RAPID SUBTREE IDENTIFICATION REVISITED

M.H. Overmars and J. van Leeuwen

RUU-CS-79-3

February 1979

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

RAPID SUBTREE IDENTIFICATION REVISITED

Mark H. Overmars and Jan van Leeuwen

technical report RUU-CS-79-3

February 1979

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht, the Netherlands

all correspondence to

Dr. Jan van Leeuwen
Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

RAPID SUBTREE IDENTIFICATION REVISITED

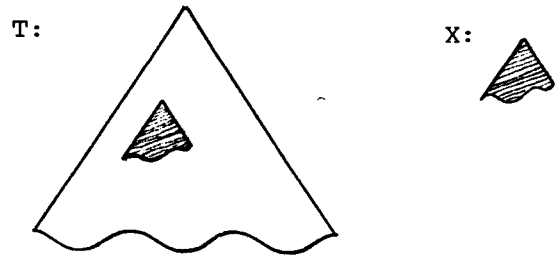
Mark H. Overmars and Jan van Leeuwen

Abstract. We generalize the pattern matching problem for strings to lexicographic trees. Some old algorithms of Karp, Miller and Rosenberg for this problem are reconsidered and analysed, and in some instances considerably improved. A new, efficient algorithm for subtree matching is presented which runs in O(n.V) steps on a reference machine, where n is the number of nodes in the object-tree and V is the number of distinct levels at which leaves occur in the pattern-tree. If a mild form of memory address calculation is permitted, then one can implement the algorithm to run in even fewer steps.

1. Introduction

The ordinary pattern matching problem asks for determining all occurrences (or just the first) of some pattern x in a text t. It is an important problem for all applications which need textfiles to be searched frequently, like text-editors and catalogue searching programs. Pattern matching became an issue in complexity theory a few years ago when Knuth, Morris and Pratt [7] showed how to perform it in only $O(|t| + |x|)$ steps.

We generalize the problem to trees, i.e. we ask for determining all occurrences of a "pattern" X as an embedded subtree of a tree T.



We allow the branches of X and T (but not the nodes) to be labeled with single symbols from some finite, ordered alphabet Σ of cardinality k and we assume (for convenience) that the ordering of labels from left to right on the edges emanating from any single node is consistent with the ordering of the alphabet. Hence trees can always be viewed as lexicographic trees (see Knuth [6]). Binary trees fit in when we label left branches with "0" and

right branches with "1". We shall refer to the problem stated as "subtree matching". Note that we do not require a matching subtree to include áll edges that emanate from the constituent nodes in T!

Subtree matching certainly is an intriguing algorithmic problem in itself, but there are several reasons why it is of wider interest. A specific problem which our algorithms for subtree matching can handle concerns the "simultaneous" identification of any one of several patterns $\{x_1,\ldots,x_l\}$ in a collection of texts $\{t_1,\ldots,t_m\}$. It is nót solved by pure subtree matching of the lexicographic tree X of all patterns in the lexicographic tree T of all texts, as one readily observes. However, the subtree matching algorithms we shall present happen to identify all paths of X (from leaves to root) in the tree T before assembling them into "subtrees" and that is what we need here. The problem of matching many patterns in a single text was considered before by Aho and Corasick [1] (see also [7]). Our algorithms further generalize, in a way, their extension of the KMP-algorithm.

Subtree matching was originally proposed by Karp, Miller and Rosenberg [5], along with several other pattern matching problems (like finding "boxes" in a multidimensional array). Their paper appeared soon after a fore-runner of the KMP-algorithm had been invented (Morris and Pratt [9], Knuth and Pratt [8]). However, we must assume that the idea of pattern-sliding as in the KMP-algorithm was not fully grasped at the time. Instead, a new technique making use of so-called "s-equivalences" (defined in sections 2 and 3) was introduced and exploited to answer a variety of problems, ranging from simple matching to the classification of áll patterns occurring in a text or tree. Informally, positions (in a text) or nodes (in a tree) are s-equivalent if the length s substrings or paths ending at these positions or nodes (respectively) are identical. The resulting algorithms for string- and subtree-matching required a good deal of preconditioning (which we now know to be largely unnecessary) and were in a few cases even too complicated for what they wanted to achieve (as we shall see). The main impetus behind this paper was to reconsider and, if possible, improve the old algorithms proposed in [5] in order to find out when it might still be viable alternatives to KMP-type matching (for strings and trees).

In section 2 we shall review a practical variant of the KMP-algorithm for texts over a small alphabet (e.g. with $k \leq 256$, that is, with characters fitting in an 8-bit byte). We shall also review an improved $O(n \log s)$ construction of the s-equivalence classes for texts of length n and present a new, efficient algorithm to actually build the s-equivalence classes on-line.

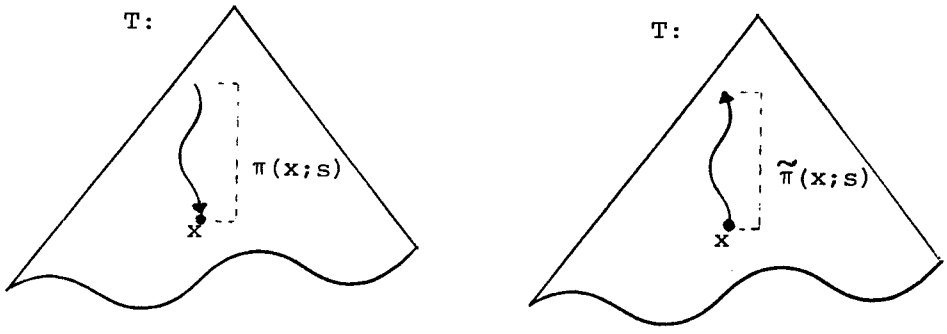
For small patterns it implies a simple, linear time matching algorithm, somewhat space-demanding but with an interesting side-effect (nót mentioned in [5]): if certain tabular information is retained with the text after scanning it for a pattern of length s once, then any later request for locating a pattern x of length $\leq s$ can be answered within $O(|x|)$ steps only.

In section 3 we first show a greatly simplified version of a subtree matching method essentially due to Karp, Miller and Rosenberg [5] (although we found no need for using s-equivalences here). We prove that it runs in O(nd) steps, where n is the number of nodes in T and d the depth of X. The algorithm contains an important accounting technique which will permeate all later algorithms. Next we show how the construction of s-equivalences can be extended to trees, withóut the assumption of free ancestor calculation made in [5]. Expanding on the use of s-equivalences, we eventually arrive at a subtree matching algorithm that runs in O(n.p) steps ... provided a considerable amount of preconditioning is performed. Here n is as before and p is the largest number of suffixes a single leaf-path of X can have which are leaf-paths of X themselves also.

In section 4 we finally show how the idea of the KMP-algorithm can be used to obtain an efficient subtree matcher, which avoids the need for retaining the mass of auxiliary information that drove the Karp-Miller-Rosenberg-type algorithms. The only difficulty which can slow down the algorithm appears to be the computation of ancestors. One implementation proves that subtree matching can be performed in O(n.V) steps on a reference machine, where n is as before and V is the number of different levels of X at which leaves occur. If we allow addresses to be manipulated as numbers (addition and subtraction are sufficient) and "random access" is permitted, then one can implement the algorithm to run in only O(n.p) steps. Hence we achieve the best of the Karp-Miller-Rosenberg-type of approach without having to resort to preconditioning.

Lastly, we comment on the application of our algorithm to the problem of matching "several" patterns together in one string in linear time "plus a constant for each match found" (compare Aho and Corasick [1]).

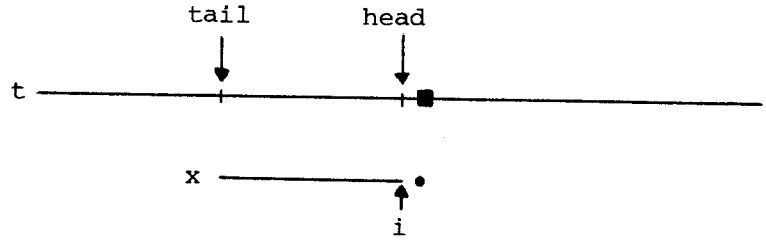
Let $\mu(T)$ denote the total number of nodes of some tree T. Given a tree T and some node $x \in T$, we shall use $\pi(x;s)$ to denote the length s path leading "into" x from the direction of the root (padded with #'s up front, if necessary, when the root is less than s edges away) and $\tilde{\pi}(x;s)$ to denote its reverse.



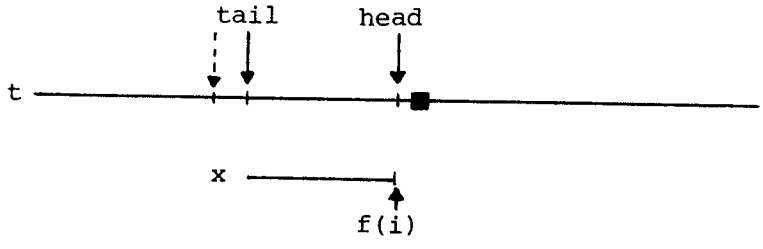
We shall simply write $\pi(x)$ (or $\tilde{\pi}(x)$) to denote the path from the root down to x (or its reverse, respectively).

2. Revisiting pattern matching for texts.

We shall outline the idea of the KMP-algorithm and describe a fast, practical variant of it. Suppose we tried matching the pattern at position tail in the text and succeeded up to position head (and i in the pattern). Suppose the next symbols do nót match.

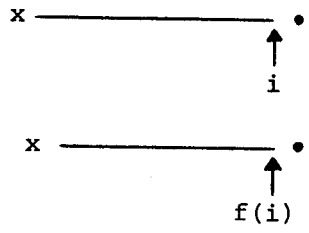


Slide x forward over a smallest possible distance so that from the new location of tail onward, text and pattern match again up to position head (and f(i) in the pattern). The "recovery" function f



can be precomputed (we ignore how) on the basis of the pattern alone. If the next symbols match we move the "front" pointers one position right, otherwise we must make another recovery attempt. At each step either tail moves right or head moves right, and we can charge the cost of the step to the appropriate pointer-move. Hence, when f is tabulated, the pattern match can be performed in only $O(|t|)$ steps.

Using f we can be certain that recovery must be repeated when the symbols following x_i and $x_{f(i)}$ in the pattern match (check!)

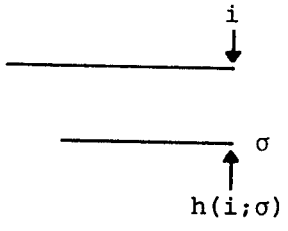


and we must apply f as many times as needed to get a symbol different from • in the matching position. (It may be impossible, in which case a new match must start from position head + 2). We'd better precompute these steps also, obtaining a much improved recovery function g. It is the function used in the KMP-algorithm. We may still have to call on g repeatedly before a match with t_{head+1} appears up front (if it ever will), although Pratt (see [7]) showed one would never have to "idle" for more than $O(\log|x|)$ steps this way.

For small alphabets it is feasible to precompute an even better recovery function h, which eliminates the need for repeated shifting of the pattern at one head-position entirely.

Definition. For a pattern x let $h(i;\sigma)$ be the largest index $< i$ such that $x[1:h(i;\sigma)]$ is a suffix of $x[1:i]$ and $x_{h(i;\sigma)+1} = \sigma$, or -1 if no such index exists.

