ZHANG, K., R. STATMAN AND D. SHASHA [1992]. "On the editing distance between unordered labeled trees" *Information Processing Letters* **42**, pp.133-139.

JIANG, T., L. WANG AND K. ZHANG [1994]. "Alignment of trees - an alternative to tree edit", *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching*, pp. 75-86.

KILPELAINEN, P. AND H. MANNILA [1991]. "Ordered and unordered tree inclusion", To appear *SIAM J. on Computing.*

KOSARAJU, S.R. [1992]. "Efficient tree pattern matching", *Proceedings of the 30th annual IEEE Symposium on Foundations of Computer Science*, pp. 178-183.

LANDAU, G.M. AND U. VISHKIN [1989]. "Fast parallel and serial approximate string matching", *J. Algorithms*, **10**, pp.157-169.

LU, S.Y. [1979]. "A tree-to-tree distance and its application to cluster analysis", *IEEE Trans. PAMI*, **1**, pp.219-224.

SELKOW, S.M. [1977]. "The tree-to-tree editing problem", *Information Processing Letters*, **6**, pp.184-186.

SHAPIRO B.A. AND K. ZHANG [1990]. "Comparing multiple RNA secondary structures using tree comparisons", *Comput. Appl. Biosci.* **6**, (4), pp.309-318.

SHASHA, D., J.T.L. WANG AND K. ZHANG [1994]. "Exact and approximate algorithms for unordered tree matching", *IEEE Trans. Systems, Man, and Cybernetics*, **24**, (4), pp.668-678.

SHASHA, D. AND K. ZHANG [1990]. "Fast algorithms for the unit cost editing distance between trees", *J. Algorithms*, **11**, pp.581-621.

TAI, K.C. [1979]. "The tree-to-tree correction problem", *J. ACM*, **26**, pp.422-433.

TANAKA, E. AND K. TANAKA [1988]. "The tree-to-tree editing problem", *International Journal of Pattern Recognition and Artificial Intelligence*, **2**, (2), pp.221-240.

THORUP, M. [1994]. "Efficient Preprocessing of Simple Binary Pattern Forests", *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science*, **824**, pp. 350-358.

UKKONEN, E. [1985]. "Finding approximate patterns in strings", *J. Algorithm*, **6**, pp.132-137.

WANG, J.T.L., K. ZHANG, K. JEONG AND D. SHASHA [1994]. "A system for approximate tree matching", *IEEE Trans. Knowledge and Data Engineering*, **6**, (4), pp.559-571.

YANG, W. [1991]. "Identifying syntactic differences between two programs", *Software – Practice and Experience*, **21**, (7), pp.739-755.

ZHANG, K. [1994]. "A constrained editing distance between unordered labeled trees", To appear *Algorithmica.*

ZHANG, K. AND T. JIANG [1994]. "Some MAX SNP-hard results concerning unordered labeled trees", *Information Processing Letters*, **49**, pp.249-254.

ZHANG, K. AND D. SHASHA [1989]. "Simple fast algorithms for the editing distance between trees and related problems", *SIAM J. Computing* **18**, (6), pp.1245-1262.

ZHANG, K., D. SHASHA, AND J.T.L. WANG [1994]. "Approximate tree matching in the presence of variable length don't cares", *J. of Algorithms*, **16**, pp.33-66.

8. Suppose we have a pair of ordered rootless trees T_1 and T_2. Define the edit distance between those two trees to be the edit distance between the rooted trees R_1 and R_2 where R_1 is isomorphic to T_1 and R_2 is isomorphic to T_2 and the distance between R_1 and R_2 is the minimum edit distance of any pairs of rooted trees R_1' and R_2' where R_1' is isomorphic to T_1 and R_2' is isomorphic to T_2. Hint: Use the algorithm you developed for the previous question as a subroutine.

14.7 Bibliographic notes

The first attempt to generalize string edit distance to ordered trees was due to Selkow [1977]. He gave an tree edit algorithm in which the insertions and deletions are restricted to the leaves of the trees. The edit distance between ordered trees was introduced by Tai [1979]. Another edit base distance was introduced by Lu [1979]. Lu treated each subtree as a whole entity and did not allow one subtree to match more than one subtrees in the other tree. Tanaka and Tanaka [1988] introduced the strongly structure preserving mapping and gave an algorithm based on this kind mapping. Their algorithm is the same as Lu's algorithm. Yang [1991] gave an algorithm based on a mapping where two nodes in the mapping implies their parents are in the mapping. Edit distance between unordered tree was considered by Zhang, Statman and Shasha [1992]. Jiang, Wang and Zhang [1994] considered the tree alignment distance problem. Tree inclusion problem was introduced by Kilpelainen and Mannila.

The algorithm for edit distance presented in this chapter is due to Zhang and Shasha. The alignment distance algorithm is due to Jiang, Wang and Zhang. It is open whether the time complexity of these algorithm can be improved. There is no non-trivial lower bound result for these problems.

The parallel algorithm for unit cost edit distance discussed in this chapter is due to Shasha and Zhang. It is open whether the restriction of unit cost can be removed or not.

The approximate tree match was considered by Zhang and Shasha. This was later extended to handle the case where pattern tree can have variable length don't cares. The algorithm presented is due to Zhang, Shasha and Wang.

The NP-completeness results for edit distance between unordered trees is due to Zhang, Statman and Shasha. The MAX SNP-hard result is due to Zhang and Jiang. It is open whether these problems can be approximated within a constant.

14.8 References

Cai, J., R. Paige and R. Tarjan [1992]. "More efficient bottom-up multi-pattern matching in trees", *Theoretical Computer Science*, **106**, pp. 21-60.

Chase, D. [1987]. "An improvement to bottom-up tree pattern matching", *Proceedings of the 14th Annual CM Symposium on Principles of Programming Languages*, pp. 168-177.

Dubiner, M., Z. Galil and E. Magen [1994]. "Faster tree pattern matching", *JACM*, **14**, (2), pp. 205-213.

Hoffman, C. and J. O'Donnel [1982]. "Pattern matching in trees", *JACM*, **29**, (1), pp. 68-95.

appears possible — certainly not for unordered trees (because of the NP-completeness result) and we conjecture not for ordered trees.

Besides pursuing new applications and letting them lead us to new algorithms, we are currently working on the problem of tree pattern discovery. The philosophy of this work is best shown by distinction to the work we have described so far. Our work to date has consisted primarily of finding the distance between a given pattern tree and a given data tree given a pattern metric. By contrast, tree pattern discovery consists of *producing* a pattern tree that, according to a given distance metric, is close to a collection of data trees. Such tree "motifs" could characterize a collection of trees representing some phenomenon in nature. We are currently working on the secondary structure of viruses.

14.6 Exercises

1. A counter-diagonal in a dynamic programming matrix extends from location (i,0) to (0,i). For trees as well as for strings, all elements in a counter-diagonal can be computed in parallel. Design a parallel version of Algorithm EDIT? Hint: The complexity should be $O(|T_1| + |T_2|)$.

2. Suppose you were only interested in the editing distance between two trees assuming they differed by no more than d. That is, your algorithm would return the exact distance if it is less than or equal to d, but would return "very different" otherwise. What would the time complexity be in that case? Hint: The complexity should be proportional to the square of d.

3. Pruning a tree at node n means removing all its children, but not removing n itself. Define the optimal pruning distance between a pattern tree P and a data tree T to be the minimum distance between P and tree T' where T' is T followed by pruning. Hint: The algorithm should be a variant of the algorithm with cuts.

4. Consider the Procedure *tree_vldc* in which eliminating a path was free if it is matched to a variable length don't care. Consider a metric in which all inserts, deletes and replacements cost one and in which deletions of paths in the text tree along with their subtrees also had unit cost. Design an algorithm to compute that cost.

5. Prove Lemma 14.4.1 to Lemma 14.4.1.

6. Try to show the MAX SNP-hardness result mentioned in section 14.4 by reduction from Maximum Bounded Covering by 3-sets.

7. Geographical data structures such as quadtrees are not rotation-invariant, but suppose we wanted to find the editing distance between two trees where we allow rotations among the children of the roots. That is, given two ordered rooted trees T_1 and T_2, find the distance between R_1 and R_2, where R_1 is T_1 but perhaps with a rotation among the children of the root of T_1; R_2 is T_2 but perhaps with a rotation among the children of the root of T_2, such that the distance is minimum among all such rotations of T_1 and T_2. Hint: If the degree of the root of the trees is no greater than the depth of the trees, then the complexity should be no greater than Algorithm EDIT.

Algorithm: ALIGN(T_1,T_2)

begin
for $i := 1$ **to** $|T_1|$
for $j := 1$ **to** $|T_2|$

$$
\begin{aligned}
A(F_1[i], F_2[j]) := \min\{ \quad & \gamma(t_1[i_2], \lambda) + A(F_1[i_2], F_2[j]) + A(T_1[i_1], \theta), \\
& \gamma(t_1[i_1], \lambda) + A(F_1[i_1], F_2[j]) + A(T_1[i_2], \theta), \\
& \gamma(\lambda, t_2[j_2]) + A(F_1[i], F_2[j_2]) + A(\theta, T_2[j_1]), \\
& \gamma(\lambda, t_2[j_1]) + A(F_1[i], F_2[j_1]) + A(\theta, T_2[j_2]), \\
& A(T_1[i_1], T_2[j_1]) + A(T_1[i_2], T_2[j_2]), \\
& A(T_1[i_1], T_2[j_2]) + A(T_1[i_2], T_2[j_1]) \ \};
\end{aligned}
$$

$$
\begin{aligned}
A(T_1[i], T_2[j]) := \min\{ \quad & \gamma(t_1[i], t_2[j]) + A(F_1[i], F_2[j]), \\
& \gamma(t_1[i], \lambda) + A(T_1[i_1], T_2[j]) + A(T_1[i_2], \theta), \\
& \gamma(t_1[i], \lambda) + A(T_1[i_2], T_2[j]) + A(T_1[i_1], \theta), \\
& \gamma(\lambda, t_2[j]) + A(T_1[i], T_2[j_1]) + A(\theta, T_2[j_2]), \\
& \gamma(\lambda, t_2[j]) + A(T_1[i], T_2[j_2]) + A(\theta, T_2[j_1]) \ \};
\end{aligned}
$$

end

Output: $A(T_1[|T_1|], T_2[|T_2|])$.

Figure 14.12: Aligning unordered binary trees.

14.5 Conclusion

As we have discovered since making our tree comparison software generally available,[3] many applications require the comparison of trees. In biology, RNA secondary structures are topological characterizations of the folding of a single strand of nucleotides. Determining the functionality of these structures depends on the topology and therefore comparing different ones based on their topology is of interest. In neuroanatomy, networks of connections starting at a single point often describe trees. Comparing them may give a hint as to structure. In genealogy, unordered trees are of interest and may give hints about the origins of hereditary diseases. In language applications, comparing parse trees can be of interest. Finally, we are currently developing a package to enable users to compare structured documents based on tree edit distance. This should be more informative than utilities such as UNIX diff.

