Applied Algorithms Final Project

Drew Hoskins

March 12, 2007

1 Overview

This paper deals with the ordered tree edit-distance problem. Imagine you have two trees T_1 and T_2, whose sets of siblings are ordered from left to right, and you wish to record the minimum sequence of steps required to get from T_1 to T_2. Each transformation has an associated cost. We will write $n_2 = |T_2|$ and $n_1 = |T_1|$, the numbers of nodes in each tree.

I'll start by mentioning the applied circumstance where this algorithm came up, then discuss the survey I read of known dynamic programming algorithms. Then I'll present my interpretation of what I learned of the voting algorithm that was used by the team I joined, and see how it compares in generality, correctness, and complexity.

2 Application

This problem came up an interview question for the group I just joined. We have an ink-based text document, which is characterized by a tree whose leaves are ink strokes. Strokes are nodes grouped as children of word nodes, which are grouped as children of line nodes, which are grouped as children of paragraph nodes, which are grouped as children of region nodes.

Thus, a block of text is represented as an ordered tree, and you may want to know the most efficient sequence of transformations to get you between two trees, such as for undo and redo operations.

3 Dynamic programming approach

When you think of edit distance, the natural approach to take is a dynamic programming approach. Subproblems will consist of forests whose roots start at various places in the tree and descend to the leaves. Imagine a cost function c which measures the cost of the three types of edits: insertion of nodes, unlinking of nodes, and relabeling of nodes. Let ED be a function which computes edit distance. As described in [1], we can write a recursion on forests F_1 and F_2 as follows: ED computes the minimum of whichever of these three options is possible for a given pair of trees (restricted by which forests are null in degenerate cases), or 0 if F_1 and F_2 are empty.

Deleting v from F_1:

$$ED(F_1, F_2) = ED(F_1 - \{v\}, F_2) + c(\texttt{unlink}\ v)$$

Adding w to F_2:

$$ED(F_1, F_2) = ED(F_1, F_2 - \{w\}) + c(\texttt{insert}\ w)$$

Or replacing v with w:

$$ED(F_1, F_2) = ED(F_1(v), F_2(w)) + ED(F_1 - T_1(v), F_2 - T_2(w)) + c(\texttt{relabel}\ v, w)$$

The notation $T(u)$ is a subtree rooted at u, and $F(u) = T(u) - \{u\}$. So in the third case, we essentially swap w's subtree in for v's.

To restrict the number of subproblems that are considered, u and v are chosen in a systematic way; in the paper, they choose the rightmost roots of F_1 and F_2 to include only forests with "prefixes" and "suffixes" excluded, meaning each tree has $|T|^2$ subproblems, for a complexity of $O(|T_1|^2|T_2|^2)$.

Getting better complexity results out of the dynamic programming technique is all about cleverly reducing the subproblems that need to be considered, and the paper mentions ideas which reduce the complexity. One technique gets it down to $O(|T_1|^2|T_2|\log|T_2|)$ by pivoting subproblems around "heavy paths" so as to get more bang for one's buck when choosing subproblems.

4 Voting approach

The voting approach that we worked out at my interview and is apparently in our code base today (I haven't seen the code) is a very different approach, so it will be interesting to compare its correctness and complexity to the dynamic programming approaches, as well as to see if there are non-obvious parallels between the two approaches. This approach will assume a constant cost for insertions and deletions, and I'll discuss the cost of relabeling later.

The algorithm has three passes.

4.1 Voting Pass

In this pass, we iterate through the nodes in T_2, and each "votes" for its parent in T_2; those votes will be used to determine which node in T_1 will become this node in T_2. Each time a node votes for a parent, it is also appended to the end of one of a set of subsequences of nodes who had the same parent in T_1. In this way, for each non-leaf node in T_2, we'll have a partition of nodes into subsequences that will be able to survive intact if they are transfered to the new parent. Each insertion thus looks through a list of nodes from T_2, so this pass takes $O(n_2^2)$.

You can imagine that we could use these subsequence partitions to somehow form pairs of nodes $(u, v) \in T_1 \times T_2$ that designate a relabeling mapping, and to assign each pairing a benefit based on the length of the subsequence from u. See below.

4.2 Node Mapping Pass

In this pass, we determine which node u from T_1 will be mapped to node v from T_2.