A picture will clarify the definition. Function h tells us how to shift

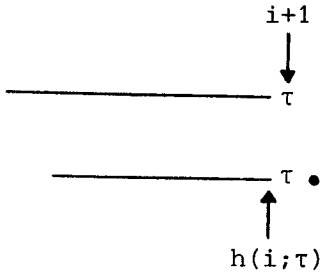


by the least amount to have a σ in the matching position. Clearly, when a mismatch between text and pattern occurs, we recover by $i := h(i;t_{head+1})$ in one move! One can tabulate h using i and the character-code of σ (i.e. its byte taken as a number) for addressing. The calculation of h is easy.

Proposition 2.1. The h-table can be computed in $O(|x|)$ steps, assuming the alphabet size is a fixed constant.

Proof

Suppose h-values are computed up to i. Let t_{i+1} be τ. Considering the situation at i, it will be clear that



$$h(i+1;\sigma) = \begin{cases} h(i;\tau)+1 \text{ when } x_{h(i,\tau)+2} = \sigma, \\ h(h(i;\tau)+1;\sigma) \text{ otherwise.} \end{cases}$$

As $h(i;\tau) < i$ and (hence) $h(i;\tau)+1 \leq i$, we need to look up at most 2 previously computed h-values to obtain another $h(i+1;\sigma)$. The total time spent will be about $2.k.|x|$, which is $O(|x|)$.

■

Some practical implementations of the KMP-algorithm we know use an h-type function for recovery rather than g, for the greater speed while matching over the text (cf. van der Steen [13]). We shall encounter a generalization of h in the later subtree matching algorithm.

Karp, Miller and Rosenberg [5] approach the text matching problem in an entirely different manner. They primarily aim at making a full inventory of all substrings of a certain length s which occur in t and then consider, after performing the routine for $s = |x|$, whether the pattern x is among them. Such an ambitious program quite obviously is more time-consuming than KMP for a single match but, as we observe, when the inventory is kept in store in a suitable form, any later match with a pattern x' of length $\leq |x|$ will take only $O(|x'|)$ steps to perform. Hence there is reason to reconsider and further explore the potentials of this kind of approach.

Definition. Positions i and j of a text t are s-equivalent if and only if $t[i-s+1:i] = t[j-s+1:j]$, where t is padded with #'s on the left to accomodate for negative index positions.

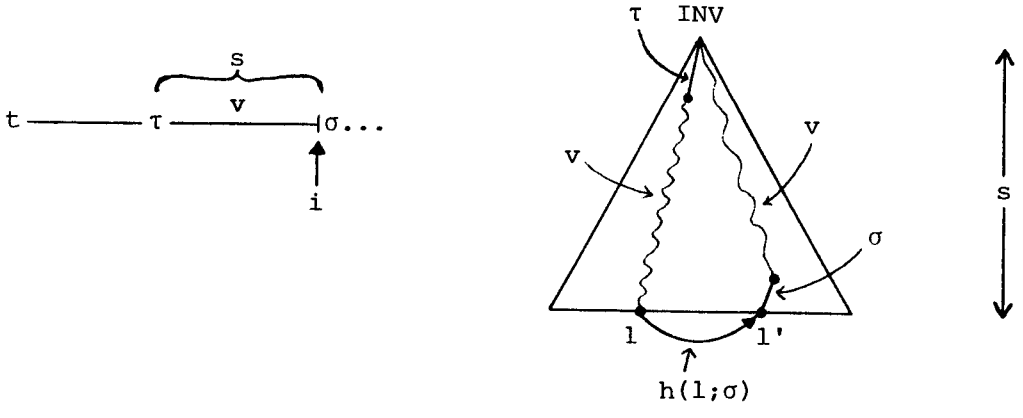
Thus, i and j are s-equivalent if and only if the length s substrings ending at positions i and j are identical. We could have classified positions according to the substrings starting there (as in [5]), but the present version will have certain notational advantages later. Karp, Miller and Rosenberg [5] argued that

Theorem 2.2. The s-equivalence classes can be constructed on a random access machine in O(n log s) steps.

The construction was reconsidered and given with a more feasible implementation in van Leeuwen [14]. Succinct disadvantages are that the construction gives more than we asked for (namely, a variety of s'-equivalences with $s' < s$ which we don't need) and that it is very definitely not on-line, i.e. we do not have the classes of t[1:i] completed as we scan t for the first time with i moving from 1 to n. To circumvent these practical deficiences we develop a new algorithm for s-equivalences.

The idea is to enter substrings of length s encountered in t in a lexicographic tree INV (for inventory). Each leaf l, at the end of the path defining the string $\pi(l)$ down the tree, contains a pointer "pos" to a linked (ordered) list of all positions i in t where an occurrence of $\pi(l)$ ends. Obviously, the naive algorithm of entering all length s substrings one by one will build INV in O(n.s) steps.

At the expense of augmenting the leaves with a few more pointers, the algorithm can be speeded up. Consider a typical step of the naive algorithm.



Let τv be the length s substring ending at i and let the corresponding leaf in INV be l. To process the next symbol σ of t we must "move" to a new leaf l', corresponding to vσ. Enter vσ in the tree, or trace the path if vσ has occurred before, in order to locate l'. Save the connection from l to l' as a

pointer $h(l;\sigma)$ at l, in order that we can make the transition at once when the same situation comes up again.

Let Q be the total number of s-equivalence classes in t (which will be the number of leaves in INV).

Lemma 2.3. INV can be build in time proportional to $|t| + s.Q$

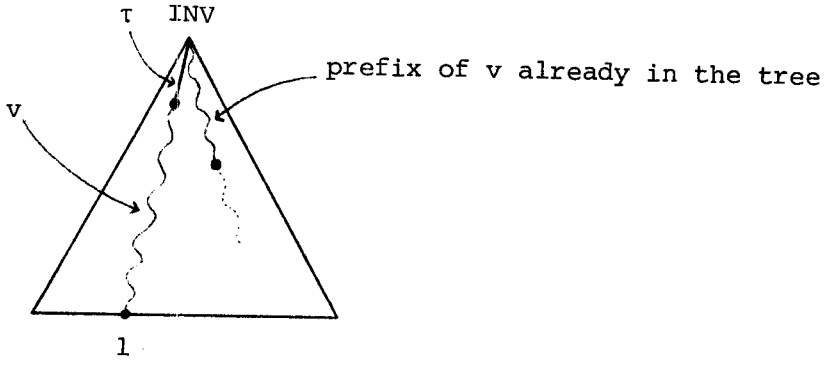
Proof

Proceed as sketched above. A new length s substring must be entered (or traced) whenever the appropriate h-pointer at the latest leaf l reached was still undefined. Otherwise we just follow a defined pointer and are done in constant time.

Since leaves can have no more than k defined pointers, we need to do the $O(s)$ amount of work for entering (or tracing) a substring at most $k.Q$ times. All remaining steps can be charged to the positions of t as they are passed. Thus we do at most $O(|t| + s.k.Q)$ steps, which is $O(|t| + s.Q)$.

■

Entering new substrings of length s is too expensive as it stands. Consider what we do when $v\sigma$ is entered, when we are at l.



We would discover that some initial portion of v is already in the tree and would waste time tracing it. Couldn't we immediately "jump" to the node where the actual work of creating new nodes to accomodate the tail of $v\sigma$ begins.

We solve the problem by changing the pointer-structure imposed on INV. At each node α (nót the root) allow for a pointer $h(\alpha)$ to the unique node β with $\pi(\beta)$ equal to "$\pi(\alpha)$ with the front symbol removed", or to "nil" when β does not exist in the tree at present. Sons α of the root obviously have $h(\alpha)$ = root. We observe

Proposition 2.4. When h is defined at node α, then so it is at all ancestors of α (minus the root).

We shall make sure that the property expressed in 2.4 is maintained as the construction of INV progresses.

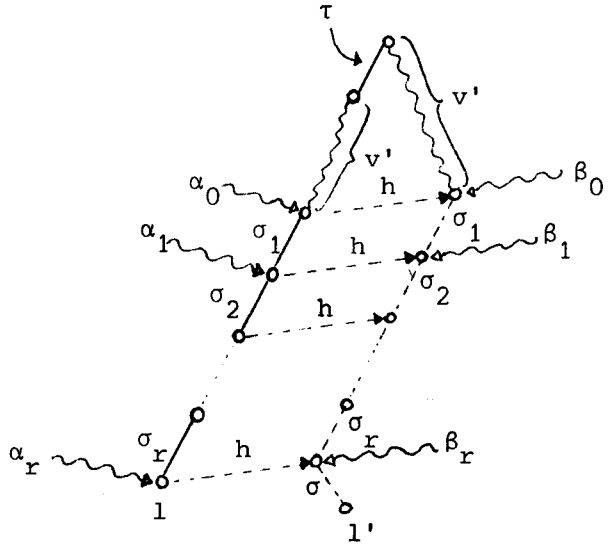
Theorem 2.5. INV can be build in time proportional to $|t| + \mu(\text{INV})$.

Proof

Suppose we are at leaf 1 (as sketched) and must move on σ. We show how to avoid redundant tracing in "entering" the new length s substring $v\sigma$, and the position where it occurs, in INV.

When h(1) is defined, we only need to follow (or ... add) the son-link labeled σ out of it. Ultimately, more and more h-values at leaves will be defined and we can move with only a constant amount of work per step (charged to t-positions).

When h(1) is undefined we proceed as follows. Climb the path from 1 towards the root until you hit the first node α_0 with $h(\alpha)$ defined. (Note that you may get as close as one edge from the root.) Let the nodes passed on the way



(note that the two paths shown need not be disjoint)

be $1 = \alpha_r, \ldots, \alpha_1, \alpha_0$. Consequently, substring v can be decomposed as $v = v'\sigma_1 \ldots \sigma_r$ where v' is the longest prefix of v that already exists as a root-path in the tree and $\tau v' = \pi(\alpha_0)$. Let $h(\alpha_0)$ be β_0.

Now enter $\sigma_1 \ldots \sigma_r$, beginning at β_0 and passing through nodes β_i and labeled edges, creating new records for those which didn't already exist. Put $h(\alpha_i) = \beta_i$ as i moves from 1 to r. Make sure to put $h(\beta_1) =$ root when β_0 is the

root. When β_r has been reached, we do one more step with σ to obtain the new leaf l' for $v\sigma$.

The work spent when h(l) is undefined is easily charged to the nodes $\alpha_r,\ldots,\alpha_1$ i.e. to nodes with h undefined for which h becomes defined immediately after. No node needs to be charged more than a constant. Hence the total amount of work spend on steps of this sort is $O(\mu(INV))$.

■

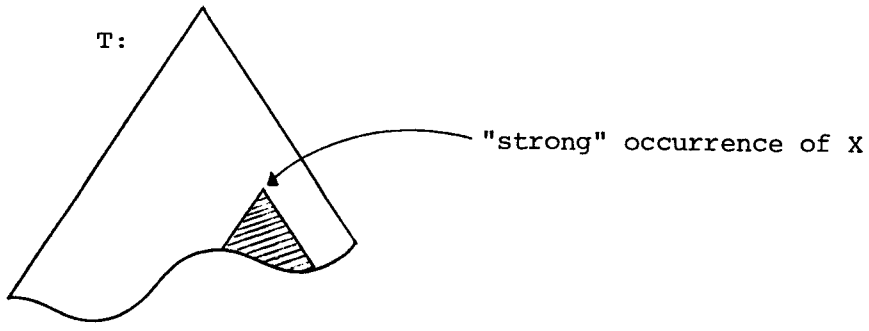
It is interesting to observe that the construction of 2.5 actually runs in time independent of k (and that it does not leave more than s nodes with their h-value undefined). The construction builds INV on-line, with the time-lag needed before a new σ can be processed obviously decreasing as INV approaches completion. It could be sped up, for instance, by maintaining two additional fields at each leaf l: one containing a pointer to its lowest ancestor α_0 with a defined h-value and one containing a number indicating by how much the scanner should be reset on τv when we jump to α_0 and wish to enter the appropriate tail of v (or, rather, of $v\sigma$). Nevertheless the construction must be optimal in order of magnitude as it stands, since we shall have to reference each position of t and each node of INV at least once in any conceivable (general) algorithm for building INV from t. (Formally, it would require a proof in the spirit of [11].)