Algorithmically, tree comparison bears much similarity to string comparison. Tree comparison uses dynamic programming, suffix trees, and, in parallel versions, counter-diagonals. We often reason by analogy to stringologic work when developing new algorithms. For this reason, we believe that treeology may be a good discipline to study for talented stringologists who are tired of one dimensional structures. But the tree and string problems are different and no reduction

[3]Send us email if you're interested.

Lemma 11 *Let M be a mapping between T_1 and T_2. If there are $d \geq 0$ nodes of T_2 not in mapping M, then $\gamma(M) \geq 3n - 2k + d$.*

Lemma 12 *$treedist(T_1, T_2) \geq 3n - 2k$.*

Lemma 13 *If there is an exact 3-cover, then $treedist(T_1, T_2) = 3(n-k) + k = 3n - 2k$.*

Lemma 14 *If $treedist(T_1, T_2) = 3n - 2k$, then there exists an exact 3-cover.*

In fact there are stronger results concerning the hardness of computing the edit distance and alignment distance for unordered labeled trees. Computing the edit distance between two unordered labeled trees is MAX SNP-hard even if the two trees are binary. Computing the alignment distance between two unordered labeled trees is MAX SNP-hard when at least one of the trees is allowed to have an arbitrary degree. The techniques for these proofs are similar to the one we just presented. The reduction is from Maximum Bounded Covering by 3-sets which is an optimization version of Exact Cover by 3-sets.

14.4.2 Algorithm for tree alignment and a heuristic algorithm for tree edit

When the degrees are bounded, we can compute the alignment distance using a modified version of Algorithm ALIGN. Lemmas 8 and 9 still work. The only difference is in the computation of $D(F_1[i], F_2[j])$. We have to revise the recurrence relation in Lemma 10 as follows: for each (forest) $C \subseteq \{T_1[i_1], \ldots, T_1[i_{m_i}]\}$ and each (forest) $D \subseteq \{T_2[j_1], \ldots, T_2[j_{n_j}]\}$,

$$A(C,D) = \min \begin{cases} \min_{T_1[i_p] \in C, T_2[j_q] \in D} A(C - \{T_1[i_p]\}, D - \{T_2[j_q]\}) + A(T_1[i_p], T_2[j_q]), \\ \min_{T_1[i_p] \in C, D' \subseteq D} A(C - \{T_1[i_p]\}, D - D') + A(F_1[i_p], D') + \gamma(t_1[i_p], \lambda), \\ \min_{C' \subseteq C, T_2[j_q] \in D} A(C - C', D - \{T_2[j_q]\}) + A(C', F_2[j_q]) + \gamma(\lambda, t_2[j_q]) \end{cases}$$

Since m_i and n_j are bounded, $A(C,D)$ can be computed in polynomial time. If T_1 and T_2 are both in fact binary trees, the algorithm can be much simplified, as shown in Figure 14.12 It is easy to see that the time complexity of this algorithm is $O(|T_1| \cdot |T_2|)$.

For the edit distance, we have an efficient enumerative algorithm. The algorithm runs in polynomial time when one of the trees has a bounded number of leaves.

For the more general cases, we have developed heuristic algorithms based on probabilistic hill-climbing. Another way to deal with the hardness results is to add more constraints on the way in which we transform one tree to the other. This leads to a constrained edit distance between two unordered trees. The complexity of this algorithm is $O(|T_1| \times |T_2| \times (deg(T_1) + deg(T_2)) \times \log_2(deg(T_1) + deg(T_2)))$.

such applications, it would be useful to compare unordered labeled trees by some meaningful distance metric. The editing distance metric, used with some success for ordered labeled trees, is a natural such metric. The alignment distance is another metric. This section presents algorithms and complexity results for these metrics.

14.4.1 Hardness results

We reduce Exact Cover by 3-Sets to the problem of computing the edit distance between unordered labeled trees. This means that computing the edit distance between unordered labeled trees is NP-hard. We assume that each edit operation has unit cost, i.e. $\gamma(a \to b) = 1$ if $a \neq b$.

Exact Cover by 3-Sets

INSTANCE: A finite set S with $|S| = 3k$ and a collection T of 3-element subsets of S.

QUESTION: Does T contain an exact cover of S, that is, a subcollection $T' \subset T$ such that every element of S occurs in exactly one member of T'?

Given $S = \{s_1, s_2, ...s_m\}$, where $m = 3k$ and $T = T_1, T_2, ...T_n$ where $T_i = \{t_{i1}, t_{i2}, t_{i3}\}$, $t_{ij} \in S$, we construct the two trees as in Figure 14.11.

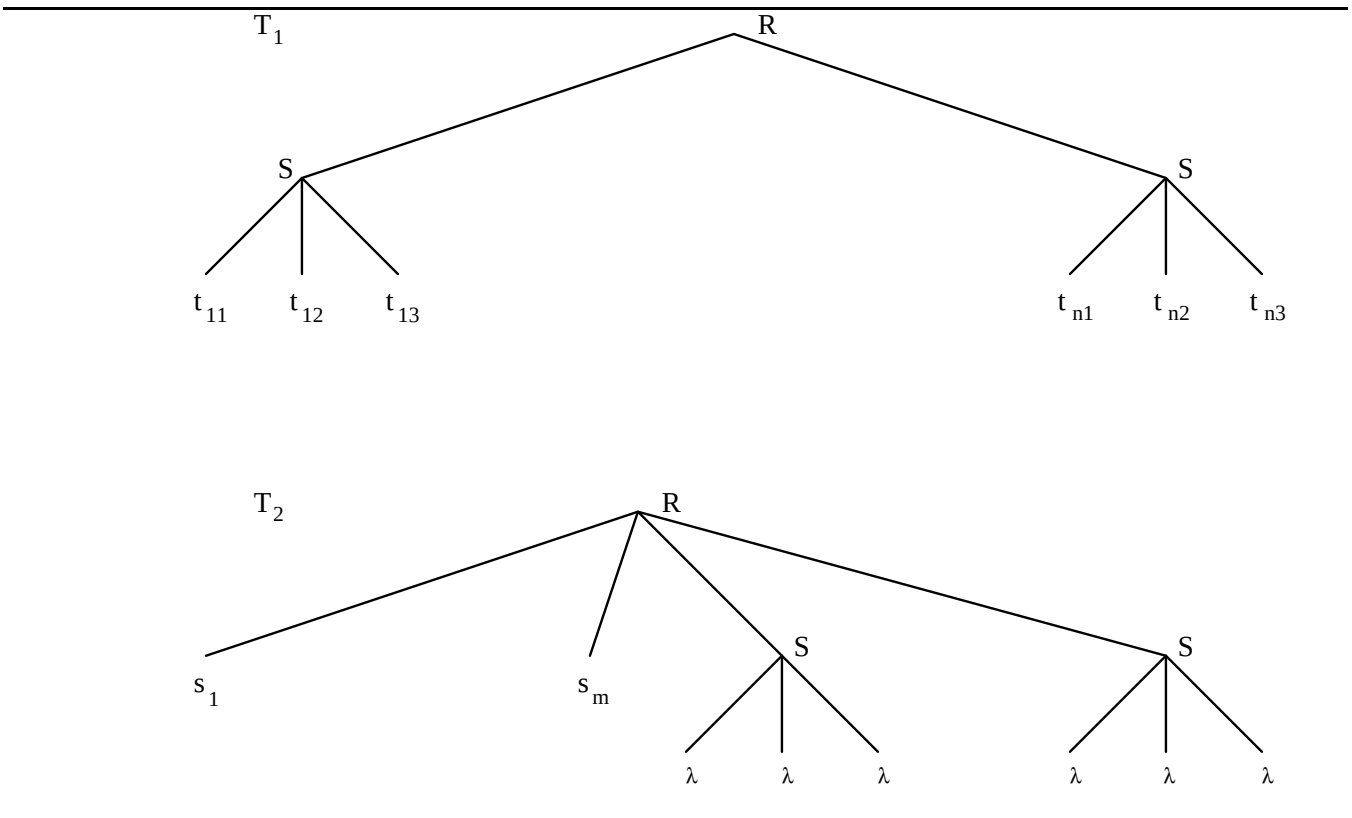

Figure 14.11: The reduction

The following lemmas show that $treedist(T_1, T_2) = 3n - 2k$ if and only if there exists an exact 3-cover. Since the problem is clearly in NP, the lemmas show that the problem is NP-complete. The proofs of these lemmas are left as exercises.

$$\leq O((deg(T_1) + deg(T_2))^2 \times \sum_{i=1}^{|T_1|} m_i \times \sum_{j=1}^{|T_2|} n_j) \quad \leq \quad O(|T_1| \times |T_2| \times (deg(T_1) + deg(T_2))^2)$$

If both T_1 and T_2 have degrees bounded by some constant, the time complexity becomes $O(|T_1| \cdot |T_2|)$. Note that the algorithm actually computes $A(T_1[i], T_2[j])$, $A(F_1[i], F_2[j])$, $A(F_1[i_s, i_t], F_2[j])$ and $A(F_1[i], F_2[j_s, j_t])$. With these data, materializing an optimal alignment can be found using back-tracking. The complexity remains the same.

14.3.6 Tree pattern discovery problem

We briefly discuss the pattern discovery problem. In matching problems, we are given a pattern and need to find a distance between the pattern and one or more objects; in discovery problems, by contrast, we are given two objects and a "target" distance d and are asked to find the largest portions of the objects that differ by at most that distance. Specializing the discovery problem to a pair of trees, we want to find the largest connected component from each tree such that the distance between them is under the target distance value.

Let us consider the connected component in one of the trees. Since it is connected, it must be rooted at a node and is generated by cutting off some subtrees. This means that a naive algorithm for tree pattern discovery would have to consider all the subtree pairs and for each subtree pair all the possible cuts of its subtrees. Since the number of such possible cuts is exponential, the naive algorithm is clearly impractical.

Instead we use a compound form of dynamic programming. By compound, we mean that dynamic programming is applied (1) to compute sizes of common patterns between two subtree pairs given a set of cuts; (2) to find the cuttings that yield distances less than or equal to the target one; (3) to compute the optimal cuttings for distance k, $1 \leq k \leq d$, given the optimal cuttings for distances 0 to $k-1$.

In the computation of an optimal solution for distance value k, we also have to solve a problem which is unique for trees. Consider a pair of subtrees s_1 and s_2 whose roots map to one another in the optimal solution for distance value k. Then, in general, there are several subtree pairs of s_1 and s_2 that map to one another. We have to determine how the distance value k should be distributed to these several subtree pairs so that we can obtain the optimal solution. We solve this problem by partitioning the subtrees of s_1, respectively s_2, into a forest and a subtree. We then compute the distance and size values from forest to forest and from subtree to subtree.

Using this general framework, we can solve the tree pattern discovery problem for edit and alignment distance measures. Given a target distance value d, the time complexity of the algorithm for edit distance measure is $(d^2 \times |T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)$ and the time complexity of the algorithm for alignment distance measure is $O(d^2 \times |T_1| \times |T_2| \times (deg(T_1) + deg(T_2))^2)$.