For example, say the node labeled A has children B, C, D in T_1, but node E owns B, C, D in T_2. Rather than moving each of B, C, D individually from A to E, we can simply relabel A to be E, thus the parent who referenced A will now reference E.

We ignore, for now, the cost of doing these relabelings, and pretend like it only costs to do insertions and deletions. We can compute that cost for any choice of relabeling. Let $S_2(v)$ be the sequence of children of v in T_2, and for each node $u \in T_1$ let $S_{12}(u, v)$ be the longest sequence of children of u that is common with $S_2(v)$. The cost of populating u having that subsequence already in place is

$$(|S_2(v)| - |S_{12}(u, v)|) + (|S_1(u) - S_{12}(u, v)|) \tag{1}$$

The first term is the number of insertions that are needed, and the second term is the number of unlinks.

However, a choice which minimizes (1) can have an adverse effect on other node mappings, as it will restrict the options available. We really need to find a mapping of nodes in T_1 to nodes in T_2 that minimizes the sum of these expressions. I suspect the best way to solve this problem is dynamic programming! A memoized subproblem would be parameterized by those nodes that have already been mapped, and its result would consist of optimizing the rest of the mappings. Not surprisingly, it seems we reduced the problem back to where the literature started us. But this problem should be at least as simple as the general problem, since we only are considering remappings, rather than insertions and deletions. It's possible that it could be done more quickly than the more general problem..

But a very natural linear greedy algorithm would be to start at some (possibly cleverly chosen) node v, and we simply pick u from T_1 to be relabeled to v which minimizes (1). Fix that mapping, and go onto the next one. There would be n_2 nodes v, and for each v, we may have to look through `degree` v subsequences, assuming n_1 is big enough. So this algorithm would take $O(n_2 \min(n_1, \max \texttt{degree}(T)))$ time.

An $n \log n$ greedy algorithm would sort the nodes v by the ones that could exhibit the most gain relative to a baseline of not relabeling any nodes, and proceed in that order.

As for the handwaving that went with costing relabeling as 0, I left out a cost for that operation to simplify (1). In addition to insertions and deletions, we must cost relabeling the node and doing something with the node it replaced. But these operations have a constant maximum cost which can be factored into the expression. In other words, we can fix (at least the greedy) algorithm by giving a constant negative bias to not relabeling a node. For a guaranteed-optimal agorithm we need to calculate the cost exactly on a case-by-case basis.

4.3 Insertion and Unlink Pass

Once the nodes have been relabeled, go through each node v in T_1, and perform insertions for nodes that voted for it, and unlinks for nodes that are its children that didn't vote for it. The result of this process will be T_2. This algorithm has complexity $O(n_1 + n_2)$ since we may have at most n_1 unlinks and n_2 insertions, and given the right data structure, such operations can occur in constant time.

4.4 Complexity

The complexity of Pass 1 was $O(n_2^2)$ and the complexity of Pass 3 was $O(n_1+n_2)$. With the simplest greedy algorithm for Pass 2, with complexity $O(n_2 \min(n_1, \max \mathtt{degree}(T)))$, the total complexity is $O(n_1 + n_2^2)$. With the full fledged solution, Pass 2 will be the gating pass for sure, but I did not investigate the complexity. I think it's fair to assume that it will be at worst $O(n_1^2 n_2 \log n_2)$, the complexity of the best algorithm presented by [1].

4.5 Conclusions and Open Issues

Voting was a reconceptualization of the problem that yielded some potentially very interesting greedy approximation algorithms that make use of an intuition that the standard dynamic programming solutions lacked: that we can somehow measure at a local level the effect that a relabeling choice will have on the edit distance. However, that measurement is not independent of other choices, so it cannot be used in a correctness proof of such an algorithm.

In a word-processor like application with trees of ink strokes, it seems likely that a greedy approach is warranted, given that these operations must be completed with low latency, and that the transformation steps involved in editing T_1 to become T_2 aren't visible to the user, and thus need not be precisely optimal.

It would be neat if I had time to formalize the dynamic programming problem that arose, as well as determine how close of an approximation greedy approaches to voting are to optimal. I'd also like to investigate available papers more thoroughly to determine if voting approaches have been tried in the literature.

Finally, it seems like the voting technique could be modified to account for more arbitrary cost metrics, though I haven't put thought into it.

5 References

[1] Philip Bille. *Tree Edit Distance, Alignment Distance and Inclusion.*, pages 1-9, 2003.