Locating patterns x of any length $\leq s$ in t is easy once INV has been build. It need take no more than $|x|$ steps to decide whether x occurs in t at all! The method will compete with the KMP-algorithm in time (but not in space) for a single match as long as $\mu(INV)$ remains "small", for instance when $s \leq \log|t|/\log k$.

3. Algorithms for subtree matching.

Let X and T be trees with m and n nodes ($m \leq n$), respectively. We shall assume that X and T are lexicographic trees, with the edge-labels chosen from some finite alphabet Σ. Hence, nodes can have at most k sons, where $k = \#\Sigma$. Let X have depth d.

We shall first give an efficient solution of a restricted subtree matching problem. "Strong subtree matching" asks for locating all nodes in T where X occurs as the full subtree down to the frontier of T.



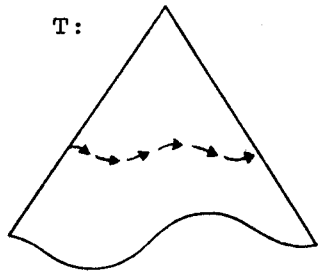
Theorem 3.1. Strong subtree matching can be done in linear time.

Proof

We shall give two, different solutions a and b.

Solution a.

Traverse X once to determine its depth d. Clearly, only those nodes α of T which have depth d also, qualify as root-candidates for a "strong" occurrence of X. Now traverse T in pre-order to determine the depth of each of its nodes. Don't bother keeping track of depths greater than d. Enter all nodes of depth d in a linked list, in the order in which they are discovered (which will be "from left to right"). If α and β are distinct nodes of the list, then



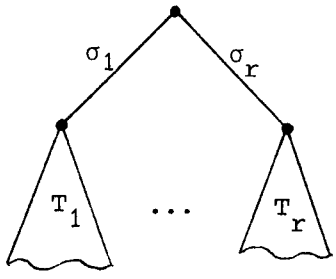
their subtrees must be disjoint. We can test which head an "X" by trying, for each α in the list, a simultaneous fixed-order traversal of its subtree and X. If the traversals "diverge" we know the subtree isn't "X", but if we keep stepping through like edges all the way it is! Testing all list-elements this way can be charged to the (say) pre-order traversal of the distinct depth-d subtrees of T, which is bounded by O(n). The total costs add up to O(n+m).

Solution b.

There is a simple trick which will reduce strong subtree matching to linear pattern matching. For trees T let str(T) be inductively defined as follows:

(i) if T consists of one node only, then str(T) = λ (the empty word)

(ii) if T is of the form (with $r \leq k$ and $\sigma_1 < \ldots < \sigma_r$)

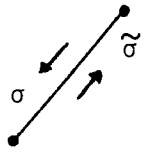


then $str(T) = \sigma_1\ str(T_1)\ \tilde{\sigma}_1\ \sigma_2 \ldots \sigma_r\ str(T_r)\ \tilde{\sigma}_r$.

Observe that str(T) can be determined symbol-after-symbol while traversing T in pre-order, recording "σ" when we go down an edge labeled σ and "$\tilde{\sigma}$" when we climb back on it. In this way each prefix of str(T) corresponds to some node of T. X is a strong subtree of T rooted at node α if and only if $str(T) = v\sigma\ str(X)\ \tilde{\sigma}\ w$ for certain v, w and σ, where $\sigma \in \Sigma$ (or both vσ and $\tilde{\sigma}w$ equal to λ) and vσ the "pre-order" path down to α.

It is now obvious how to apply a (fast) string matcher, e.g. the KMP-algorithm, to determine the strong occurrences of X in T. Note that we can leave str(X) and str(T) implicit. Pointers moving along str(X) or str(T) merely trace the path of the pre-order traversal algorithm in X or T (respectively). Whenever a strong occurrence of X is detected in the "T-string", the pointers "tail" and "head" (of the KMP-algorithm discussed in sect. 2) both point to the node α in T where it is attached. The strong occurrences of X are produced in the same left-to-right order as in solution a.

The algorithm will run in time O(n'+m') regardless of alphabet-size, where $n' = |str(T)|$ and $m' = |str(X)|$. Note that the symbols of str(T) can be grouped into pairs $\sigma, \tilde{\sigma}$ which each correspond to the traversal of a distinct σ-labeled



edge of T. Edges of T can be counted by the node they lead into. Hence n' = 2n and, likewise, m' = 2m. We conclude that this algorithm runs in O(n+m) also.

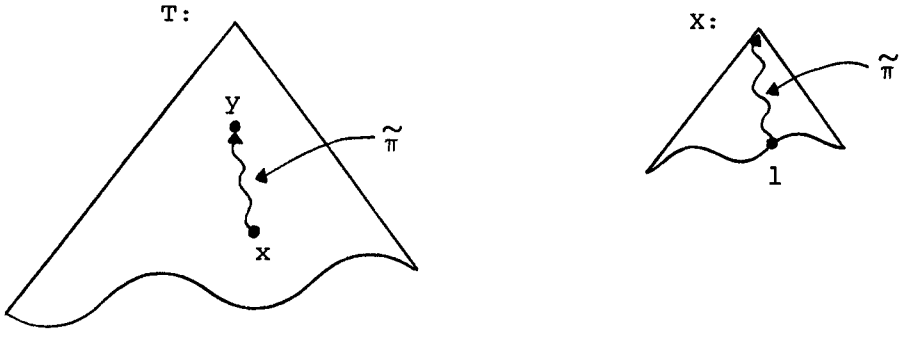
■

We proceed to the more general problem of finding all (weakly) embedded occurrences of X in T. We shall give an inefficient but pretty algorithm first, to illustrate an important accounting technique due to Karp, Miller and Rosenberg [5] which will be applied in all later, more efficient algorithms.

Let $\tilde{\pi}_1,\ldots,\tilde{\pi}_q$ be the leaf-to-root paths of X. Let $\tilde{X}$ be the lexicographic tree obtained by entering $\tilde{\pi}_1,\ldots,\tilde{\pi}_q$ at the root and marking each node where a $\tilde{\pi}$ "ends". Clearly all leaves of $\tilde{X}$ will get marked, but so will several internal nodes (in general). A somewhat similar "inversion" of trees is proposed in [5], where it is erroneously called an "anti-isomorphism". The algorithm we are about to describe presented itself when we tried to evaluate algorithms 4 and 5 of [5]. Much of what these algorithms did could be eliminated and we were left with the following not so efficient, yet elegant method.

Proposition 3.2. Subtree matching can be done in O(n.d) steps.

Proof



Let node y be called a "root-candidate" whenever there is an x "below" such that the path from x up to y coincides with some leaf-to-root path in X. Intuitively, when sufficiently many "distinct" x's contribute to y's candidacy then y must be the root of an embedded X.

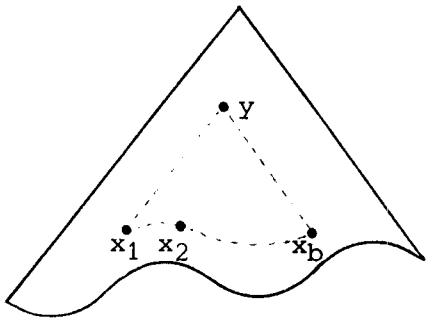
Set up a counter in each node of T. Initialize the counters at 0. For each node x, add 1 to the counter of every node y above it which is a root-candidate by virtue of the path from x to y. Let X have q leaves.

Claim. Node y is the root of an embedded X if and only if the final value of its counter is q.

Proof of claim.

First observe that when x contributes into y, then no node below it or above it can contribute into y also (for no leaf of X can be on the trail of another leaf of X). Hence, the value of y's counter is $\geq b$ if and only if there exist

"independent" nodes $x_1,\ldots,x_b$ below it which all "identify" distinct leaves of X (when we go by their relative position to y). Obviously, the "skeleton"

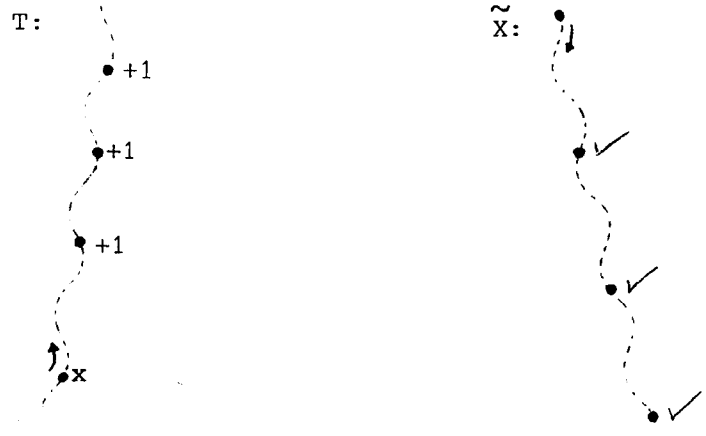


below y is a full X if and only if there is a representative for each leaf below it ... which happens exactly when y's counter gets up to q.

{End proof of claim}

Note from this argument that no counter can actually ever get lárger than q.

We are done when we show how to fill the counters. Here $\tilde{X}$ comes in handy. In order to determine which nodes y an x contributes into, start climbing T at x while moving down like edges in $\tilde{X}$ beginning at the root. Whenever you pass



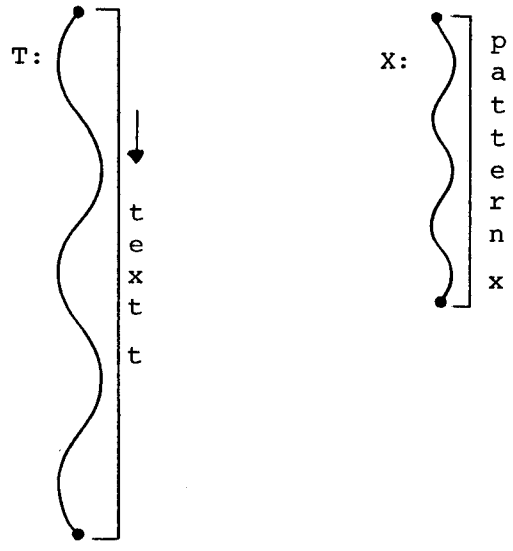
a marked node in $\tilde{X}$, add 1 to the counter of the node reached on your way up in T. Stop when you cannot move along like edges anymore. (In particular, one would stop when the root of T or a leaf of $\tilde{X}$ is reached.) It is easily verified that this will "add in" all contributions x can possibly make.

The algorithm could proceed as follows. First build $\tilde{X}$. Next, visit the nodes of T in some order once and carry out the accounting scheme. Finally, record all nodes in T whose counter has value q. The time-requirement is easily estimated. To build $\tilde{X}$, visit the leaves of X (and cóunt them) one at a time and

enter their leaf-path into $\tilde{X}$ while (simultaneously) traversing it in X. It will need no more than O(d) steps per leaf, or O(qd) total, to build $\tilde{X}$. Traversing T in pre-order will take O(n) and for each node visited we certainly need go no more than d steps up to do the necessary accounting. It will take at most O(nd) total. By the time one visits a node "for the last time" its counter will have its final value and one can immediately decide whether to list it or not. Alltogether it comes to O(n.d) steps, using that $q \leq m \ (\leq n)$.