14.4 Algorithms and hardness results for unordered trees

Recall that unordered labeled trees are rooted trees whose nodes are labeled and in which only ancestor relationships are significant (the left-to-right order among siblings is not significant). Such trees arise naturally in genealogical studies, for example, or in parts explosions. For many

---

Algorithm: ALIGN(T_1,T_2)

begin
$A(\theta, \theta) := 0$;
for $i := 1$ **to** $|T_1|$
  Initialize $A(T_1[i], \theta)$ and $A(F_1[i], \theta)$ as in Lemma 8;
for $j := 1$ **to** $|T_2|$
  Initialize $A(\theta, T_2[j])$ and $A(\theta, F_2[j])$ as in Lemma 8;
for $i := 1$ **to** $|T_1|$
  for $j := 1$ **to** $|T_2|$
    for $s := 1$ **to** m_i
      Call Procedure $forest_align$ on $F_1[i_s, i_{m_i}]$ and $F_2[j]$;
    for $t := 1$ **to** n_j
      Call Procedure $forest_align$ on $F_1[i]$ and $F_2[j_t, i_{n_j}]$;
    Compute $A(T_1[i], T_2[j])$ as in Lemma 9.
end

Output: $A(T_1[i], T_2[i])$, where $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$.

Figure 14.10: Computing $A(T_1, T_2)$.

Input: $F_1[i_s, i_{m_i}]$ and $F_2[j_t, j_{n_j}]$.

Procedure: $forest_align()$

begin
  $A(F_1[i_s, i_{s-1}], F_2[j_t, j_{t-1}]) := 0$;
  for $p := s$ **to** m_i
    $A(F_1[i_s, i_p], F_2[j_t, j_{t-1}]) := A(F_1[i_s, i_{p-1}], F_2[j_t, j_{t-1}]) + A(T_1[i_p], \theta)$;
  for $q := t$ **to** n_j
    $A(F_1[i_s, i_{s-1}], F_2[j_t, j_q]) := A(F_1[i_s, i_{s-1}], F_2[j_t, j_{q-1}]) + A(\theta, T_2[j_q])$;
  for $p := s$ **to** m_i
    for $q := t$ **to** n_j
      Compute $A(F_1[i_s, i_p], F_2[j_t, j_q])$ as in Lemma 10.
end

Output: $A(F_1[i_s, i_p], F_2[j_t, j_q])$, where $s \leq p \leq m_i$ and $t \leq q \leq n_j\}$.

Figure 14.9: Computing $\{A(F_1[i_s, i_p], F_2[j_t, j_q]) | s \leq p \leq m_i, t \leq q \leq n_j\}$ for fixed s and t.

Algorithm

It follows from the above lemmas that, for each pair of subtrees $T_1[i]$ and $T_2[j]$, we have to compute $A(F_1[i], F_2[j_s, j_t])$ for all $1 \leq s \leq t \leq n_j$, and $A(F_1[i_s, i_t], F_2[j])$ for all $1 \leq s \leq t \leq m_i$. That is, we need align $F_1[i]$ with each subforest of $F_2[j]$, and conversely align $F_2[j]$ with each subforest of $F_1[i]$. Note that we do not have to align an arbitrary forest of $F_1[i]$ with an arbitrary forest of $F_2[j]$. Otherwise the time complexity would be higher.

For each fixed s and t, where either $s = 1$ or $t = 1$, $1 \leq s \leq m_i$ and $1 \leq t \leq n_j$, the procedure in Figure 14.9 computes $\{A(F_1[i_s, i_p], F_2[j_t, j_q]) | s \leq p \leq m_i, t \leq q \leq n_j\}$, assuming that all $A(F_1[i_k], F_2[j_p, j_q])$ are known, where $1 \leq k \leq m_i$ and $1 \leq p \leq q \leq n_j$, and all $A(F_1[i_p, i_q], F_2[j_k])$ are known, where $1 \leq p \leq q \leq m_i$ and $1 \leq k \leq n_j$.

Hence we can obtain $A(F_1[i], F_2[j_s, j_t])$ for all $1 \leq s \leq t \leq n_j$ by calling Procedure $forest_align$ n_j times, and $A(F_1[i_s, i_t], F_2[j])$ for all $1 \leq s \leq t \leq m_i$ by calling Procedure $forest_align$ m_i times. Our algorithm to compute $A(T_1, T_2)$ is given in Figure 14.10.

For an input $F_1[i_s, i_{m_i}]$ and $F_2[j_t, j_{n_j}]$, the running time of Procedure $forest_align$ is bounded by

$$O((m_i - s) \times (n_j - t) \times (m_i - s + n_j - t)) = O(m_i \times n_j \times (m_i + n_j)).$$

So, for each pair i and j, Algorithm ALIGN spends $O(m_i \times n_j \times (m_i + n_j)^2)$ time. Therefore, the time complexity of Algorithm ALIGN is

$$\sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_2|} O(m_i \times n_j \times (m_i + n_j)^2) \quad \leq \quad \sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_2|} O(m_i \times n_j \times (deg(T_1) + deg(T_2))^2)$$

Lemma 9

$$A(T_1[i], T_2[j]) = \min \begin{cases} A(\theta, T_2[j]) + \min_{1 \le r \le n_j} \{A(T_1[i], T_2[j_r]) - A(\theta, T_2[j_r])\} \\ A(T_1[i], \theta) + \min_{1 \le r \le m_i} \{A(T_1[i_r], T_2[j]) - A(T_1[i_r], \theta)\} \\ A(F_1[i], F_2[j]) + \gamma(t_1[i], t_2[j]) \end{cases}$$

Proof: Consider an optimal alignment (tree) $\mathcal{A}$ of $T_1[i]$ and $T_2[j]$. There are four cases: (1) $(t_1[i], t_2[j])$ is a label in $\mathcal{A}$, (2) $(t_1[i], \lambda)$ and $(t_1[k], t_2[j])$ are labels in $\mathcal{A}$ for some k, (3) $(t_1[i], t_2[k])$ and $(\lambda, t_2[j])$ are labels in $\mathcal{A}$ for some k, (4) $(t_1[i], \lambda)$ and $(\lambda, , t_2[j])$ are labels in $\mathcal{A}$. We actually need not consider Case 4 since in this case we can delete the two nodes and then add $(t_1[i], t_2[j])$ as the new root, resulting in a better alignment.

Case 1. The root of $\mathcal{A}$ must be labeled as $(t_1[i], t_2[j])$. Clearly, $A(T_1[i], T_2[j]) = A(F_1[i], F_2[j]) + \gamma(t_1[i], t_2[j])$.

Case 2. The root of $\mathcal{A}$ must be labeled as $(t_1[i], \lambda)$. In this case k must be a node in $T_1[i_r]$ for some $1 \le r \le m_i$. Therefore, $A(T_1[i], T_2[j]) = A(T_1[i], \theta) + \min_{1 \le r \le m_i} \{A(T_1[i_r], T_2[j]) - A(T_1[i_r], \theta)\}$.

Case 3. Similar to Case 2. □

Note, the above implies that $A(F_1[i], F_2[j])$ is required for computing $A(T_1[i], T_2[j])$.

Lemma 10 *For any s, t such that $1 \le s \le m_i$ and $1 \le t \le n_j$,*

$$A(F_1[i_1, i_s], F_2[j_1, j_t]) = \min \begin{cases} A(F_1[i_1, i_{s-1}], F_2[j_1, j_t]) + A(T_1[i_s], \theta) \\ A(F_1[i_1, i_s], F_2[j_1, j_{t-1}]) + A(\theta, T_2[j_t]) \\ A(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}]) + A(T_1[i_s], T_2[j_t]) \\ \gamma(\lambda, t_2[j_t]) + \min_{1 \le k < s} \{A(F_1[i_1, i_{k-1}], F_2[j_1, j_{t-1}]) + A(F_1[i_k, i_s], F_2[j_t])\} \\ \gamma(t_1[i_s], \lambda) + \min_{1 \le k < t} \{A(F_1[i_1, i_{s-1}], F_2[j_1, j_{k-1}]) + A(F_1[i_s], F_2[j_k, j_t])\} \end{cases}$$

Proof: Consider an optimal alignment (forest) $\mathcal{A}$ of $F_1[i_1, i_s]$ and $F_2[j_1, j_t]$). The root of the rightmost tree in $\mathcal{A}$ is labeled by either $(t_1[i_s], t_2[j_t])$, $(t_1[i_s], \lambda)$, or $(\lambda, t_2[j_t])$.

Case 1: the label is $(t_1[i_s], t_2[j_t])$. In this case, the rightmost tree must be an optimal alignment of $T_1[i_s]$ and $T_2[j_t]$. Therefore $A(F_1[i_1, i_s], F_2[j_1, j_t]) = A(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}]) + A(T_1[i_s], T_2[j_t])$.

Case 2: the label is $(t_1[i_s], \lambda)$. In this case, there is a k, $0 \le k \le t$, such that $T_1[i_s]$ is aligned with the subforest $F_2[j_{t-k+1}, j_t]$. A key observation here is the fact that subtree $T_2[j_{t-k+1}]$ is not split by the alignment with $T_1[i_s]$. There are three subcases.

2.1 $(k = 0)$ *I.e.*, $F_2[j_{t-k+1}, j_t] = \theta$. Therefore,
$A(F_1[i_1, i_s], F_2[j_1, j_t]) = A(F_1[i_1, i_{s-1}], F_2[j_1, j_t]) + A(T_1[i_s], \theta)$.

2.2 $(k = 1)$ *I.e.*, $F_2[j_{t-k+1}, j_t] = T_2[j_t]$. This is the same as in Case 1.

2.3 $(k \ge 2)$ This is the most general case. It is easy to see that

$$A(F_1[i_1, i_s], F_2[j_1, j_t]) = \gamma(t_1[i_s], \lambda) + \min_{1 \le k < t} \{A(F_1[i_1, i_{s-1}], F_2[j_1, j_{k-1}]) + A(F_1[i_s], F_2[j_k, j_t])\}.$$

Case 3: the label is $(\lambda, t_2[j_t])$. Similar to Case 2. □

Definitions

Let T_1 and T_2 be two labeled trees. An alignment $\mathcal{A}$ of T_1 and T_2 is obtained by first inserting nodes labeled with λ into T_1 and T_2 such that the two resulting trees T_1' and T_2' are topologically isomorphic, *i.e.*, they are identical if the labels are ignored, and then T_1' is *overlaid* on T_2'. An example alignment is shown in Figure 14.8. An edit cost is defined for each pair of labels. The *value* of alignment $\mathcal{A}$ is the sum of the costs of all pairs of corresponding labels. An *optimal* alignment is one that minimizes the value over all possible alignments. The *alignment distance* between T_1 and T_2 is the value of an optimal alignment of T_1 and T_2.

