for our component
- Key improvement here is to minimize actual accesses to the DOM

- Compare current and next props of React element in JS, rather than comparing component properties to real DOM
- Only update the DOM properties if there was a change in the corresponding JS object properties

```
{
    className: 'tab'
    style: {
        width: 100,
        height: 30
    }
}
```



```
{
    className: 'tab selected'
    style: {
        width: 100,
        height: 30
    }
}
```

Additional Optimizations

Batched Updates

- A single user action may trigger changes in many components. Some later changes may obviate earlier ones.
- Idea is to collect a series of updates together in a ***batched update*** and call render() for each dirty component once at the end of the batch

Batched Update Flow

- User performs an action, setState() or setProps() called. This marks component is ***dirty*** and triggers the start of a batched update
- While a batched update is active, any dirtied components are added to a list
- At the end of the batched update, **sort the components by depth** from the root
- Update the dirty components, starting with the ones nearest the top of the tree

Batching Strategies

Which updates should be collected together in a batch?

- Too few - we might do too many DOM updates
- Too many - we might wait not update the UI often enough

React comes with two built-in strategies - a default strategy and one based on requestAnimationFrame()

- **Default -** Start batch when setState() is called
- **Request Animation Frame -** Start batch when setState() is called and invoke requestAnimationFrame(). All updates between the setState() call and the requestAnimationFrame() callback are collected in a batch.

Measuring React Performance

- React Perf tools
  - facebook.github.io/react/docs/perf.html
- General purpose DOM update monitoring - **mutation observer API**

```
Perf.start(); // start measuring

component.setState(); // update components

Perf.stop(); // stop measuring

Perf.printInclusive(); // print total time for updating components
Perf.printExclusive(); // print time excluding mounting
Perf.printWasted();    // print time spent doing work that
                       // didn't result in DOM changes
Perf.printDOM();       // print DOM update details
```

Further Reading

Blog post on diff-ing two XHTML documents:
- useless-factor.blogspot.co.uk/2008/01/matching-diffing-and-merging-xml.html

Facebook's article on React's reconciliation algorithm:
- facebook.github.io/react/docs/reconciliation.html

Article from engineer on FB's Photos teams:
- calendar.perfplanet.com/2013/diff/

Rob Knight

github.com/robertknight
@robknight_
robertknight@gmail.com