PATTERN MATCHING FOR PERMUTATIONS*

Prosenjit Bose[1] **Jonathan F. Buss**[2] **Anna Lubiw**[3]

Abstract

Given a permutation T of 1 to n, and a permutation P of 1 to k, for $k \leq n$, we wish to find a k-element subsequence of T whose elements are ordered according to the permutation P. For example, if P is $(1, 2, \ldots, k)$, then we wish to find an increasing subsequence of length k in T; this special case can be done in time $O(n \log \log n)$ [CW]. We prove that the general problem is NP-complete. We give a polynomial time algorithm for the decision problem, and the corresponding counting problem, in the case that P is *separable*—i.e. contains neither the subpattern $(3, 1, 4, 2)$ nor its reverse, the subpattern $(2, 4, 1, 3)$.

1. Introduction

The *pattern matching problem for permutations* is the following: We are given a permutation $T = (t_1, t_2, \ldots, t_n)$ of 1 to n, which we call the *text*, and a permutation $P = (p_1, p_2, \ldots, p_k)$ of 1 to k, $k \leq n$, which we call the *pattern*. We wish to know whether there is a length k subsequence of T, say $T' = (t_{i_1}, t_{i_2}, \ldots, t_{i_k})$, with $i_1 < i_2 < \ldots < i_k$, such that the elements of T' are ordered according to the permutation P—i.e. $t_{i_r} < t_{i_s}$ iff $p_r < p_s$. If T does contain such a subsequence, we will say that T *contains* P, or that P *matches into* T.

Herb Wilf spoke at the 1992 SIAM Discrete Math meeting about counting pattern matchings of permutations. He asked whether there is an algorithm to decide the pattern matching problem for permutations that runs faster than exponential time.

This note contains one negative and one positive result. We show that the general decision problem is NP-complete, and the counting problem is #P-complete. (See [GJ, Section 7.3] for the notion of #P-completeness.) We give a polynomial time algorithm for the decision and counting problems in case the pattern P contains neither the pattern $(3, 1, 4, 2)$ nor the pattern $(2, 4, 1, 3)$. Such a pattern P is called *separable*, for reasons given in Section 3.

There has been considerable interest in counting the number of permutations T that do *not* contain a smaller permutation P. Knuth [K, Section 2.2.1] showed that the number of permutations of $1, \ldots, n$ without the pattern $(3, 1, 2)$ is equal to the n^{th} Catalan number. Lovász [L, p. 27] considered permutations without the pattern $(2, 1, 3)$. Rotem [R] considered permutations without

* Work supported in part by NSERC. A preliminary version appeared in Algorithms and Data Structures, eds. F. Dehne, J.-R. Sack, N. Santoro, S. Whitesides, Lecture Notes in Computer Science 709, 1993, 200–209.

1 Dept. Computer Science, McGill University, Montreal, Canada, H3A 2A7. jit@muff.cs.mcgill.ca

2 Dept. Computer Science, University of Waterloo, Waterloo, Canada, N2L 3G1. jfbuss@uwaterloo.ca

3 Dept. Computer Science, University of Waterloo, Waterloo, Canada, N2L 3G1. alubiw@uwaterloo.ca

either of the patterns $(2,3,1)$ or $(3,1,2)$. Simion and Schmidt [SS2] go further, counting the number of permutations avoiding all the patterns in any subset of the permutations on $1,2,3$.

More recently, there have been results on 4 element patterns: West [W] showed that the number of separable n-permutations is the $(n-1)^{\text{st}}$ Schröder number. He used a general technique of generating trees, which also allowed him to count the number of permutations that do not contain various pairs of 3- and/or 4-permutations [W2]. Further work has been done by Stankova [S].

There has also been interest in characterizing the permutations T that do not contain a smaller permutation P in terms of how T can be sorted. Knuth [K, Section 2.2.1] showed that permutations without the pattern $(2,3,1)$ are precisely the permutations sortable using one stack. This idea of characterizing permutations sortable via some network of queues and stacks was pursued further by Tarjan [T], Even and Itai [EI], and Pratt [P]. An analogous characterization of separable permutations will be given.

2. The NP-Completeness Result

Theorem 2.1. The pattern matching problem for permutations is NP-complete.

Proof. We will reduce 3-SATISFIABILITY to this problem. Suppose we have an instance of 3-SATISFIABILITY with variables $v_1,\ldots,v_l$ and clauses $C_1,\ldots,C_m$, where each clause C_i is the disjunction of three literals, and each literal is a variable or the negation of a variable.