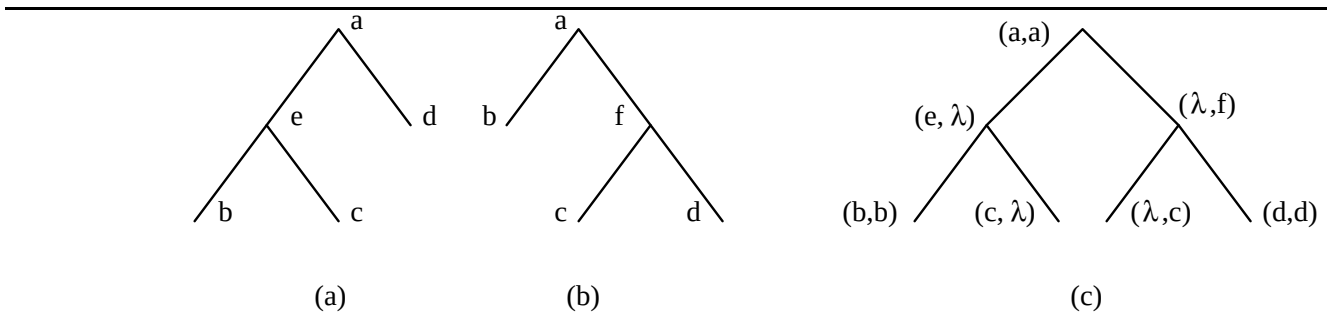

Figure 14.8: (a) Tree T_1. (b) Tree T_2. (c) The optimal alignment of T_1 and T_2.

It is easy to see that in general the edit distance is smaller than the alignment distance for trees. The reason is that each alignment of trees actually corresponds to a restricted tree edit in which all the insertions precede all the deletions. Note that, the order of edit operations is not important for sequences. Also, it seems that alignment charges more for the structural dissimilarity at the top levels of the trees than at the lower levels, whereas edit treats all the levels the same.

The notion of alignment can be easily extended to ordered forests. The only change is that it is now possible to insert a node (as the root) to join a consecutive subsequence of trees in the forest. Denote the alignment distance between forests F_1 and F_2 as $A(F_1, F_2)$. Let θ denote the empty tree, and $\gamma(a, b)$ denote the cost of the opposing letters a and b. Let T_1 and T_2 be two fixed ordered labeled trees throughout this section.

Formulas

Let $t_1[i]$ be a node of T_1 and $t_2[j]$ a node of T_2. Suppose that the degrees (number of children) of $t_1[i]$ and $t_2[j]$ are m_i and n_j, respectively. Denote the children of $t_1[i]$ as $t_1[i_1], \ldots, t_1[i_{m_i}]$ and the children of $t_2[j]$ as $t_2[j_1], \ldots, t_2[j_{n_j}]$. For any s, t, $1 \leq s \leq t \leq m_i$, let $F_1[i_s, i_t]$ represent the forest consisting of the subtrees $T_1[i_s], \ldots, T_1[i_t]$. For convenience, $F_1[i_1, i_{m_i}]$ is also denoted $F_1[i]$. Note that $F_1[i] \neq F_1[i, i]$. $F_2[j_s, j_t]$ and $F_2[j]$ are defined similarly.

The following lemmas form the basis of our algorithm. The first lemma is trivial.

Lemma 8 $A(\theta, \theta) = 0$; $A(F_1[i], \theta) = \sum_{k=1}^{m_i} A(T_1[i_k], \theta)$; $A(T_1[i], \theta) = A(F_1[i], \theta) + \gamma(t_1[i], \lambda)$; $A(\theta, F_2[j]) = \sum_{k=1}^{n_j} A(\theta, T_2[j_k])$; $A(\theta, T_2[j]) = A(\theta, F_2[j]) + \gamma(\lambda, t_2[j])$.

Procedure: $tree_vldc(i, j)$

```
begin
  for i_1 := l(i) to i do
    for j_1 := l(j) to j do
      if l(i_1) ≠ l(i) or l(j_1) ≠ l(j)then
          Calculate forest_vldc(T_1[l(i)..i_1], T_2[l(j)..j_1]) as in Lemma 5;
      else begin      /* l(i_1) = l(i) and l(j_1) = l(j) */
          if (p[i_1] ≠ | or j_1 = l(j)) then
              Calculate forest_vldc(T_1[l(i)..i_1], T_2[l(j)..j_1]) as in Lemma 6;
          if (p[i_1] = | and j_1 ≠ l(j)) then
              Calculate forest_vldc(T_1[l(i)..i_1], T_2[l(j)..j_1]) as in Lemma 7;
          tree_vldc(i_1, j_1) := forest_vldc(T_1[l(i)..i_1], T_2[l(j)..j_1]);
      end
end
```

Figure 14.7: Computing $tree_vldc(i, j)$.

is that preserving ancestor relationships in the mapping between trees prevents the analogous implication from holding. In addition, to compute the distance between two forests at stage k sometimes requires knowing whether two contained subtrees are distance k apart.

We overcome these problems by studying the relationship between identical subforests and distance mappings. We find the relevant identical forests by using suffix trees corresponding to different traversals.

The preprocessing time complexity is dominated by the cost of constructing suffix trees. It is bounded by $O(\log(|T_1| + |T_2|)) \leq O(\log(2 \times \min(|T_1|, |T_2|) + k)) \leq O(\log(\min(|T_1|, |T_2|)) + \log(k))$. The time complexity of the algorithm is: $O(k \times \log(k) \times \log(\min(|T_1|, |T_2|)))$ where k is the actual distance between T_1 and T_2.

14.3.5 Tree alignment problem

It is well known that edit and alignment are two equivalent notions for sequences. In particular, for any two sequences x_1 and x_2, the edit distance between x_1 and x_2 equals the value of an optimal alignment of x_1 and x_2. However, edit and alignment turn out to be very different for trees, see Figure 14.8. Here, we introduce the notion of *alignment of trees* as another measure of similarity of labeled trees. The notion is a natural generalization of alignment of sequences.

replaced by $t[j_1]$, in which case $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1 - 1, l(j)..j_1 - 1) + \gamma(p[i_1] \rightarrow t[j_1])$. The distance is the minimum of these two cases.

Since $j_1 = l(j)$, $forest_vldc(l(i)..i_1, l(j)..j_1 - 1) + \gamma(\lambda \rightarrow t[j_1]) = forest_vldc(P[l(i)..i_1], \emptyset) + \gamma(\lambda \rightarrow t[j_1]) \geq forest_vldc(P[l(i)..i_1], \emptyset) = forest_vldc(P[l(i)..i_1 - 1], \emptyset) = forest_vldc(l(i)..i_1 - 1, l(j)..j_1 - 1) + \gamma(p[i_1] \rightarrow t[j_1])$. Thus, we can add an additional item $forest_vldc(l(i)..i_1, l(j)..j_1 - 1) + \gamma(\lambda \rightarrow t[j_1])$ to the minimum expression, obtaining the formula asserted by the lemma. □

Lemma 7 *If $p[i_1] = |$ and $j_1 \neq l(j)$, then*

$$forest_vldc(l(i)..i_1, l(j)..j_1) = \mathbf{min} \begin{cases} forest_vldc(l(i)..i_1, \emptyset), \\ forest_vldc(l(i)..i_1 - 1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda), \\ forest_vldc(l(i)..i_1, l(j)..j_1 - 1) + \gamma(\lambda \rightarrow t[j_1]), \\ forest_vldc(l(i)..i_1 - 1, l(j)..j_1 - 1) + \gamma(p[i_1] \rightarrow t[j_1]), \\ \min_{t_k}\{tree_vldc(i_1, t_k) | 1 \leq k \leq n_{j_1}\} \end{cases}$$

where t_k, $1 \leq k \leq n_{j_1}$, are children of j_1.

Proof: Again, if $T[j_1]$ is cut, the distance should be $forest_vldc(l(i)..i_1, \emptyset)$. Otherwise, let M be a minimum-cost mapping between $P[l(i)..i_1]$ and $T[l(j)..j_1]$ after performing an optimal removal of subtrees of $T[l(j)..j_1]$. There are three cases.

(1) In the best substitution, $p[i_1]$ is replaced by an empty tree. So, $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1 - 1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda)$.

(2) In the best substitution, $p[i_1]$ is replaced by a nonempty tree and $t[j_1]$ is not touched by a line in M. So, $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1, l(j)..j_1 - 1) + \gamma(\lambda \rightarrow t[j_1])$.

(3) In the best substitution, $p[i_1]$ is replaced by a nonempty tree and $t[j_1]$ is touched by a line in M. So, $p[i_1]$ must be replaced by a path of the tree rooted at $t[j_1]$. Let the path end at node $t[d]$. Let the children of $t[j_1]$ be, in left-to-right order, $t[t_1], t[t_2], \ldots, t[t_{n_{j_1}}]$. There are two subcases.

(a) $d = j_1$. Thus, $|$ is replaced by $t[j_1]$. So $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1 - 1, l(j)..j_1 - 1) + \gamma(p[i_1] \rightarrow t[j_1])$.

(b) $d \neq j_1$. Let $t[t_k]$ be the child of $t[j_1]$ on the path from $t[j_1]$ to $t[d]$. We can cut all subtrees on the two sides of the path. So, $forest_vldc(l(i)..i_1, l(j)..j_1) = tree_vldc(i_1, t_k)$. The value of k ranges from 1 to n_{j_1}. Therefore, the distance is the minimum of the corresponding costs. □

These lemmas suggest the following algorithm. We omit the initialization steps.

14.3.4 Fast parallel algorithms for small differences

In our research, we have often imported technology developed for strings to develop fast tree algorithms. A particularly blatant example is our algorithm for the unit cost edit distance (unit cost means that node deletions, node relabellings, and node insertions all have the same cost). The algorithm starts from Ukkonen's 1983 technique of computing in waves along the center diagonals of the distance matrix. At the beginning of stage k, all distances up to $k-1$ have been computed. Stage k then computes in parallel all distances up to k. We use suffix trees as Landau and Vishkin to perform this computation fast.

In the string case, if $S_1[i..i+h] = S_2[j..j+h]$, then the distance between $S_1[1..i-1]$ and $S_2[1..j-1]$ is the same as between $S_1[1..i+h]$ and $S_2[1..j+h]$. The main difficulty in the tree case

Proof: Trivial. □

We compute $tree_vldc(i,j)$ for $1 \leq i \leq |P|$ and $1 \leq j \leq |T|$. In the intermediate steps, we need to calculate $forest_vldc(l(i)..i_1, l(j)..j_1)$ for $l(i) \leq i_1 \leq i$ and $l(j) \leq j_1 \leq j$. The algorithm considers the following two cases separately: (1) $P[l(i)..i_1]$ or $T[l(j)..j_1]$ is a forest; (2) both are trees. The overall strategy is to try to find a best substitution for the VLDCs in $P[l(i)..i_1]$, and ask whether or not $T[j_1]$ is cut. (Note that in the algorithm, $\gamma(p[i_1] \rightarrow \lambda) = 0$ and $\gamma(p[i_1] \rightarrow t[j_1]) = 0$ when $p[i_1] = |$.)

Lemma 5 *If $P[l(i)..i_1]$ or $T[l(j)..j_1]$ is a forest, then*

$$forest_vldc(l(i)..i_1, l(j)..j_1) = \mathbf{min} \begin{cases} forest_vldc(l(i)..i_1, l(j)..l(j_1)-1), \\ forest_vldc(l(i)..i_1-1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda), \\ forest_vldc(l(i)..i_1, l(j)..j_1-1) + \gamma(\lambda \rightarrow t[j_1]), \\ forest_vldc(l(i)..l(i_1)-1, l(j)..l(j_1)-1) + tree_vldc(s,t) \end{cases}$$

Proof: If $T[j_1]$ is cut, then $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1, l(j)..l(j_1)-1)$. Otherwise, consider a minimum-cost mapping M between $P[l(i)..i_1]$ and $T[l(j)..j_1]$ after performing an optimal removal of subtrees of $T[l(j)..j_1]$. The distance is the minimum of the following three cases.