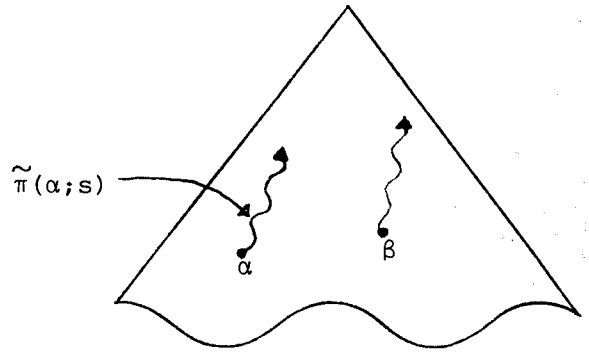
■

The method of 3.2 has some merits but lacks all further tuning. For the case of two "linear" trees, corresponding to single strings t and x



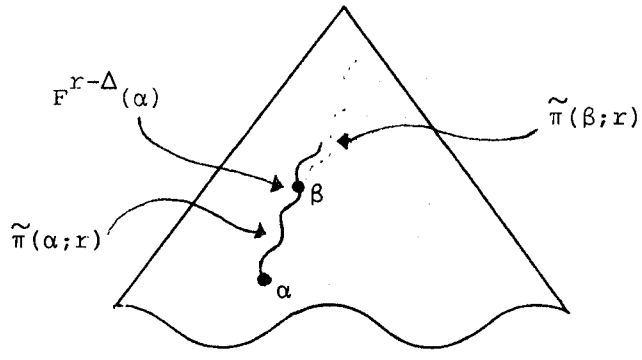
it degenerates to a version of the "naive", O(nm) solution of the pattern matching problem.

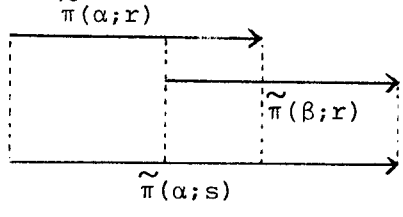
The given method makes it clear that we can solve the subtree matching problem by carefully correlating leaf-paths of X with identical path-segments (from some internal node upwards) in T. Karp, Miller and Rosenberg [5] made it into a formal equivalence relation on nodes, very much like the s-equivalence for positions in a text (see sect. 2). Let $\tilde{\pi}(\alpha;s)$ be the path-segment of length s (as a string over Σ) beginning at node α, leading up towards the root of the tree containing α. When α is closer than s steps from the root, pad $\tilde{\pi}(\alpha;s)$ with #'s.



Definition. Nodes α and β of a tree T are called s-equivalent if $\tilde{\pi}(\alpha;s) = \tilde{\pi}(\beta;s)$.

Karp, Miller and Rosenberg [5] argued that the s-equivalence classes for a tree T can be computed in O(n log s) steps, just as for strings. The idea is to compute the s-classes by "gluing" equivalent, connecting (perhaps indeed overlapping) front-end and tail-end segments of length r together (some r with $\frac{1}{2}s \le r < s$), where the necessary r-equivalences have been computed by a recursive application of the same technique first. Karp, Miller and Rosenberg did not make it explicit that we need to know the father $F^r(\alpha)$ of height r of a node α in order to glue $\tilde{\pi}(\alpha;r)$ to the right, connecting segment $\tilde{\pi}(\beta;r)$. When segments overlap by Δ we even need $F^{r-\Delta}(\alpha)$ instead of $F^r(\alpha)$.





We show how this can be organised at no significant extra costs.

Lemma 3.3. For any r one can determine the height r fathers in T (collectively) in only O(n) steps.

Proof

Let a pointer "tip" traverse T in pre-order. Let it mark its "life-line" from the root as it moves, maintaining it as a push-down store embedded in the tree. Have another pointer "toe" trail at a fixed distance r behind tip on the life-line. When tip moves down one step, then so does toe (along the life-line). When tip moves up, then toe climbs back an edge along the life-line. Clearly, whenever tip visits a node α, toe points at $F^r(\alpha)$. The entire procedure adds only constant time to each step of the ordinary pre-order traversal algorithm, and must run in O(n).

■

We sketch the necessary ingredients to obtain a feasible implementation of the construction of s-classes as envisioned by Karp, Miller and Rosenberg [5].

Theorem 3.4. The s-equivalence classes for a tree can be constructed in O(n log s) steps.

Proof

Let $c = \max\{i \mid 2^i \leq s\}$. Assume that $c > 0$, since there is nothing to prove otherwise. Build the $1,2,\ldots,2^c$-equivalence classes as for strings (see van Leeuwen [14]), making sure that, for each i, nodes α partitioned according to their $\tilde{\pi}(\alpha;2^i)$ carry a pointer to $F^{2^i}(\alpha)$ in an additional field G. When we build the 2^{i+1} equivalence classes from the 2^i equivalence classes, the additional information needed is maintained as follows. If $\tilde{\pi}(\alpha;2^i)$ is to be "doubled" we glue it onto $\tilde{\pi}(\alpha.G;2^i)$ and re-classify α as required by the algorithm and record $(\alpha.G).G$ in a back-up field. When the names of the new equivalence classes are assigned, we overwrite the G-field of each node with the contents of its back-up field. The extra work can be fully charged by adding a fixed constant to each step of the construction-algorithm. It remains O(n log s), or rather O(n.c).

When $2^c = s$ we are done, otherwise we must go through one more cycle. Since $\frac{1}{2}s < 2^c < s$ we can build the s-equivalence classes by reclassifying 2^c-equivalent nodes on the basis of the equivalence class their height $s-2^c$ father belongs to. It requires a computation of the necessary fathers first, which we can in linear time by 3.3. The final reclassification then proceeds in no more than O(n) steps also.

■

It is interesting to see how the technique of "gluing" equivalence relations together can be applied when we need to compute the s-equivalence classes for s equal to

$$s_1 < s_2 < \ldots < s_V.$$

The bound of $n \log s_1 + \ldots + n \log s_v \leq n.V.\log s_V$ obtained from calculating each of the s_i-equivalences separately, can certainly be improved. Here is one, simple technique. Construct the s_1 and $s_2-s_1,\ldots,s_V-s_{V-1}$ equivalences (recursively or ... directly). By gluing one can build the s_2-classes from the s_1 and s_2-s_1 classes, the s_3-classes from the s_2-classes (which we now know) and the s_3-s_2 classes, and so on until we have the s_V-classes as well. Each "composition" of two equivalences takes only O(n) to build. The total amount of work spend adds up to

$$n \log s_1 + n \log(s_2-s_1) + \ldots + n \log(s_V-s_{V-1}) + nV \leq n.V.\log \frac{s_V}{V} + nV = n.V.\log \frac{2s_V}{V}$$

(which is a valid bound even when $s_i - s_{i-1} = 1$ for some i). In general one should try and build the $s_1,\ldots,s_V$ equivalences from the elementary 1-classes by repeated "addition" of as small a number of likewise computed intermediate equivalences as possible. The problem is intimately related to the theory of addition chains for "many numbers" (see e.g. Yao [15], Pippenger [10]).

We do obtain another subtree matching algorithm out of this. Let X have all its leaves in levels $s_1,\ldots,s_V$ (where $s_V = d$).

Theorem 3.5. One may perform subtree matching in only $O(n.V.\log \frac{2d}{V})$ steps.

Proof

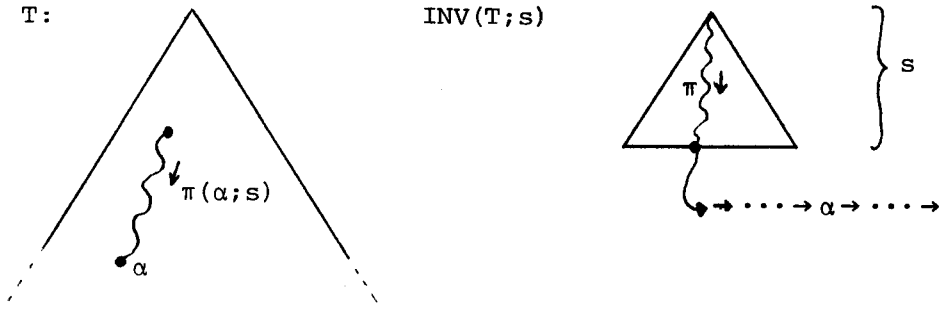
It takes only O(m) steps to determine s_1 to s_V in order (!) by means of breadth-first search. Determine the $s_1,\ldots,s_V$ equivalence classes, working as if T and X were joined into one tree. Note that we can keep track of the height s_i fathers of all nodes at the same time. Now resort to the accounting scheme introduced in 3.2. For i from 1 to V, add 1 to the counter of the height s_i father of each node in T which belongs to an s_i-class containing a leaf of X. The nodes in T whose counter finishes at value q identify all X-copies T contains, as before. The algorithm takes $n.V.\log \frac{2d}{V}$ for building the necessary equivalence classes, another n.V to do the accounting. It all adds up to the bound stated.

■

One should consider 3.5 as an improvement over 3.2 which still lies within the philosophy of the Karp-Miller-Rosenberg approach.

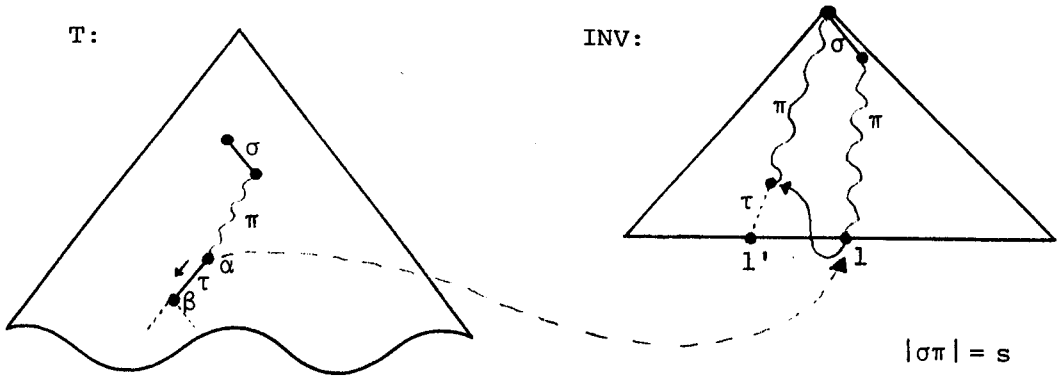
There is yet another approach in this respect, which can offer some advantages. Note that the present representation of classes does not keep track of the actual path $\pi(*;s)$ that goes with each class. But suppose we do. We saw for strings that it enabled us to respond faster when "more" patterns of the same length s had to be searched for later. We will see that for trees a similar speed-up can be attained.

