F# Tree Diff Algorithm

Implementing an algorithm to find the difference between two trees

Luke Burgess-Yeo Follow
Jan 1, 2020 · 7 min read

Photo by niko photos on Unsplash

Suppose you have two tree data structures and you want to find the difference between them. What follows is a description and implementation of a simple algorithm with takes two trees —

one original, one modified — as inputs and outputs a single tree with nodes labelled to indicate whether they have been inserted, deleted or had their values updated when compared with the original tree.

This algorithm is inspired by the diff algorithms behind Virtual n JavaScript frameworks like React. This implementation in particular is based on a post by deathmood.

An F# script file with the full implementation code and simple test case can be found here: https://github.com/LukeBurgessYeo/tree-diff.

Introduction To Trees

Let's begin with a quick introduction to tree data structures. The idea is to define a hierarchical structure containing nodes where each node has a value and a (potentially empty) collection of child nodes. We will be concerning ourselves with labeled, unordered trees.

Defining a tree

For the purposes of our algorithm we will give each node an Id — so we can match nodes between different trees — as well as a label specifying whether the node has been changed (modeled here as a discriminated union):

```
type Change =
    | Unchanged
    | Inserted
    | Deleted
    | Updated of string

type Node = {
    Id : int
    Value : string
    Modified : Change
    Children : Node list
}
```

(For simplicity I am setting the type of Value to string, but there is no reason why it could not be any other type, or in fact a generic type — as in the repo linked above.)

Instances of trees can then be constructed easily. For example, this tree with a “Root” node that has two children — “A” and “B”:

```
let tree0 = {
  Id = 0
  Value = "Root"
  Modified = Unchanged
  Children = [
    { Id = 1; Value = "A"; Modified = Unchanged; Children
= [] }
    { Id = 2; Value = "B"; Modified = Unchanged; Children
= [] }
  ]
}
```

Working with trees

Using this data structure is slightly more difficult than built in types such as Lists. In order to read data from our tree we will have to traverse the nodes using a tree traversal algorithm. The simplest to demonstrate is Depth First Search.

Depth First Search attempts to traverse the tree by visit leaf nodes (nodes with no children) first — i.e. going as deep into the tree as possible first — before going back up the tree and down the next branch. Here is an implementation in F# which we can use to print out our trees in a more human readable form:

```
let printTree tree =
  let rec traverse node level =
    let spacing =
      List.init (level * 4) (fun _ -> " ")
      |> String.concat ""
    let nodeString =
      sprintf "(%i) %s - %A" node.Id node.Value
node.Modified
    printfn "%s%s" spacing nodeString
    node.Children
    |> List.iter (fun child -> traverse child (level +
1))
  traverse tree 0
```

The above function uses a recursive inner function to traverse each node of the tree — printing out the node in a readable format — before visiting the children of that node. The traverse function also takes a state parameter “level” which keeps track of the depth of the tree so the appropriate indentation can be added to each line, making the output easier to read.

Here as the output from fsi when printTree is called on tree0:

```
> printTree tree0;;
(0) Root - Unchanged
    (1) A - Unchanged
    (2) B - Unchanged
val it : unit = ()
```

The Tree Diff Algorithm

Overview of the algorithm

The algorithm itself will have to deal with a number of different cases when comparing nodes in two trees:

1. A new node has been inserted.
2. An old node has been deleted.
3. The node exists in both trees.

If the node exists in both trees then we need to check to see if the value of the node needs to be modified. We also need to compare the children of the old node and the new node.

Implementation — Part 1: Helper functions

Firstly, let's define some helper functions to mark a node as inserted, deleted, or updated. For inserted and deleted nodes, we also need to mark the rest of the sub-tree based at these nodes as inserted or deleted. Hence it is useful to define the

Depth First Search implementation:

```
let rec markNode change node =
    { node with
        Modified = change
        Children =
            node.Children
            |> List.map (markNode change) }
```

This allows us to define the insert and delete function succinctly:

```
let insert node = markNode Inserted node

let delete node = markNode Deleted node
```

(The use of the explicit “node” parameter is required if using this code with fsi and a generic type tree.)

As the value of child nodes are not necessarily updated when a node has changed value, the update function can be written simply as a record update:

```
let update node value =
    { node with
        Modified = Updated value }
```

Where “value” indicates the new value, while the node itself will retain it’s original “Value”, so we can track the exact change.