(1) $p[i_1]$ is not touched by a line in M. (This includes the case where $p[i_1] = |$ is replaced by an empty tree.) So, $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1-1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda)$.

(2) $t[j_1]$ is not touched by a line in M. So, $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1, l(j)..j_1-1) + \gamma(\lambda \rightarrow t[j_1])$.

(3) $p[i_1]$ and $t[j_1]$ are both touched by lines in M. (This includes the case where $p[i_1] = |$ is replaced by a path of nodes in T.) By the ancestor and sibling conditions on mappings, (i_1, j_1) must be in M. By the ancestor condition on mapping, any node in $P[i_1]$ (the subtree of P rooted at i_1) can be touched only by a node in $T[j_1]$. Hence, $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..l(i_1)-1, l(j)..l(j_1)-1) + tree_vldc(i_1, j_1)$. □

Lemma 6 *If $p[i_1] \neq |$ or $j_1 = l(j)$, then*

$$forest_vldc(l(i)..i_1, l(j)..j_1) = \mathbf{min} \begin{cases} forest_vldc(l(i)..i_1, \emptyset), \\ forest_vldc(l(i)..i_1-1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda), \\ forest_vldc(l(i)..i_1, l(j)..j_1-1) + \gamma(\lambda \rightarrow t[j_1]), \\ forest_vldc(l(i)..i_1-1, l(j)..j_1-1) + \gamma(p[i_1] \rightarrow t[j_1]) \end{cases}$$

Proof: If $T[j_1]$ is cut, then the distance should be $forest_vldc(l(i)..i_1, \emptyset)$. Otherwise, consider a minimum-cost mapping M between $P[l(i)..i_1]$ and $T[l(j)..j_1]$ after performing an optimal removal of subtrees of $T[l(j)..j_1]$. There are two cases.

(1) $p[i_1] \neq |$. Depending on whether $p[i_1]$ or $t[j_1]$ is touched by a line in M, we argue similarly as in Lemma 5.

(2) $p[i_1] = |$ and $j_1 = l(j)$. Then, in the best substitution, either $|$ is replaced by an empty tree, in which case $forest_vldc(l(i)..i_1, l(j)..j_1) = forest_vldc(l(i)..i_1-1, l(j)..j_1) + \gamma(p[i_1] \rightarrow \lambda)$, or $|$ is

Approximate tree matching with variable length don't cares

Approximate tree matching is a generalization of approximate string matching. Given two trees, we view one tree as the pattern tree and the other as the data tree. We want to match, approximately, the pattern tree to the data tree. In the match, we allow the pattern tree to match only part of the data tree. For this purpose we allow subtrees of the data tree to be cut freely. Also we allow the pattern tree to contain *variable length don't cares* indexvariable length don't cares to suppress the details of the data tree which are not interested. Intuitively, these VLDC's match part of a path with or without the subtrees branching off that path. We now give the formal definitions for cut, variable length don't cares, and approximate tree matching.

Cutting at node $t[i]$ means removing the subtree rooted at $t[i]$. Let C be a set of nodes. We define C to be a set of *consistent subtree cuts* if $t[i], t[j] \in C$ implies that neither is an ancestor of the other. We use $Cut(T, C)$ to represent the tree T with all subtrees in rooted at nodes of C removed. Let $subtree(T)$ be the set of all possible sets of consistent subtree cuts. The term approximate tree matching (without VLDC's) is defined as computing

$$tree_cut(P, T) = \min_{C \in subtree(T)} \{treedist(P, cut(T, C))\}.$$

Intuitively, this is the distance between the pattern tree and the cut data tree, where the cut yields the smallest possible distance.

We consider two VLDC's: | and $\wedge$. A node with | in the pattern tree can substitute part of a path from the root to a leaf of the data tree. A node with $\wedge$ in the pattern tree can substitute part of such path and all the subtrees emanating from the nodes of that path, except possibly at the lowest node of that path. At the lowest node, the $\wedge$ symbol can substitute for a set of leftmost subtrees and a set of rightmost subtrees. We call | a path VLDC and $\wedge$ an umbrella VLDC, because of the shape they impose on the tree.

Let P be a pattern tree that contains both umbrella-VLDCs and path-VLDCs and let T be a data tree. A VLDC-substitution s on P replaces each path-VLDC in P by a path of nodes in T and each umbrella-VLDC in P by an umbrella pattern of nodes in T. We require that any mapping from the resulting (VLDC-free) pattern $\bar{P}$ to T map the substituting nodes to themselves. (Thus, no cost is induced by VLDC substitutions.) The approximate matching between P and T w.r.t. s, is defined as $tree_vldc(P, T, s) = tree_cut(\bar{P}, T, s)$. Then,

$$tree_vldc(P, T) = \min_{s \in \mathcal{S}} \{tree_vldc(P, T, s)\}$$

where $\mathcal{S}$ is the set of all possible VLDC-substitutions.

The algorithm

The following lemma shows that the two kinds of VLDCs are the same in the presence of free subtree cuts.[2]

Lemma 4 *A path-VLDC can be substituted for an umbrella-VLDC or vice versa without changing the mapping or the distance value when we allow subtrees to be cut freely from the text tree.*

[2]The case for matching without cuttings is much more involved. In that case, we have to consider the two kinds of VLDCs separately and need an auxiliary suffix forest distance measure when dealing with umbrella-VLDCs.

The computation of $treedist(i,j)$ makes strong use of the above lemmas. From the algorithm,

---

Procedure: $treedist(i,j)$

begin
$forestdist(\theta,\theta) = 0$;
for i_1:=$l(i)$ **to** i
$forestdist(T_1[l(i)..i_1],\theta) = forestdist(T_1[l(i)..i_1-1],\theta) + \gamma(t_1[i_1] \rightarrow \lambda)$
for j_1:=$l(j)$ **to** j
$forestdist(\theta,T_2[l(j)..j_1]) = forestdist(\theta,T_2[l(j)..j_1-1]) + \gamma(\lambda \rightarrow t_2[j_1])$
for i_1:=$l(i)$ **to** i
for j_1:=$l(j)$ **to** j
if $l(i_1) = l(i)$ and $l(j_1) = l(j)$ then
Calculate $forestdist(T_1[l(i)..i_1],T_2[l(j)..j_1])$ as in Lemma 3 (1).
$treedist(i_1,j_1) = forestdist(T_1[l(i)..i_1],T_2[l(j)..j_1])$
/* put in permanent array */
else
Calculate $forestdist(T_1[l(i)..i_1],T_2[l(j)..j_1])$ as in Lemma 3 (2).
end

Output: $treedist(T_1[s],T_2[t])$, where $s \in desc(i)$ and $t \in desc(j)$, $l(s) = l(i)$ and $l(t) = l(j)$.

Figure 14.6: Computing $treedist(i,j)$.

it is easy to see that for any subtree pair $T_1[i]$ and $T_2[j]$ the time complexity for $treedist(i,j)$ is $O(|T_1[i]| \times |T_2[j]|)$ provided all the necessary treedist() values are available are available. If we compute all the $treedist()$ bottom up, we can compute the distance between T_1 and T_2. Therefore the time complexity of the algorithm can be bounded by

$$O(\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|}|T_1[i]| \times |T_2[j]|) = O(\sum_{i=1}^{|T_1|}|T_1[i]| \times \sum_{j=1}^{|T_2|}|T_2[j]|) = O(|T_1| \times |T_2| \times depth(T_1) \times depth(T-2)).$$

In fact the complexity is a bit better than this. After more careful analysis, we can show that the complexity is $(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$. where $leaves(T_1)$ is the number of leaves in T_1. One implication is that this algorithm can be used to compute the distance between two strings in time $O(|T_1| \times |T_2|)$.

14.3.3 Pattern trees with variable length don't cares

Many problems in strings can be solved with dynamic programming. Similarly, our algorithm applies not only to tree distance but also to a variety of tree problems with the same time complexity.

Algorithm: EDIT(T_1,T_2)

begin
Preprocessing:
(To compute $l()$, $LR_keyroots(T_1)[]$ and $LR_keyroots(T_2)[]$)
for $s := 1$ **to** $|LR_keyroots(T_1)|$
for $t := 1$ **to** $|LR_keyroots(T_2)|$
$i = LR_keyroots(T_1)[s]$;
$j = LR_keyroots(T_2)[t]$;
Compute $treedist(i, j)$;
end

Output: $tree_dist(T_1[i], T_2[j])$, where $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$.

Figure 14.5: Computing $treedist(T_1, T_2)$.

Algorithm

Lemma 3 has three important implications.

First the formulas it yields suggest that we can use a dynamic programming style algorithm to solve the tree distance problem.

Second, from (2) of Lemma 3 we observe that in order to compute $treedist(i_1, j_1)$ we need in advance almost all values of $treedist(i, j)$ where i is the root of a subtree containing i_1 and j is the root of a subtree containing j_1. This suggests using a bottom-up procedure for computing all subtree pairs.

Third, from (1) in Lemma 3 we can observe that when i is in the path from $l(i_1)$ to i_1 and j is in the path from $l(j_1)$ to j_1, we do not need to compute $treedist(i, j)$ separately. These subtree distances can be obtained as a byproduct of computing $treedist(i_1, j_1)$.

These implications lead to the following definition and then our algorithm. Let us define the set $LR_keyroots$ of tree T as follows:

$$LR_keyroots(T) = \{k \mid \text{there exists no } k' > k \text{ such that } l(k) = l(k')\}$$

That is, if k is in $LR_keyroots(T)$ then either k is the root of T or $l(k) \neq l(p(k))$, i.e. k has a left sibling. Intuitively, this set will be the roots of all the subtrees of tree T that need separate computations.

It is easy to see that there is a linear time algorithm to compute the function $l()$ and the set $LR_keyroots$. We can also assume that the result is in array l and $LR_keyroots$. Further in array $LR_keyroots$ the order of the elements is in increasing order.

We are now ready to present a simple algorithm for computing edit distance.

We use dynamic programming to compute $treedist(i, j)$. The forestdist values computed and used here are put in a temporary array; that is freed once the corresponding treedist is computed. The treedist values are put in the permanent treedist array.

Proof: Trivial. □

Lemma 2 *Let $i \in desc(i_1)$ and $j \in desc(j_1)$. Then*

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ +forestdist(l(i)..i-1, l(j)..j-1) + \gamma(t_1[i] \to t_2[j]) \end{cases}$$

Proof: We compute $forestdist(l(i_1)..i, l(j_1)..j)$ for $l(i_1) \leq i \leq i_1$ and $l(j_1) \leq j \leq j_1$. We are trying to find a minimum-cost map M between $forest(l(i_1)..i)$ and $forest(l(j_1)..j)$. The map can be extended to $t_1[i]$ and $t_2[j]$ in three ways.

$t_1[i]$ is not touched by a line in M. Then $(i, \lambda) \in M$. So, $forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(t_1[i] \to \lambda)$.

$t_2[j]$ is not touched by a line in M. Then $(\lambda, j) \in M$. So, $forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\lambda \to t_2[j])$.