Let INV(T;s) be the lexicographic tree obtained by entering all distinct paths $\pi(*;s)$ encountered in T. It means that each "downward" path π of length



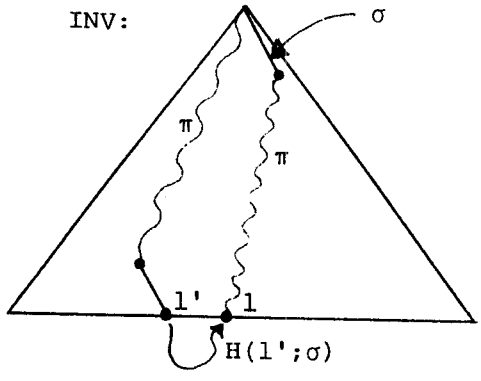
s embedded in T, is recorded in like order in INV. Each leaf of INV(T;s) shall contain a pointer to a list of all nodes α for which $\pi(\alpha;s)$ is just equal to the path from the root to this leaf. Clearly, leaves stand for s-equivalence classes this way.

For constructional purposes nodes of INV will carry a number of additional pointer-fields. To explain which, we shall anticipate the procedure for building INV from T. Suppose we reached α in T and just recorded it in the right "class" at leaf l of INV.



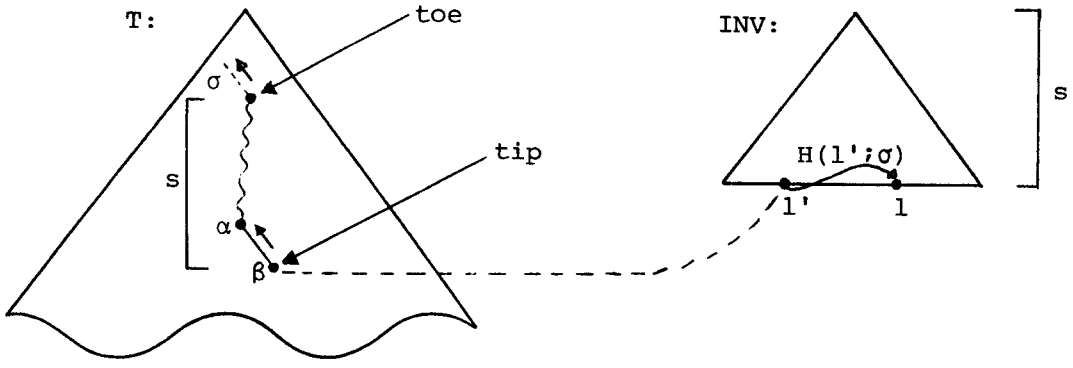
When we move on to β (e.g. following a pre-order traversal scheme) and wish to record $\pi\tau$ in INV, then it is very handy to have the pointers h(*) again. Eventually h(l) will get defined and we can "move" to l' in constant time. After working through the subtree below β, the procedure gets back to β in T and l' in INV and want us to return to α, in order to choose the "next" branch out of α or to back up even further. It is easy enough to go from β back to α, but ... how do we find l? The easiest solution would be to record a pointer to l in node α at the first visit. This does nót require an auxiliary field with nodes in T, but merely a stack that records the leaves of INV "entered into" along the life-line of the node currently being visited!

If T is large and the demand for stack-storage space prohibitive, then there is yet another solution. With each leaf l' of INV and for each symbol σ, record a pointer H(l';σ) to the leaf l at the end of $\sigma\pi$ (if it exists ...), where π is the path leading to the father of l'. It looks like each leaf of



INV will need to have up to k extra pointer fields, but that is not true. We can store the H-values in the empty son-fields of a leaf (assuming that all nodes have a uniform record-structure) at no extra cost. All we need is óne bit somewhere in a field to mark that the node actually is a leaf, so its son-fields will not be mistaken for son-pointers. Note that H(l';σ) keeps track of the "way back" when we move from l to l' during a "forward" step of the construction procedure.

Thus, if H(l';σ) hasn't been set earlier, it will be when we move from α to β during the pre-order traversal of T. It guarantees that whenever we need to back-up from β to α, the necessary H-value at the leaf corresponding to β will be available! To know which H-value (i.e. which σ) to select though, we need our pointer "toe" again (which moves up and down at a fixed distance of s behind "tip" along the current life-line). At all times the proper σ to use is just the label of the edge leading into the node pointed at by toe.



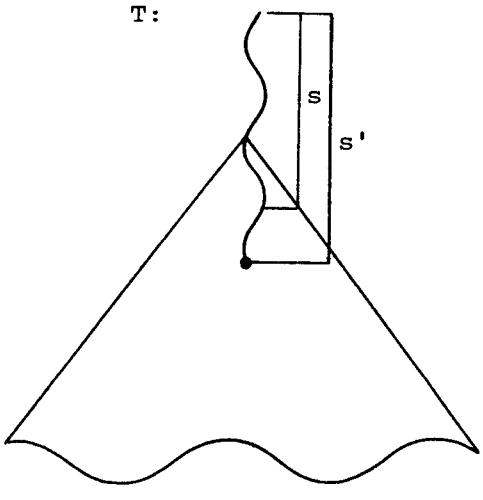
Theorem 3.6. INV(T;s) can be built in time proportional to $n + \mu(INV)$.

Proof

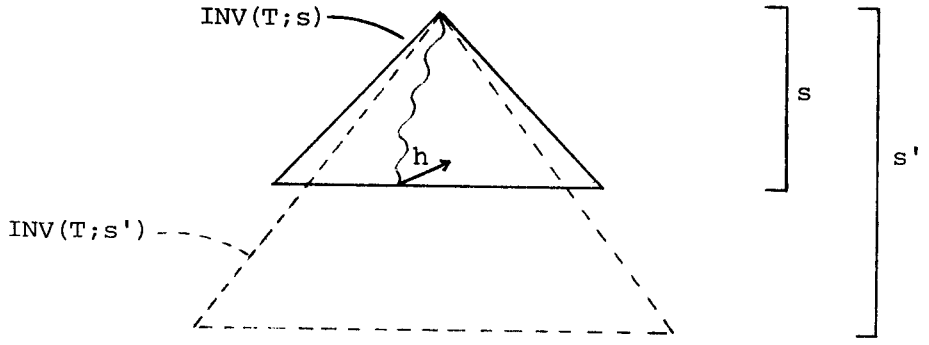
The construction goes exactly like that of 2.5, except that the length s substrings to be entered are now generated by a pre-order tree-walk. Whenever we need to back-up in T, the pointers H(*;*) enable us to reset our position in INV in only O(1) time. Hence the extra steps needed can all be charged to the nodes of T and no node accumulates more than O(k), i.e. O(1). The given time-estimate follows.

■

For the same reasons as given for 2.5, the result of 3.6 must be essentially optimal. Once INV(T;s) has been built and a new INV(T;s') is needed for some s', s' > s, we don't have to start from scratch. All paths of length s' to be entered into INV(T;s') (even those padded with #'s up front) have an initial portion of length s that must have been included in the "length s inventory" earlier. In particular, the length s prefix has already been recorded in



INV(T;s). Rather than recording the entire length s' path all over, it would be wiser (and more economical in storage) to extend the length s prefix in INV(T;s). Thus "overlaying" INV(T;s') and INV(T;s), we obtain a tree INV(T;s,s'). It will be obvious how to extend this to more inventories.

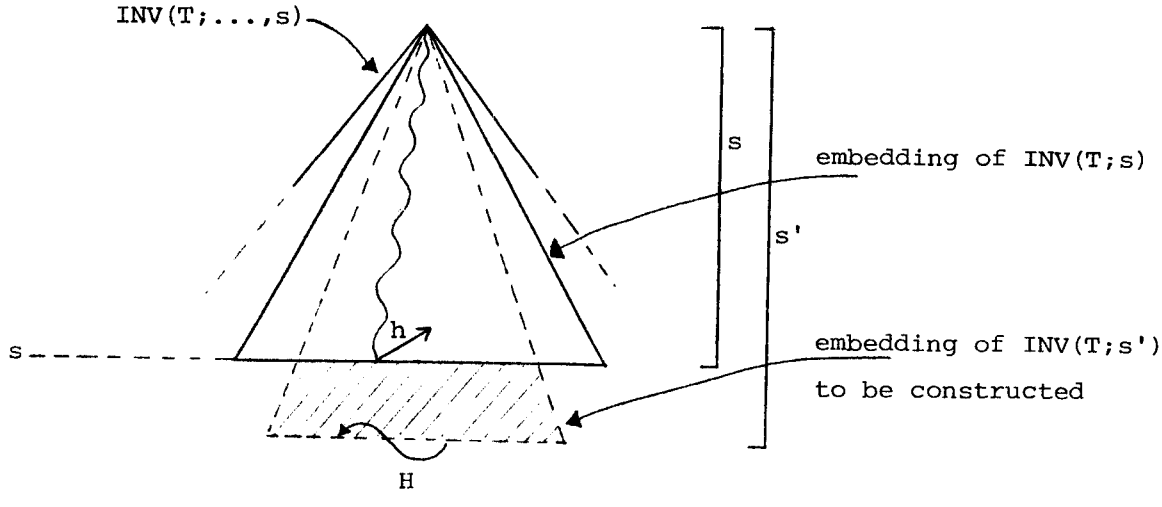


Theorem 3.7. For $s_1 < \ldots < s_v$ one can build $INV = INV(T;s_1,\ldots,s_v)$ in time proportional to $n.V + \mu(INV)$.

Proof

We shall argue that for $s' > s$ the construction of INV(T;...,s,s') from INV(T;...,s) in the overlayed structure takes time only proportional to $n + \mu(INV(T;...,s,s')) - \mu(INV(T;...,s))$.

The idea is that we can make use of the internal "shifting" information



previously calculated, viz. that we do not need to recompute the h-values within the "INV(T;s) portion" of INV(T;s'). Remember that the h-values were instrumental in making the algorithm of 3.6 (see also 2.5) efficient.

The construction of INV(T;s') proceeds along the lines of 3.5 without change, while performing a completely new pre-order traversal of T. After the left "ridge" of INV(T;s') has been set (it consists of #'s only and takes s'-s) we keep entering length s' paths as usual, except that it can be noticed that we will never run into a node at level s whose h-value was not defined before, nor that we ever need to climb beyond level s to determine the proper starting point from where a new path must be extended. Thus, the cost for recording another path of length s' either takes a constant (which can be charged to the current traversal step) or can be charged fully to nodes in the shaded region whose h-value switches from "undefined" to "defined". The number of nodes in the latter category is precisely $\mu(INV(T;...,s,s')) - \mu(INV(T;...,s))$. Finally, the cost for maintaining the proper H-pointers along the frontier of INV(T;s') and for "backing up" is bounded by O(n) as before.

To build $INV(T;s_1,...,s_v)$, construct $INV(T;s_1)$, $INV(T;s_1,s_2)$, ... in this order by means of the given algorithm. Putting $\mu(INV(T;)) = 0$, the total cost is proportional to

$$\sum_{1}^{v}\{n + \mu(INV(T;...,s_i)) - \mu(INV(T;...,s_{i-1}))\} = nV + \mu(INV(T;s_1,...,s_v))$$

■

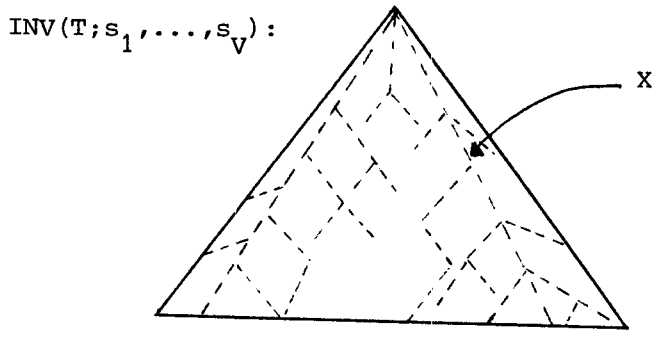
The way the traversal algorithm for T in 3.6 and 3.7 is set up (see the discussion preceding 3.6) and a pointer toe is maintained, should remind the reader of 3.3. It means that when we build the s_i-equivalence classes at some stage, then with each node of T classified we can record its height s_i father at no extra cost. We'll assume this to be included in the monstruous inventory structure $INV(T;s_1,...,s_v)$.