Implementation — Part 2: Main diff function

Now we are ready to compare two nodes. As neither node is guaranteed to exist, we shall use the Option type to write a function which takes two nodes which may, or may not, exist. The type of this function will therefore be:

```
Node option * Node option -> Node
```

Using a tuple rather than currying will turn out to be useful later, but also this is not a function where partial application makes sense. Note that the function will also need to be recursive so we can call it on the children of the input nodes.

We will match on the cases of our two option inputs. The first three cases are straightforward:

1. Delete the old node if there is no new node.
2. Insert the new node if there is no old node.
3. Do nothing if the two nodes are identical.

```
let rec diff (oldNodeOpt, newNodeOpt) =
  match oldNodeOpt, newNodeOpt with
```

```
    | Some oldNode, None -> delete oldNode
    | None, Some newNode -> insert newNode
    | Some oldNode, Some newNode when oldNode = newNode -
> oldNode
```

Next we should do a simple update the value of a node has changed, but everything else is identical. To do this we can define an infix operator to do this comparison:

```
let (==) node1 node2 =
  node1.Id = node2.Id
  && node1.Children = node2.Children
  && node1.Value <> node2.Value
```

and add the following line to our diff function: (note the use of "==")

```
| Some oldNode, Some newNode when oldNode == newNode ->
  update oldNode newNode.Value
```

The final (non-degenerative) case is when we have two nodes with different children. To handle this case, we need to pass the children of the two nodes into our diff function to be compared. This means we have to construct a list of Node option tuples and iterate over the list, passing each pair back into our diff function.

The first item in each pair will contain the child of the old node (if it exists), and the second item will contain the child of the new node (if it exists). The determine whether a given node should exist, we must iterate first over the children of the old node, checking to see if that node exists in the list of children of the new node. Then do similarly for the new node, checking the list of children of the old node. The following function does exactly that:

```
let combine node1 node2 =
  let first =
    node1.Children
    |> List.map (fun x ->
      let maybeNode =
        node2.Children
        |> List.tryFind (fun y -> y.Id = x.Id)
      (Some x, maybeNode))

  let second =
    let maybeNode =
      node2.Children
      |> List.map (fun x ->
        (node1.Children
        |> List.tryFind (fun y -> y.Id = x.Id)
    (maybeNode, Some x))

  first @ second |> List.distinct
```

Note the use of List.distinct at the end to remove any double counted nodes.

We are now in a position to complete our diff function by using List.map to pass each child node combination back into our diff function to get the list of compared child nodes. This is the reason why we chose to pass a tuple to diff. Here is the completed function, with the newly added case in **bold**:

```
let rec diff (oldNodeOpt, newNodeOpt) =
  match oldNodeOpt, newNodeOpt with
   | Some oldNode, None -> delete oldNode
   | None, Some newNode -> insert newNode
   | Some oldNode, Some newNode when oldNode = newNode ->
oldNode
   | Some oldNode, Some newNode when oldNode == newNode ->
     update oldNode newNode.Value
   | Some oldNode, Some newNode ->
     { oldNode with
         Modified =
           if oldNode.Value = newNode.Value then
             Unchanged
           else Updated newNode.Value
         Children =
           combine oldNode newNode
           |> List.map diff }
   | None, None ->
     failwith "Must have at least one node to compare."
```

The final match case is “None, None” which does not make sense as an input to the function and so we raise an exception. This function should never receive two None’s as inputs. Even if it did, we would not have a node to return and creating an empty node to continue the execution of the program could cause problems later on.

Finally we can wrap our diff function in a more intuitive API for external use:

```
let compare oldTree newTree = diff (Some oldTree, Some
        newTree)
```

Final Comments

This algorithm can likely be further optimized and improved upon — for example respecting the order of nodes as well as their Id's.

The careful reader may have noticed that the match case which deals with children of the given nodes also deals with the previous two cases (when the nodes are equal, and when they are equal in all but value). The inclusion of these two cases improves the performance of the algorithm for large trees, and trees with very few changes. We can quickly eliminate identical sub-trees by comparing nodes by value (as F# does by default), and the case where only the value has changed gives a practical example of defining a custom infix operator.

Of course, if the user is comparing extremely large trees then other optimizations may be necessary, and F# may not even be the most suitable tool for the task. Other algorithms exist which try to find the "minimum update script" — a list of edits to transform one tree into another as efficiently as possible (where you can decide how to measure efficiency).

Finally, please feel free to contribute any bug fixes or improvements to the code at https://github.com/LukeBurgessYeo/tree-diff.

Programming Fsharp Trees Data Structures Data