$t_1[i]$ and $t_2[j]$ are both touched by lines in M. Then $(i, j) \in M$. Here is why. Suppose (i, k) and (h, j) are in M. if $l(i_1) \leq h \leq l(i) - 1$, then i is to the right of h so k must be to the right of j by the sibling condition on mappings. This is impossible in $forest(l(j_1)..j)$. Similarly, if i is a proper ancestor of h, then k must be a proper ancestor of j by the ancestor condition on mappings. This too is impossible. So, $h = i$. By symmetry, $k = j$ and $(i, j) \in M$.

Now, by the ancestor condition on mappings, any node in subtree $T_1[i]$ can only be touched by a node in subtree $T_2[j]$. Hence, $forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + forestdist(l(i)..i-1, l(j)..j-1) + \gamma(t_1[i] \to t_2[j])$.

Since these three cases express all the possible mappings yielding $forestdist(l(i_1)..i, l(j_1)..j)$, we take the minimum of these three costs. Thus the lemma is proved. □

Lemma 3 *Let $i \in desc(i_1)$ and $j \in desc(j_1)$. Then*

(1) if $l(i) = l(i_1)$ and $l(j) = l(j_1)$

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..i-1, l(j_1)..j-1) + \gamma(t_1[i] \to t_2[j]) \end{cases}$$

(2) if $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$ (i.e., otherwise)

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(t_1[i] \to \lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\lambda \to t_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedist(i, j) \end{cases}$$

Proof: Immediately from Lemma 2. □

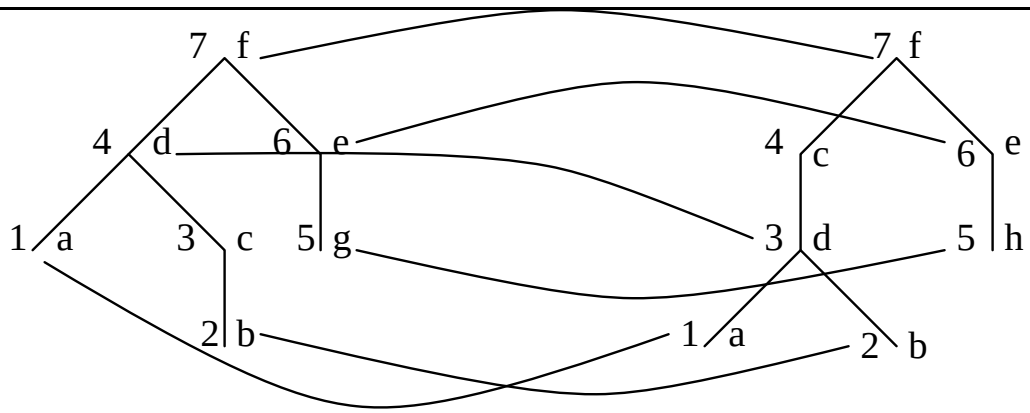

Figure 14.4: Mapping

We will use M instead of (M,T_1,T_2) if there is no confusion. Let M be a mapping from T_1 to T_2, the cost of M is defined as follows:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(t_1[i] \rightarrow t_2[j]) + \sum_{\{i|\ \nexists\ j\ s.t.\ (i,j)\in M\}} \gamma(t_1[i] \rightarrow \lambda) + \sum_{\{j|\ \nexists\ i\ s.t.\ (i,j)\in M\}} \gamma(\lambda \rightarrow t_2[j])$$

Mappings can be composed. Let M_1 be a mapping from T_1 to T_2 and let M_2 be a mapping from T_2 to T_3. Define $M_1 \circ M_2 = \{(i,j) \mid \exists\ k\ s.t.\ (i,k) \in M_1 \text{ and } (k,j) \in M_2\}$ It is easy to show that $M_1 \circ M_2$ is a mapping and $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2)$.

The relation between a mapping and a sequence of edit operation is as follows: given S, a sequence $s_1, \ldots, s_k$ of edit operations from T_1 to T_2, there exists a mapping M from T_1 to T_2 such that $\gamma(M) \leq \gamma(S)$; conversely, for any mapping M, there exists a sequence of editing operations such that $\gamma(S) = \gamma(M)$. This implies that $\gamma(T_1,T_2) = \min\{\gamma(M) \mid M \text{ is a mapping from } T_1 \text{ to } T_2\}$. Specifically, nodes in T_1 that are untouched by M correspond to deletions from T_1, nodes in T_1 connected by M to T_2 correspond to null edits (if the connected nodes have the same label) or relabelings (if the connected nodes have different labels), and nodes in T_2 that are untouched by M correspond to insertion operations. We will use the mapping idea to design the algorithm in the next subsection since the concept of mapping is easy to visualize and is order-independent.

General formula

The distance between $T_1[i'..i]$ and $T_2[j'..j]$ is denoted $forestdist(T_1[i'..i], T_2[j'..j])$ or simply $forestdist(i'..i, j'..j)$ if the context is clear. We use a more abbreviated notation for certain special cases. The distance between $T_1[i]$ and $T_2[j]$ is sometimes denoted $treedist(i,j)$.

We first present three lemmas and then give the algorithm.

Lemma 1 *(i)* $forestdist(\theta,\theta) = 0$
(ii) $forestdist(l(i_1)..i,\theta) = forestdist(l(i_1)..i-1,\theta) + \gamma(t_1[i] \rightarrow \lambda)$
(iii) $forestdist(\theta,l(j_1)..j) = forestdist(\theta,l(j_1)..j-1) + \gamma(\lambda \rightarrow t_2[j])$
where $i \in desc(i_1)$ *and* $j \in desc(j_1)$

and T have the same label, their algorithm tries to embed P into T by embedding the subtrees of P as deeply and as far to the left as possible in T. The time complexity of their algorithm is $O(|T_1| \times |T_2|)$.

They showed that the unordered inclusion problem is NP-complete.

14.3 Tree edit and tree alignment algorithms for ordered trees

14.3.1 Notation

While computing the tree-to-tree edit distance, we must compute, as subroutines, the distance between certain pairs of subtrees and between certain pairs of ordered subforests. An ordered subforest of a tree T is a collection of subtrees of T appearing in the same order as they appear in T.

Specifically, we use a left-to-right postorder numbering for the nodes in the trees. For a tree T, $t[i]$ represents the ith node of tree T We use $T[i]$ to represent subtree of T rooted at $t[i]$ and $F[i]$ to represent the ordered subforest obtained by deleting $t[i]$ from $T[i]$. We use $desc(i)$ to denote the set of descendants of $t[i]$

We use $T[i..j]$ to denote the substructure of T induced by the nodes numbered i to j inclusive. In general $T[i..j]$ is an ordered forest.

Let $t[i_1], t[i_2], ..., t[i_{n_i}]$ be the children of $t[i]$. We use $F[i_r, i_s]$, $1 \leq r \leq s \leq n_i$, to represent the forest consisting of the subtrees $T[i_r], ..., T[i_s]$. $F[i_1, i_{n_i}] = F[i]$ and $F[i_p, i_p] = T[i_p] \neq F[i_p]$.

Let $l(i)$ be the postorder number of the leftmost leaf descendant of the subtree rooted at $t[i]$. When $t[i]$ is a leaf, $l(i) = i$. With this notation $T[i] = T[l(i)..i]$ and $F[i] = T[l(i)..i-1]$.

We use $depth(T)$ to represent the depth of tree T; $leaves(T)$ to represent the number of leaves of tree T; and $deg(T)$ to represent the degree of tree T.

14.3.2 Basic tree edit distance computation

Mapping and edit distance

The edit operations give rise to a *mapping* which is a graphical specification of which edit operations apply to each node in the two trees (or two ordered forests). The mapping in Figure 14.4 shows a way to transform T_1 to T_2. It corresponds to the edit sequence (delete(node with label c), change(node with label g to label h), insert(node with label c)).

Formally we define a triple (M, T_1, T_2) to be a mapping from T_1 to T_2, where M is any set of pair of integers (i, j) satisfying:

(1) $1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|$;

(2) For any pair of (i_1, j_1) and (i_2, j_2) in M,

  (a) $i_1 = i_2$ iff $j_1 = j_2$ (one-to-one)

  (b) $t_1[i_1]$ is to the left of $t_1[i_2]$ iff $t_2[j_1]$ is to the left of $t_2[j_2]$ (sibling order preserved)

  (c) $t_1[i_1]$ is an ancestor of $t_1[i_2]$ iff $t_2[j_1]$ is an ancestor of $t_2[j_2]$ (ancestor order preserved)

3. $t_1[i]$ matches $t_2[j]$. In this case, consider the subtrees $t_1[i_1], t_1[i_2], ..., t_1[i_{n_i}]$ and $t_2[j_1], t_2[j_2], ..., t_2[j_{n_j}]$ as two sequences and each individual subtree as a whole entity. Use the sequence edit distance to determine the distance between $t_1[i_1], t_1[i_2], ..., t_1[i_{n_i}]$ and $t_2[j_1], t_2[j_2], ..., t_2[j_{n_j}]$.

From the above description it is easy to see the difference between this distance and the edit distance. This algorithm considers each subtree as a whole entity. It does not allow one subtree of T_1 to map to more than one subtrees of T_2. Using the definition of edit distance, we can delete the root of one subtree and then map the remaining subtrees of this subtree to more than one subtrees.

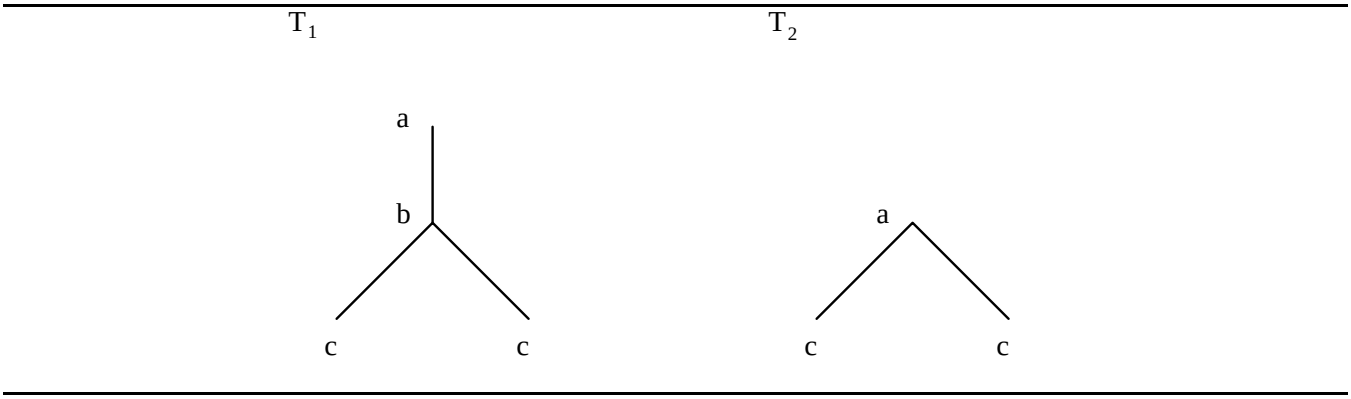

Figure 14.3: Lu's distance are different from edit distance

Figure 14.3 shows an example. the edit distance is 1 since we only need to delete node b. The distance according to Lu's algorithm is 2 since we can delete node a of tree T_1 and than replace node b by a. We cannot directly delete node b since if we map a to a, then subtree rooted at b can only map to one of the two subtrees of tree T_2, resulting a distance of 3. For two level trees, this algorithm does in fact compute the edit distance between two ordered trees, but not for trees with more levels.