We shall now explore the relevance of this approach to subtree matching. Let X be an arbitrary pattern tree. Assume that its leaves occur only at levels $s_1,...,s_v$. We can find their numeric value in sorted order by breadth-first search in only O(m) steps. Let p be the largest number of different suffixes π any single leaf-path of X can have such that π is a leaf-path of X also.

Theorem 3.8. Assuming that $INV(T;s_1,...,s_v)$ has been built, one can do subtree matching in only O(n.p) steps.

Proof

If X occurs in T, then one must be able to locate each of its leaf-paths at least somewhere. It follows that X must be subtree of $INV(T;s_1,...,s_v)$, with



the roots coinciding. (Clearly the reverse is, in general, nót true.)

One can find all occurrences of leaf-paths of X and eventually determine that some connect up to an embedded copy of X in T in the following way. We resort to the counting method introduced in 3.2. Traverse X in pre-order, while simultaneously performing the exact same steps on INV. If we ever reach **a** point where we cannot make a same move, then halt and report that there can be no occurrence of X in T. Otherwise, whenever we reach a leaf l at depth s in X (where clearly $s \in \{s_1,\ldots,s_V\}$), then we know that the node α reached in INV contains a pointer to the list of all nodes β in T which are s-equivalent with l! (It must be, since we have overlayed the inventories of all s_i-equivalence relations in this structure.) When it happens, traverse the list and add 1 to the counter of the heigth s father of each node β encountered. (Remember that these fathers were listed with the nodes in each equivalence class.)

The algorithm certainly needs O(m) for the traversal procedure as such. Note that in the accounting of s-equivalences for some fixed $s \in \{s_1,\ldots,s_V\}$, no node β will come up more than once. When s varies, we can run into β no more often than the number of i's for which β occurs in an s_i-equivalence class whose $\pi(*;s_i)$ was matched to a leaf-path in X. Thinking of how the $\pi(\beta;s_i)$'s relate suffixwise in T, the corresponding leaf-paths in X must relate likewise and we see that this number is bounded by p. Charging the cost for list-traversal to the nodes β encountered, we conclude that no one can be charged more than p times. The total cost of the algorithm is bounded by $m + O(n.p)$.

■

The bound of 3.8 is obviously a rather pessimistic estimate.

Theorem 3.9. One can perform subtree matching in time proportional to $nV + \mu(INV)$, where INV is as before.

Proof

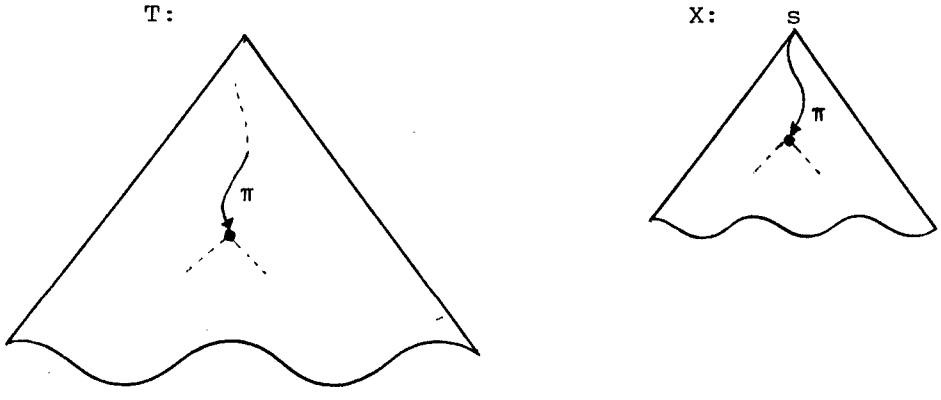
Observe that $p \leq V$ and add the time bounds of 3.7 and 3.8.

■

This result has been the last we could derive from the Karp-Miller-Rosenberg kind of approach. It shows that one can match in O(n.V) steps, which is better than 3.5, provided INV (hence X) is not too large (which it won't when e.g. $s_V < \log n/\log k$). If we need to match many patterns which have their leaves at levels $s_1,\ldots,s_V$, then it pays to precompute INV no matter what ... assuming there is sufficient auxiliary space available to store it when INV is big. Theorem 3.8 learns that under these conditions each single match takes only O(n.p) steps. In the next section we shall see algorithms that achieve the n.V or n.p bound without the need for bulky preconditioning.

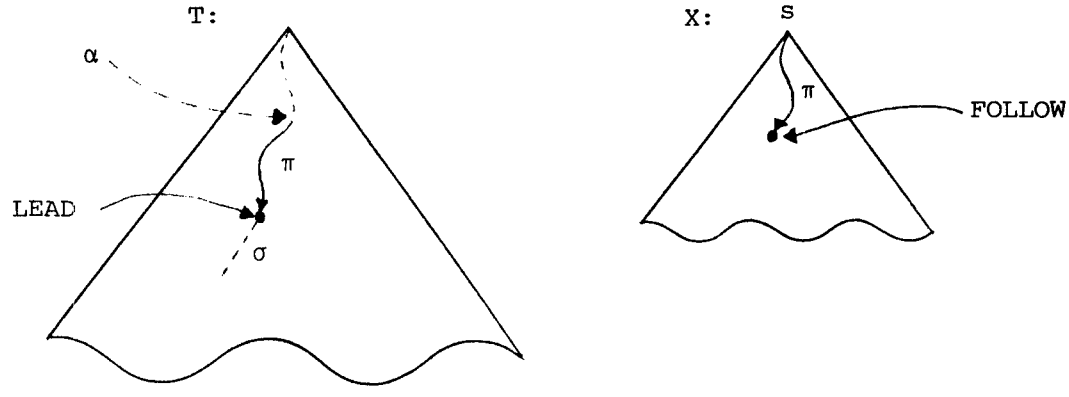
4. KMP-type subtree matching.

In this section a good number of ideas embedded in earlier algorithms will be combined into the most efficient subtree matching algorithm we have to offer. Recall that in some algorithms for subtree matching, we were really just



finding all occurrences of paths from the root down in X as strings in T and had the clever accounting method of 3.2 accumulate sufficient information to decide eventually where sufficiently many paths matched to form an embedded copy of X. We shall try to speed up the "path-matching" part along the lines of the KMP-algorithm. Since there will be some minor difficulties in knowing the location of the proper root-candidates by the time one needs to do some accounting, we'll ignore the details of their computation at first and pretend that they can be referenced free of charge.

The details of the algorithm are a bit tedious. The general idea is best developed from the fundamental invariant maintained by the algorithm. All action takes place during a pre-order traversal of T using pointer LEAD. The traversal procedure is crucial for everything that happens. A pointer FOLLOW on X identifies the end of a path π at the root s in X that can currently be read above LEAD. Since in general there is more than one location for FOLLOW on X to satisfy such a qualification, we shall resolve any danger of confusion as follows (it will become the "invariant" that makes everything go):



"FOLLOW (always) points to the deepest node of X such that the path s⇝FOLLOW in X occurs immediately above LEAD".

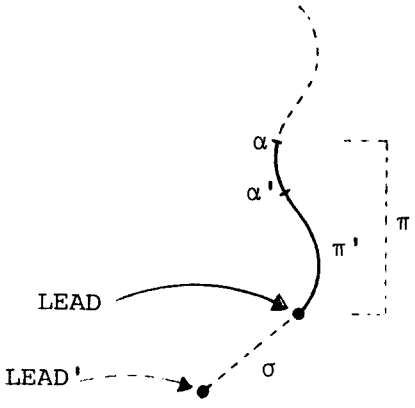
Another way of phrasing the invariant is to say that FOLLOW is located so that the node α above LEAD pertinent to the current position of FOLLOW is at maximum distance. We won't actually maintain a pointer to the "α" above LEAD at present, but merely note that we always know how far "up" it is when we let FOLLOW maintain the value of its depth in X.

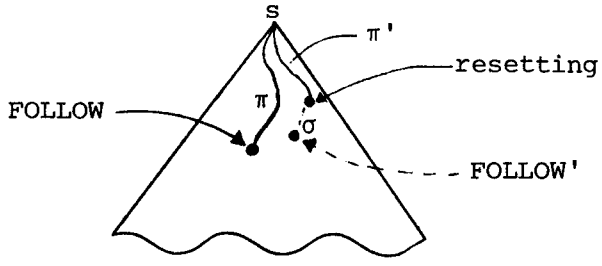
What happens when LEAD prepares to move down along the edge labeled σ, when the traversal algorithm says it should?

First of all, since we shall return to the position later for going down another edge in the same way, we must preserve the current value of FOLLOW at the node of departure. Put the address in an auxiliary field or, better yet, maintain it on a stack. When the traversal algorithm lets us return along edge σ, we reset FOLLOW to the value saved, restoring the proper invariant at no extra cost.

When there is an edge labeled σ out of FOLLOW in X, let both pointers move down along this edge. The invariant is trivially valid for the new configuration.

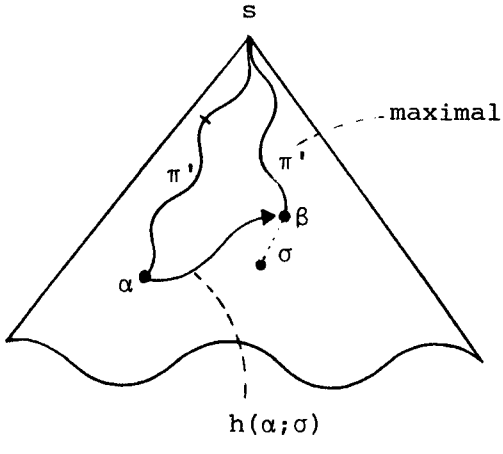
When there is no edge labeled σ out of FOLLOW in X, we must take special action not to let the invariant break down. Let π be the current "maximal" X-path above LEAD. Before we can move down, we must "somehow" reset FOLLOW





to another node in X from which a σ-branch does emanate. Clearly, it should be reset to the node at the end of the longest suffix π' of π for which this holds or to "⊥" when no such node exists. Here ⊥ essentially means "undefined", but it is better interpreted as a virtual node which has edges labeled with all symbols of Σ running into s. After resetting FOLLOW, we can let LEAD (as required) and FOLLOW move down the σ-branch we now know they both have (even when FOLLOW was set at ⊥ ...) and be sure that the invariant is preserved in the new configuration.

There is no need for the latter type of "move" to take more than constant time, because the resetting of FOLLOW desired depends only on X, and could have been precomputed and stored as a pointer in some auxiliary field of the node we visited. To this end, we define the following function h on X.



For nodes α and symbols $\sigma \in \Sigma$ we let

$$h(\alpha;\sigma) = \begin{cases} \text{the node } \beta, \text{ which has an outgoing } \sigma\text{-edge, and whose} \\ \text{"path" is the longest possible proper suffix of } \alpha\text{'s} \\ \text{path among all nodes with that property} \\ \\ \perp, \text{ if no such } \beta \text{ exists.} \end{cases}$$

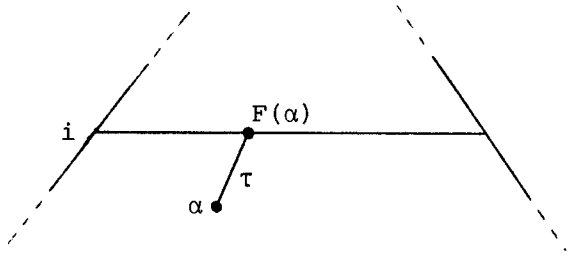
Recall that $k = \#\Sigma$ is assumed to be a fixed constant throughout. Note that h is pretty much like the revised recovery function for the KMP-algorithm, presented in section 2.