Variants of the problem

Selkow gave the another tree edit algorithm in which the insertions and deletions are restricted to the leaves of the trees. Only leaves may be deleted and a node may be inserted only as a leaf.

In this case, it is easy to see that if $t_1[i]$ maps to $t_2[j]$ then the parent of $t_1[i]$ must map to the parent of $t_2[j]$. The reason is that if $t_1[i]$ is not deleted, its parent can not be deleted or inserted. This means that if two nodes are matched, then their parents must also be matched. Yang later give an algorithm to identify the syntactic differences between two programs. His algorithm is basically a variation of Selkow's.

It is easy to design an algorithm using string edit algorithm as a subroutine to solve this problem. The time complexity is $O(|T_1| \times |T_2|)$.

Kilpelainen and Mannila introduced the tree inclusion problem. Given a pattern tree P and a target tree T, tree inclusions asks whether P can be embedded into to T. An alternative definition is to get P by deleting nodes of T. Both ordered trees and unordered trees are considered.

Since there may be exponentially many ordered embeddings of P to T, they used a concept called left embedding to avoid searching among these embeddings. Assume that the roots of P

$t_2[s]$. Furthermore, all the nodes on the path from $t_1[i]$ to $t_1[r]$ are deleted and all the nodes on the path from $t_2[j]$ to $t_2[s]$ are inserted.

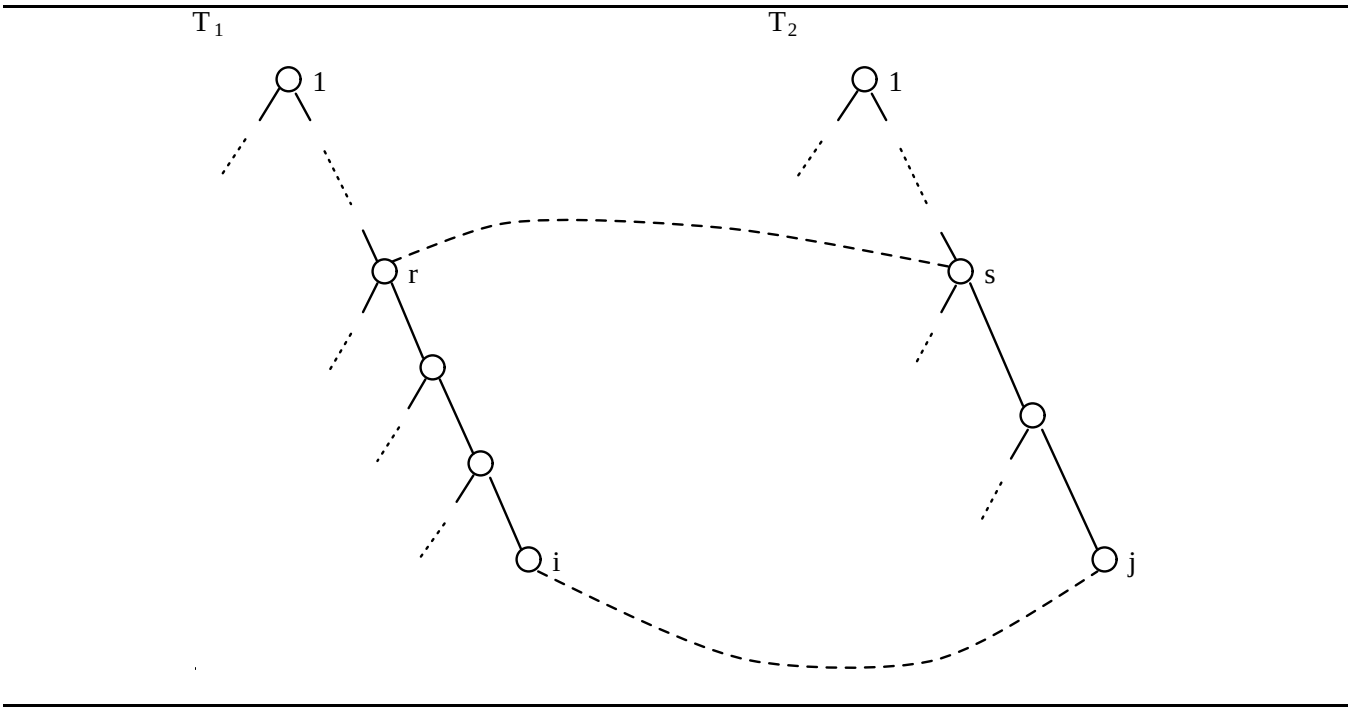

Figure 14.2: The difficult case

However in the optimal edit sequence for $T_1[1..i-1]$ and $T_2[1..j-1]$ we may not find such a pair $t_1[r]$ and $t_2[s]$. This means that in general we cannot derive $D_t(T_1[1..i], T_2[1..j])$ from $D_t(T_1[1..i-1], T_2[1..j-1])$.

In order to deal with this difficulty, Tai introduces another two measures between trees and the resulting algorithm is quite complex with a time and space complexity of $O(|T_1| \times |T_2| \times depth(T_1)^2 \times depth(T_2)^2)$.

Lu's algorithm

Another edit distance algorithm between ordered trees is reported by Lu. Lu's definition of the edit operations are the same as Tai's. However the algorithm given by Lu does not compute the edit distance as it defined. Nevertheless it does provide another edit based distance.

We will briefly discuss this algorithm and show its properties. Let $t_1[i_1], t_1[i_2], ..., t_1[i_{n_i}]$ be the children of $t_1[i]$ and $t_2[j_1], t_2[j_2], ..., t_2[j_{n_j}]$ be the children of $t_2[j]$. the algorithm consider the following three cases.

1. $t_1[i]$ is deleted. In this case the distance would be to match $T_2[j]$ to one of the subtrees of $t_1[i]$ and then to delete all the rest of the subtrees.

2. $t_2[j]$ is inserted. In this case the distance would be to match $T_1[i]$ to one of the subtrees of $t_2[j]$ and then insert all the rest of the subtrees.

Suppose each node label is a symbol chosen from an alphabet $\sum$. Let λ, a unique symbol not in $\sum$, denote the null symbol. We represent an edit operation as $a \to b$, where a is either λ or a label of a node in tree T_1 and b is either λ or a label of a node in tree T_2. We call $a \to b$ a change operation if $a \neq \lambda$ and $b \neq \lambda$; a delete operation if $b = \lambda$; and an insert operation if $a = \lambda$. Let T_2 be the tree that results from the application of an edit operation $a \to b$ to tree T_1; this is written $T_1 \to T_2$ via $a \to b$.

Let S be a sequence $s_1, ..., s_k$ of edit operations. An S-derivation from tree A to tree B is a sequence of trees $A_0, ..., A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \to A_i$ via s_i for $1 \leq i \leq k$. Let γ be a cost function which assigns to each edit operation $a \to b$ a nonnegative real number $\gamma(a \to b)$.

We constrain γ to be a distance metric. That is, i) $\gamma(a \to b) \geq 0$, $\gamma(a \to a) = 0$; ii) $\gamma(a \to b) = \gamma(b \to a)$; and iii) $\gamma(a \to c) \leq \gamma(a \to b) + \gamma(b \to c)$.

We extend γ to the sequence of edit operations S by letting $\gamma(S) = \sum_{i=1}^{|S|} \gamma(s_i)$.

Edit and alignment distances

The *edit distance* between two trees is defined by considering the minimum cost edit operations sequence that transforms one tree to the other. Formally the edit distance between T_1 and T_2 is defined as:

$D_e(T_1, T_2) = \min_S \{\gamma(S) \mid S \text{ is an ed it operation sequence taking } T_1 \text{ to } T_2\}$.

The *alignment distance* between two trees is defined by considering only those edit operation sequences such that all the insertions precede all the deletions. The reason why this is called alignment distance will be clear when we discuss it later.

Note that *edit distance* is in fact a distance metric while *alignment distance* is not since it does not satisfy the triangle inequality.

14.2.4 Early history of approximate tree matching algorithms

Tai's classical

Kuo-Chung Tai gave the definition of the edit distance between ordered labeled trees and the first non-exponential algorithm to compute it. The algorithm is quite complicated, making it hard to understand and to implement. The space complexity is too large to be practical. We sketch the algorithm here.

Tai used preorder number to number the trees. The convenient aspect of this notation is that for any $i, 1 \leq i \leq |T|$, nodes from $T[1]$ to $T[i]$ is a tree rooted at $T[1]$.

Given two trees T_1 and T_2, let $D_t(T_1[1..i], T_2[1..j])$ be the edit distance between $T_1[1]$ to $T_1[i]$ and $T_2[1]$ to $T_2[j]$.

We can now use the same approach as in sequence editing. Assume that $D_t(T_1[1..i-1], T_2[1..j-1])$, $D_t(T_1[1..i-1], T_2[1..j])$ and $D_t(T_1[1..i], T_2[1..j-1])$ are already known, we would like to extend them into $D_t(T_1[1..i], T_2[1..j])$. If either $t_1[i]$ or $t_2[j]$ is not involved in a substitution, then it is exactly the same as in sequence editing. That is, we just need to use either $D_t(T_1[1..i-1], T_2[1..j])$ or $D_t(T_1[1..i], T_2[1..j-1])$ plus the cost of deleting $t_1[i]$ or inserting $t_2[j]$.

The difficult case occurs when we substitute $t_1[i]$ by $t_2[j]$. In this case, there must be $t_1[r]$ and $t_2[s]$ such that $t_1[r]$ is an ancestor of $t_1[i]$, $t_2[s]$ is an ancestor of $t_2[j]$, and we substitute $t_1[r]$ by

The basic algorithm for pattern matching with variables is due to Hoffman and O'Donnell. Improvements using better data structures or variations of the algorithm have been proposed by Chase and Cai, Paige and Tarjan. Recent work by Thorup presents a short algorithm (with a rather subtle amortized analysis) that improves the space complexity and usually the time complexity for preprocessing simple patterns of size p to $O(p \log p)$ time and $O(p)$ space.

14.2.3 Edit operations and edit distance

Edit operations

We consider three kinds of operations for ordered labeled trees. *Changing* a node n means changing the label on n. *Deleting* a node n means making the children of n become the children of the parent of n and then removing n. *Inserting* is the complement of deleting. This means that inserting n as the child of m will make n the parent of a *consecutive subsequence* of the current children of m.

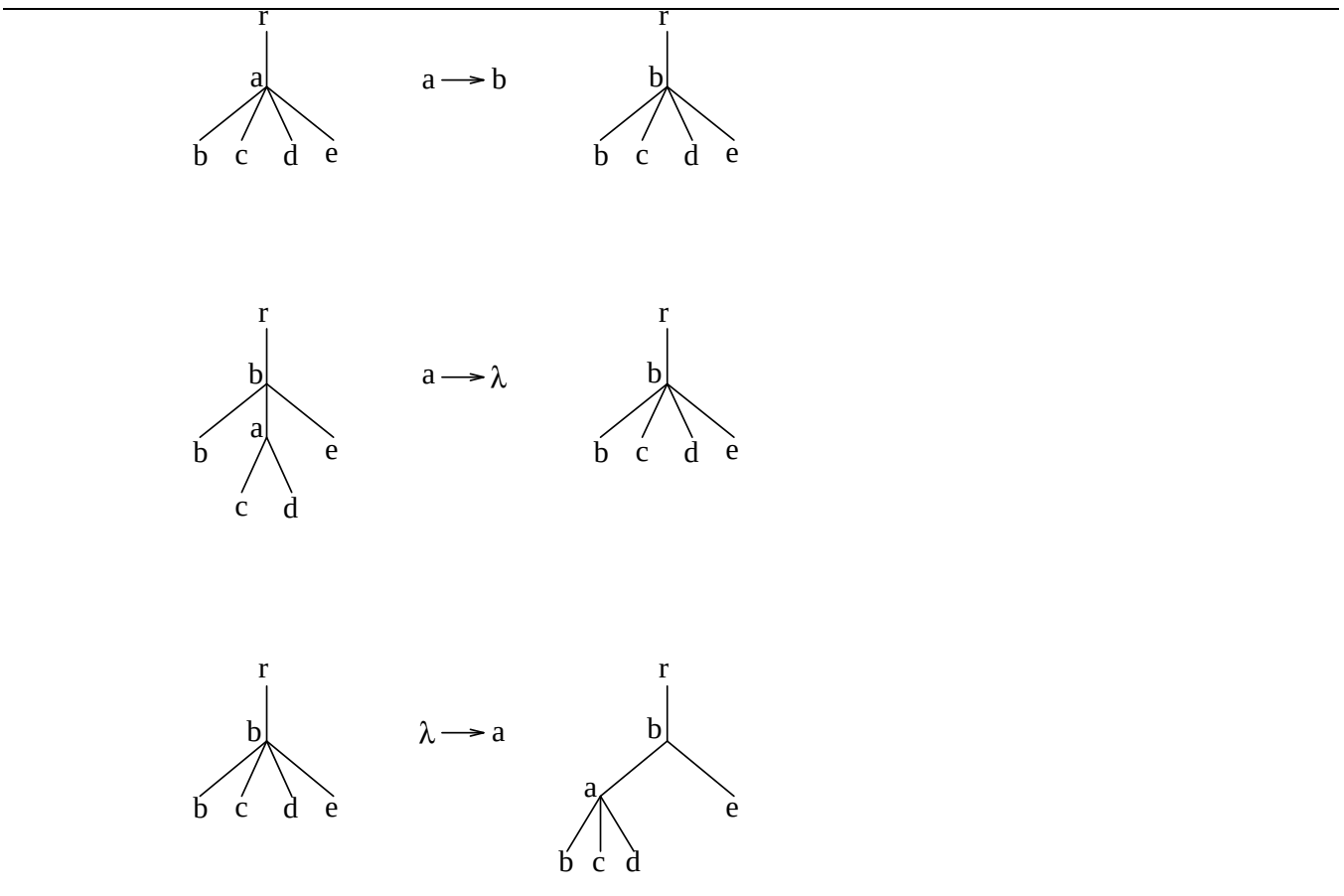

Figure 14.1: Edit operations

We can consider the same kind of operations for unordered labeled trees. In this case, in the insertion operation, we have to change *consecutive subsequence* to *subset*.

The obvious algorithm takes $O(nm)$ time. A classic open problem was whether this bound could be improved. Kosaraju broke the $O(nm)$ barrier for this problem with an $\tilde{O}(nm^{0.75})$ algorithm. (Note that $\tilde{O}(f(n,m)) = O(f(n,m)polylog(m))$) He introduced three new techniques: a suffix tree of a tree; the convolution of a tree and a string; and partitioning of trees into chains and anti-chains. More recently, Dubine, Galil and Magen improved this result giving an $\tilde{O}(n\sqrt{m})$ algorithm. Their result was based on the use of "k-truncated" suffix trees that, roughly speaking, shorten the representation of paths from the root of the pattern P to descendants of the root to have length no more than k. They also used periodical strings. (A string α is a *period* of a string β if β is a prefix of α^k for some $k > 0$.)

Dubiner, Galil and Magen first construct a $3\sqrt{m}$-truncated suffix tree, Σ, in $O(m\sqrt{m})$ time. Depending on how many leaves Σ has, there are two cases:

- Σ has at least $\sqrt{m}$ leaves. They show that there are at most $n/\sqrt{m}$ "possible roots" in the target tree. They can find these "possible roots" and then check to see if there is a match in $O(n\sqrt{n})$ time.
- Σ has at most $\sqrt{m}$ leaves. They show that by using the properties of periodical strings, a matching can be found in $\tilde{O}(n\sqrt{m})$ time. This gives an $\tilde{O}(n\sqrt{m})$ time algorithm.

Exact pattern matching with variables

Exact pattern matching has many applications in term-rewriting systems, code generation, and logic programming, particularly as a restricted form of unification. In this application, patterns are constructed recursively from a single "wild-card" variable v, a constant c, or a function $f(p_1, ..., p_k)$ where the arguments $p_1, ..., p_k$ are patterns in the language. Thus, v, $f(c)$, $f(f(v), c, v)$ are all patterns. The recursion induces a tree: the expression $f(p_1, ..., p_k)$ is the parent of the arguments $p_1, ..., p_k$ and the p_i's are children or "subpatterns" of $f(p_1, ..., p_k)$.

Pattern p_1 *matches* p_2 if it is "more general" (i.e., $\geq$) than p_2. This holds if either

1. p_1 is v or
2. p_1 is $f(x_1, ..., x_k)$, p_2 is $f(y_1, ..., y_k)$ and $x_i \geq y_i$ for all i between 1 and k inclusive.

Note that this allows a variable to match an entire subtree, but if p_1 isn't a variable, then p_1 and p_2 must begin with the same function symbol.

Given a set of patterns P and a "subject" pattern t, the *multi-pattern matching problem* is to find the set of elements in P which match some subpattern (i.e., subtree) in t.

There are two approaches to this problem: algorithms that start from the roots of the trees (top-down) and those that start from the leaves (bottom-up). The bottom-up approaches require significant preprocessing time of the patterns, but handle each subject faster (in time proportional to the size of the subject plus the number of matches). In rewriting systems, the subject is constantly changing, so bottom-up is more attractive. However, in the *development* of rewriting systems and in the construction of conventional compilers (which use pattern matching in back-end code generation phases), patterns change frequently. Once the compiler is constructed, the patterns become static.

The basic technique in the bottom-up algorithms is to construct the set PF of all subpatterns of P. Since this can be exponential in the size of P, the auxilliary space and time requirement can be large and much effort has gone into finding good data structures to hold this set.

14.2 Preliminary definitions and early history

14.2.1 Trees

A *free tree* is a connected, acyclic, undirected graph. A *rooted tree* is a free tree in which one of the vertices is distinguished from the others and is called the root. We refer to a vertex of a rooted tree as a *node* of the tree. An *unordered tree* is just a rooted tree. We use the term unordered tree to distinguish it from the rooted, ordered tree defined below. An *ordered tree* is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then we can designate them as the first child, the second child, and so on up to the kth child.

Unless otherwise stated, all trees we consider are either ordered labeled rooted trees or unordered labeled rooted trees.

Given a tree, it is usually convenient to use a *numbering* to refer to the nodes of the tree. For an ordered tree T, the left-to-right postorder numbering or left-to-right preorder numbering are often used to number the nodes of T from 1 to $|T|$, the size of tree T. For an unordered tree, we can fix an arbitrary order for each of the node in the tree and then use left-to-right postorder numbering or left-to-right preorder numbering. Suppose that we have a numbering for each tree. Let $t[i]$ be the ith node of tree T in the given numbering. We use $T[i]$ to denote the subtree rooted at $t[i]$.

14.2.2 A brief review of algorithmic results in exact tree matching

We distinguish between exact and approximate matching as follows. A match between two objects o and o' is *exact based on a a matching relation* R if o' is a member of $R(o)$. It is in this sense, in strings, that *w*ing* matches both "willing" and "windsurfing" where R is defined so that * can match any sequence of non-blank characters. A match between two objects o and o' given a matching relation R is inexact or approximate if it isn't exact. For example, *w*ing* matches "widen" only approximately. In the case of an approximate match, the *distance* is normally based on some monotonic function of the smallest changes to o and/or o' that result in objects p and p' respectively such that p' is a member of $R(p')$. Using edit distance *w*ing* matches "widen" with distance three, the number of changes to "widen" to transform it to "wing."

Most of our work has concerned approximate matching in trees, so our review of the results of exact matching in trees is extremely brief, serving mostly to give pointers to some of the important papers with the barest hint of algorithmic idea.

Exact tree matching without variables

Let pattern P and target T be ordered labeled trees of size m and n respectively, P matches T at node v if there exists a one-to-one mapping from the nodes of P into the nodes of T such that

1. the root of P maps to v,
2. if x maps to y, then x and y have the same labels, and
3. if x maps to y and x is not a leaf, then the ith child of x maps to the ith child of y. (This does not imply that P maps to the subtree rooted at v, but merely that the degree of y is no less than the degree of x.)

Chapter 14

Approximate Tree Pattern Matching

Dennis Shasha and Kaizhong Zhang

14.1 Introduction

Most of this book is about stringology, the study of strings. So why this chapter on trees? Why not graphs or geometry or something else? First, trees generalize strings in a very direct sense: a string is simply a tree with a single leaf. This has the unsurprising consequence that many of our algorithms specialize to strings and the happy consequence that some of those algorithms are as efficient as the best string algorithms.

From the point of view of "treeology", there is the additional pragmatic advantage of this relationship between trees and strings: some techniques from strings carry over to trees, e.g., suffix trees, and others show promise though we don't know of work that exploits it. So, treeology provides a good example area for applications of stringologic techniques.

Second, some of our friends in stringology may wonder whether there is some easy reduction that can take any tree edit problem, map it to strings, solve it in the string domain and then map it back. We don't believe there is, because, as you will see, tree editing seems inherently to have more data dependence than string editing. (Specifically, the dynamic programming approach to string editing is always a local operation depending on the left, upper, and upper left neighbor of a cell. In tree editing, the upper left neighbor is usually irrelevant — instead the relevant cell depends on the tree topology.) That is a belief not a theorem, so we would like to state right at the outset the key open problem of treeology: can all tree edit problems on ordered trees (trees where the order among the siblings matters) be reduced efficiently to string edit problems and back again? [1]

The rest of this article proceeds on the assumption that this question has a negative response. In particular, we discuss the best known algorithms for tree editing and several variations having to do with subtree removal, variable length don't cares, and alignment. We discuss both sequential and parallel algorithms. We present negative results having to do with unordered trees (trees whose sibling order is arbitrary) and a few approximation algorithms. Finally, we discuss the problem of finding commonalities among a set of trees.

[1] Since the editing problem for unordered trees is NP-complete, we can say that it is not possible to map it into a string problem.