We have

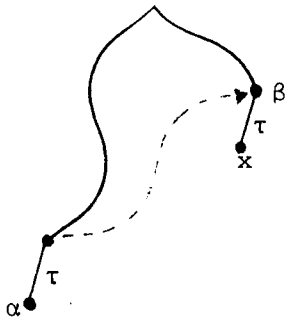
Lemma 4. The h-table can be computed in O(m) steps.

Proof

We construct the $h(\alpha;*)$ values "level after level" during a breadth-first traversal of X. Note that the $h(s;*)$ values, at the root, are all $\perp$. Suppose that the h-values have been computed up to some level i and the traversal algorithm leads us to a "next" node α in level i+1. Let the edge to its father be labeled τ.



How do we compute $h(\alpha;\sigma)$? First see what $h(F(\alpha);\tau)$ is. If it is $\perp$, then $h(\alpha;\sigma)$ is s (the root) if s has a σ-edge and $\perp$ otherwise. So let $h(F(\alpha);\tau)$ be defined and equal to some real node β. Its τ-son be x. Note that x is at some level $\leq$ i,



which means that its h-values are available. One easily verifies that

$$h(\alpha;\sigma) = \begin{cases} x, \text{ if } x \text{ has an outgoing } \sigma\text{-edge} \\ h(x;\sigma), \text{ if it has not.} \end{cases}$$

Computing the h-values thus required only O(k), i.e. O(1), per node. ■

Hence, if the node visited by LEAD has no σ-son, we need only reset FOLLOW to h(FOLLOW;σ) before we can move on.

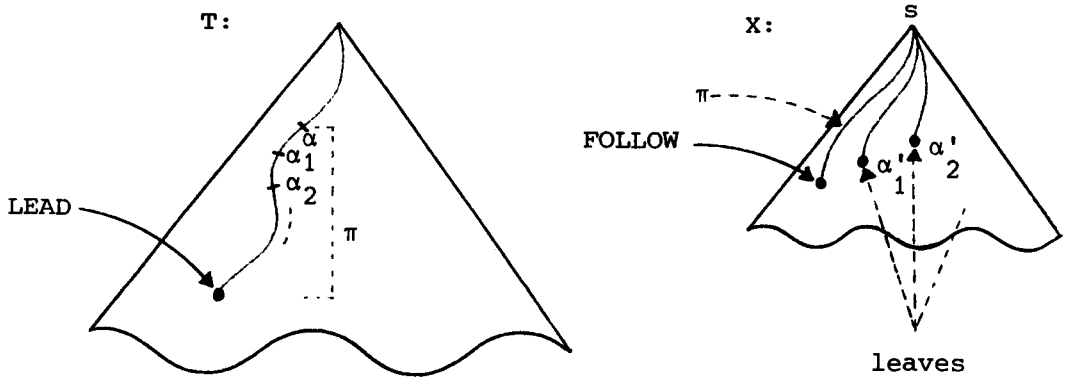
This covers all details of the traversal part. Note that we did provide for a means to re-instate FOLLOW to its proper value when we back-up. We conclude

Lemma 4.2. Pre-order traversal of T using LEAD, with FOLLOW jumping across X to maintain the invariant, requires only O(n) time.

Let DEP be a counter which keeps track of the depth of FOLLOW as it moves on X. DEP is easily maintained when FOLLOW moves up or down an edge by adding or subtracting 1, but it needs to be changed more drastically when FOLLOW is reset. Note that the proper decrement for DEP is just the difference in depth between the two nodes involved in the jump of FOLLOW. To let this go fast, we precompute the depth dep(α) of each node α in X. It is well-known that this requires no more than O(m) steps.

We shall now discuss how all information we need for subtree matching (along the lines of 3.2) can be accumulated as the traversal algorithm progresses. It will show the real intricacy of the traversal algorithm.

Let nodes of T have "counters" again, which are initially set at 0. Let LEAD begin its pre-order traversal of T. Each time some node gets visited for the first time, we should like to know the exact location of the root-candidates $\alpha_1, \ldots$ (if any) above it which make the LEAD-node look like a leaf



of X, and add 1 to their counters. It means that we want all nodes α_i for which the path $\pi_i : \alpha_i \rightsquigarrow$ LEAD in T is a leaf-path in X. If we just consider the paths as such, it is obvious that they must all be suffixes of the lóngest path π "ending" at LEAD that can be traced in X from the root s down. It is exactly the path from s to FOLLOW (of length DEP)!

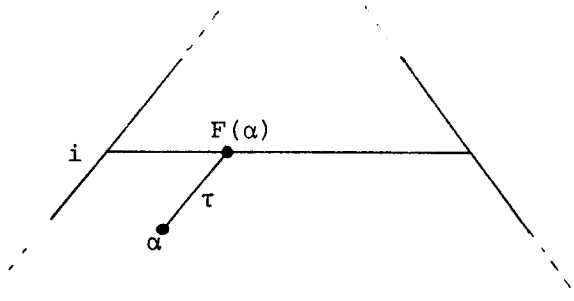
The question what segments leading into LEAD are leaf-paths of X can thus be answered completely by looking at the path s$\rightsquigarrow$FOLLOW in X. The answer could be precomputed with each location that FOLLOW visits! To this end we define a function L on nodes of X as follows

$$L(\alpha) = \begin{cases} \text{the lowest leaf } \beta \text{ of X for which } s \rightsquigarrow \beta \\ \text{is a proper suffix of } s \rightsquigarrow \alpha \\ \\ \perp, \text{ if no such leaf exists.} \end{cases}$$

Lemma 4.3. The L-function can be computed in only O(m) steps.

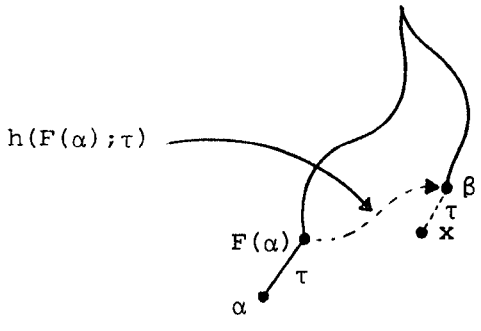
Proof

We shall determine the L(*)-values "level after level" during a breadth-first traversal of X, in very much the same way as the h-values in 4.1. Clearly L(s), at the root, is $\perp$. Assume that we have computed the L values up to some level i and now visit a "next" node α in level i+1. Let the edge to its father have label τ



If $h(F(\alpha);\tau)$ is $\perp$, then $L(\alpha)$ is sure to be $\perp$ also. So let $h(F(\alpha);\tau)$ be defined and equal to some real node β. Its son along the τ-edge be x.

Note that x is at some level $\leq$ i, which means that its L-value is available. One easily verifies

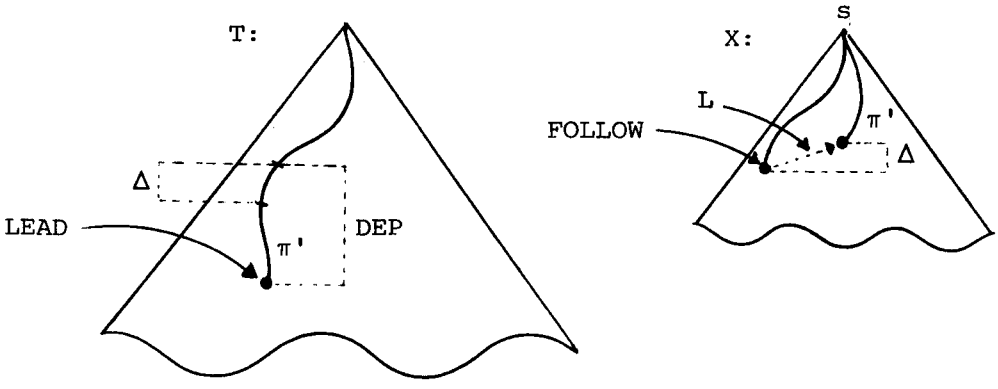


$$L(\alpha) = \begin{cases} x, \text{ if } x \text{ is a leaf,} \\ L(x), \text{ if it is not.} \end{cases}$$

Computing the L-values thus costs no more than O(1) per node.

■

More important than $L(\alpha)$ itself is the decrement in depth (counting from a current FOLLOW position) that goes along with it. If it is Δ, then we know that the first root-candidate that makes LEAD look like an X-leaf is at height DEP-Δ above it! (When FOLLOW happens to be a leaf itself, the first root-candidate is "α" at height DEP away and the node at height DEP-Δ is second.)



Let $del(\alpha)$ be equal to the (positively measured) difference in depth between α and $L(\alpha)$. The development can now be summarized in the following, fundamental result.

Theorem 4.4. Whenever LEAD visits a node of T, the root-candidates that make it look like an X-leaf are at heights DEP (only when FOLLOW happens to be a leaf), DEP - del(FOLLOW), DEP - del(FOLLOW) - del(L(FOLLOW)), ... above it on its life-line.

In the statement, L is to be iterated until one ends at $\perp$. We note that this must happen within p times.

Theorem 4.4 shows that, with all the precomputed information at each node (all together no more than O(k) per node), there is a way to "identify" the root-candidates with each LEAD-position ... due to the twists of the traversal algorithm. To do the necessary accounting for subtree matching, we should augment the counter of each root-candidate of a node by 1 when LEAD visits the node for the first time. Since there can be no more than p root-candidates for

a node, the algorithm will essentially make at most p steps per node and runs in O(n.p) total ... assuming that the actual referencing of the necessary height s_i fathers does not take more time. Thus, the ultimate complexity of the subtree matching algorithm seems to depend more on ancestor calculation than on anything else!

There are several ways to compute the required fathers, but which one to use will depend on the model of computation adopted. Let us first consider an implementation of the subtree matching algorithm on a reference machine. Briefly, a reference machine makes a strict separation between data (numbers) and pointers (references or addresses). No arithmetic operations are allowed on pointers, in fact it is best not to associate any numeric value to pointers at all. Memory locations can be accessed only when you have an explicit pointer to it. The lack of any facility for address-calculation makes reference machines a perfect vehicle to study how powerful pure list-processing in itself can be. For more details about reference machines, see Tarjan [12].

Theorem 4.5. One can perform subtree matching in O(n.V) steps on a reference machine.

Proof

Let T and X be represented as linked structures in memory. We keep pointers to their root at known locations, so we can always enter them (... at the root only, but that will be sufficient). Traverse X in breadth-first order to compute with each node, in the way explained, the record of h, dep and L values for that node. The constructions given work on a reference machine without loss of time.

Let LEAD begin its traversal of T, with FOLLOW hopping along on X. DEP is maintained as a data-value. Each time a node is visited for the first time, we create a linked list of auxiliary records in which we enter the distances to its root-candidates one by one, as computed along the lines of 4.4. It needs to cost no more than O(1) for each additional value recorded. The node visited will be given a pointer to the beginning of the list so created.

Observe that the auxiliary lists attached to the nodes of T this way are all sorted, by virtue of the way 4.4 picks the suffixes. No list is more than p long, but the actual values range from s_1 to s_V. There seems to be no other way but to resort to the algorithm of 3.3 (which runs perfectly on a reference machine) to compute the fathers at these distance.

So, we first determine s_1 to s_V in sorted order (by means of a breadth-first traversal of X) and proceed as follows. For each i from 1 to V we calculate

the heigth s_i fathers (according to 3.3) and, as we go along, we update the counter of the father found whenever the auxiliary list of the node visited contains s_i. It takes no search to determine this, because we can maintain an entry point on each list which is positioned immediately áfter the last record picked off (this works because the lists are sorted and values are picked in order).

The total time spend is $O(m) + O(n.p)$, plus $O(nV)$ for the final accounting phase. It only takes another traversal of T to identify all locations where the counter reached q, to signal the embedded occurrences of X.

■

We shall see a slightly different method of father-calculation momentarily, which could occasionally be better than what we did. Note that the reference machine only needs to have the "+" in its repertoire of arithmetic operators.

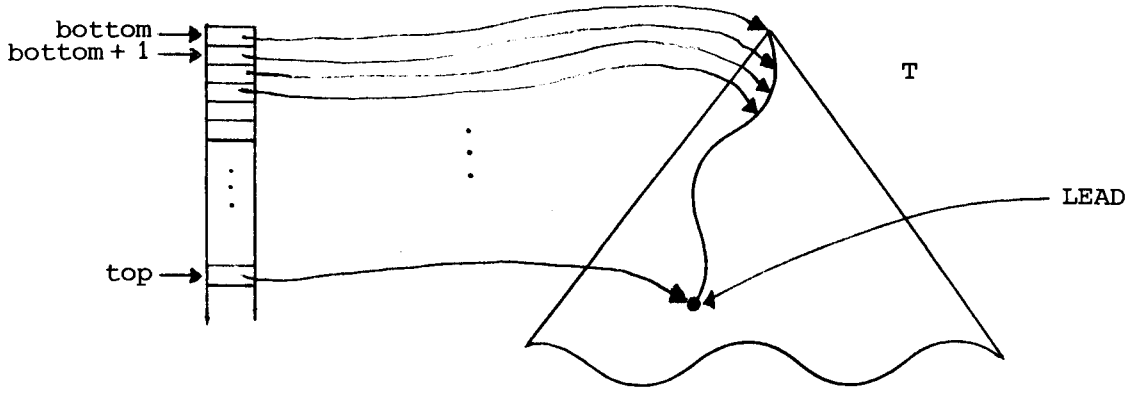
Next we shall implement the algorithm on a random access machine (RAM), which offers the succinct advantage of address-calculation. See Aho, Hopcroft and Ullman [2] for details about RAMs. We note that there can be a great difference in power between RAMs, depending on the arithmetic operators we allow in the instruction set. Hence we should be careful and state that we shall only have a need for "+". The resulting device is sometimes called a "plus-RAM".

Theorem 4.6. One can perform subtree matching in only $O(n.p)$ steps on a (plus) RAM.

Proof

The basic algorithm is exactly as before, and we shall only consider how the selected root-candidates can now be accessed at "no" extra cost.

As LEAD traverses T, maintain a stack of contiguous locations in which you keep the addresses of (i.e. pointers to) the successive nodes on the path from the root to the current position of LEAD.



Whenever we need to access the node at some distance Δ above LEAD, we merely have to pick up the contents of address top - Δ (which one can access "immediately", on a RAM).

Hence, when LEAD traverses T and visits nodes, then, each time it gets to a new node for the first time, we can immediately access the root-candidates (and add 1 to their counter) as their distances above LEAD appear from the calculations in 4.4.

It shows that we stay within the O(n.p) run-time of the basic algorithm.

■

The bound of 4.6 is actually a poor substitute for the following, more precise expression. For nodes α in T, let match(α) be the number of leaf-paths of X that occur "at" α. (Thus, for instance, $0 \leq \text{match}(\alpha) \leq p$ for all α.) Careful counting in the algorithm shows

Corollary 4.7. One can perform subtree matching on a (plus) RAM in time proportional to $n + \sum_{\alpha \in T} \text{match}(\alpha)$.

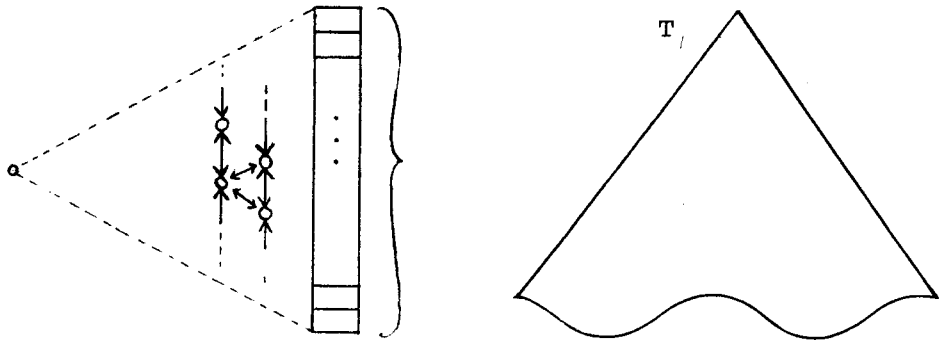
Since each "match" must be accounted for in our algorithm, it may well be optimal as it stands. The addressing technique of 4.6 could be simulated on a reference machine and come out better than 4.5 under certain conditions.

Corollary 4.8. One can perform subtree matching on a reference machine in only $O(n.p.\log \frac{2d}{p})$ steps.

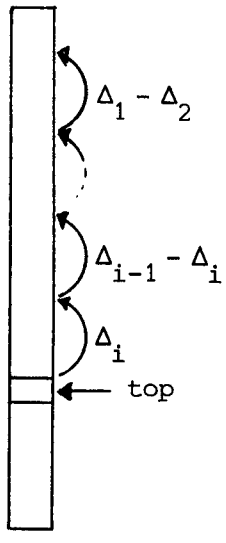
Proof

It will be sufficient to show how one can efficiently simulate "random access" in the stack. Fortunately the "values" recorded in the stack were pointers, so we don't have to go further than that.

Create a list of records (of length $\leq n$) sufficient to hold the stack as it grows and shrinks. Number records consecutively in linked order, to simulate stack-addresses. Build a chained, balanced tree on top of it (e.g. a chained 2-3 tree as in [3]) and record with each node the value of the "smallest index" in the subtree below it.



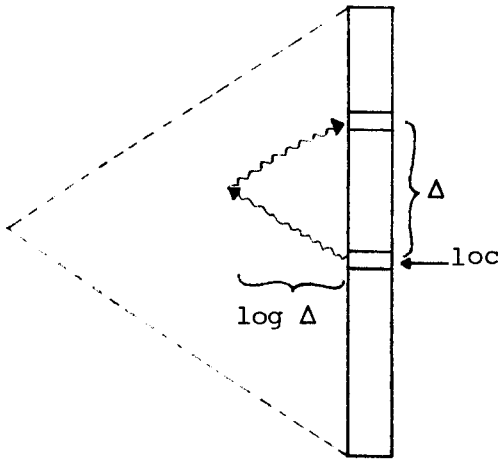
Let the subtree matching algorithm run, and suppose that at some point we need to access the root-candidates (ancestors) at distances $\Delta_1,\ldots,\Delta_i$. We assume that $i \geq 1$, since there would be nothing to prove otherwise. Remember that 4.4 produces these values in the order $\Delta_1 > \ldots > \Delta_i$ (and note that necessarily $\Delta_1 \leq d$). From the currently visited location top in the stack, we therefore need to descend to i different spots as follows



When you are at some location loc in the stack and want to move down by Δ, it takes only $O(\log 2\Delta)$ to do so using the auxiliary tree. See Brown and Tarjan [3] for the details. Hence, the complete addressing sequence can be traversed in only

$$\log 2\Delta_i + \log 2(\Delta_{i-1} - \Delta_i) + \ldots + \log 2(\Delta_1 - \Delta_2) \leq$$
$$\leq i \log \frac{2\Delta_1}{i} = O(p \log \frac{2d}{p})$$

steps on a reference machine.



Adding O(n) for building the balanced tree still keeps the total time spent within $O(n.p.\log \frac{2d}{p})$.

■

Note that the algorithm of 4.6 (4.7) essentially reduces to the KMP-variant presented in 2.1, if we let T and X be the "linear" trees corresponding to the lexicographic representation of a single text t and a single pattern x. The full power of 4.6 (4.7) becomes apparent when we consider the problem of matching many patterns $x_1,\ldots,x_1$ in a text t. Enter x_1 to x_1 in a lexicographic tree. If they overlap prefix-wise, some internal nodes will get marked as "pseudo-leaves". Modify the L-function to also take these nodes as "leaves" into account. View t as a (degenerate) lexicographic tree and let the "subtree matching" algorithm run. If we disregard the accounting (which has no purpose now) and merely note the values that come out of 4.4 (together with an identifier for each string-pattern found as segment at the present LEAD location), then we see that we need to spend only O(n) time plus a constant for each match found. This is the nature of a result of Aho and Corasick [1], in more realistic terms. It is optional whether one likes to include the $O(|x_1| + \ldots + |x_1|)$ bound for building the "tree" of patterns or not.

5. References (reference [4] not cited in the text)

[1] Aho, A.V. and M.J. Corasick, Efficient string matching: an aid to bibliographic search, Comm. ACM 18 (1975) 333-340.

[2] Aho, A.V., J. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass. (1974).

[3] Brown, M.R. and R.E. Tarjan, A representation for linear lists with movable fingers, Proc. 10th Annual ACM Symp. on Theory of Computing, San Diego, 1978, pp. 19-29.

[4] Galil, Z. and J. Seiferas, Saving space in fast string matching, Techn. Rep. 223, Dept. of Computer Science, the Pennsylvania State Univ. (1977).

[5] Karp, R.M., R.E. Miller and A.L. Rosenberg, Rapid identification of repeated patterns in strings, trees and arrays, Proc. 4th Annual ACM Symp. on Theory of Computing, Denver, 1972, pp. 125-136.

[6] Knuth, D.E., The art of computer programming, vol. 3: sorting and searching, Addison-Wesley, Reading, Mass. (1973).

[7] Knuth, D.E., J.H. Morris and V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323-350.

[8] Knuth, D.E. and V.R. Pratt, Automata theory can be useful, STAN-CS-71- , Computer Science Dept., Stanford Univ. (1971).

[9] Morris, J.H. and V.R. Pratt, A linear pattern matching algorithm, Rep. #40, Computing Centre, Univ. of California, Berkeley (1970).

[10] Pippenger, N.J., On the evaluation of powers and related problems, Conf. Rec. 17th Annual IEEE Symp. on Found. of Computer Science, Houston, 1976, pp. 258-263.

[11] Rivest, R.L., On the worst-case behavior of string searching algorithms, SIAM J. Comput. 6 (1977) 669-674.

[12] Tarjan, R.E., Reference machines require non-linear time to maintain disjoint sets, STAN-CS-77-603, Computer Science Dept., Stanford Univ. (1977).

[13] van der Steen, G., private comm., Amsterdam, 1979.

[14] van Leeuwen, J., The complexity of data organisation, in: K.R. Apt and J.W. de Bakker (eds.) Foundations of Computer Science, vol. I, Math. Centre Tract 81 (1976) 37-147.

[15] Yao, A.C., On the evaluation of powers, SIAM J. Comput. 5 (1976) 100-103.