We will first construct a pattern and a text with repeated elements, and later show how to eliminate the need for the repetitions. When the pattern has some element repeated, then copies of the repeated element must match to the copies of some repeated element in the text. In other words, using the notation of the introduction, we add the condition that $t_{i_r}=t_{i_s}$ iff $p_r=p_s$. For example, the pattern $(1,3,1,2)$ matches into the text $(1,3,2,6,3,4,5)$ (in particular, it matches to $(3,6,3,4)$) but does not match into the text $(1,4,2,3)$.

The pattern P will have a substring (i.e., contiguous subsequence) P_V for the variables, and a substring P_{C_i} for each of the clauses. Similarly, the text T will have a substring T_V for the variables, and a substring T_{C_i} for each clause. We will join these substrings of P and of T one after another, with "guards" in-between to ensure that elements of one of the substrings of the pattern can only match to elements of the corresponding substring of the text. The construction of these guards will be described later.

Construct $P_V=(1,2,\ldots,l+1)$, and $T_V=(2,1,4,3,6,5,\ldots,2l,2l-1,2l+1)$. Note that a match of P_V into T_V will force i in P_V to match with either $2i$ or $2i-1$ in T_V, for $1\le i\le l$. The former will correspond with setting the variable v_i TRUE, and the latter with setting it FALSE. Also, $l+1$ in P_V must match with $2l+1$ in T_V; these values will separate the variables from other values in the other substrings.

Consider the clause C_i containing variables v_{i_1},v_{i_2},v_{i_3} in either positive or negated form, with $i_1<i_2<i_3$. Construct $P_{C_i}=(l+i+1,i_1,i_2,i_3,l+i+1)$. For $j=1,2,3$, let t_j be $2i_j$ if variable v_{i_j}

appears in C_i positively, and $2i_j - 1$ otherwise; and let f_j be the other of the two values (i.e. f_j is $2i_j - 1$ if variable v_{i_j} appears in C_i positively, and $2i_j$ otherwise). Let $b_i = 2l + 7(i-1) + 1$. There are seven truth assignments to the variables that make the clause true; these assignments correspond to the seven sequences $s_{i,1} = (t_1, t_2, t_3)$, $s_{i,2} = (t_1, t_2, f_3)$, $s_{i,3} = (t_1, f_2, t_3)$, $s_{i,4} = (t_1, f_2, f_3)$, $s_{i,5} = (f_1, t_2, t_3)$, $s_{i,6} = (f_1, t_2, f_3)$, $s_{i,7} = (f_1, f_2, t_3)$. The text string T_{C_i} comprises these seven sequences, each bracketed by a pair of equal values; i.e., T_{C_i} is the sequence

$$(b_i + 1, s_{i,1}, b_i + 1, b_i + 2, s_{i,2}, b_i + 2, b_i + 3, s_{i,3}, b_i + 3, \ldots, b_i + 7, s_{i,7}, b_i + 7).$$

For example, consider the clauses $C_1 = (v_1 \vee v_2 \vee \overline{v_4})$ and $C_2 = (\overline{v_2} \vee v_3 \vee \overline{v_5})$ over variables $\{v_1 \ldots, v_5\}$. The corresponding patterns and text strings are

$$\begin{aligned}
P_V &= (1, 2, 3, 4, 5, 6),\\
T_V &= (2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 11),\\
P_{C_1} &= (7, 1, 2, 4, 7),\\
T_{C_1} &= (12, 2, 4, 8, 12, 13, 2, 4, 7, 13, 14, 2, 3, 8, 14, 15, 2, 3, 7, 15,\\
&\qquad 16, 1, 4, 8, 16, 17, 1, 4, 7, 17, 18, 1, 3, 8, 18),\\
P_{C_2} &= (8, 2, 3, 5, 8),\\
T_{C_2} &= (19, 4, 6, 10, 19, 20, 4, 6, 9, 20, 21, 4, 5, 10, 21, 22, 4, 5, 9, 22,\\
&\qquad 23, 3, 6, 10, 23, 24, 3, 6, 9, 24, 25, 3, 5, 10, 25).
\end{aligned}$$

In any matching of P_{C_i} into T_{C_i} that respects a matching of P_V into T_V, the occurrences of $l + i + 1$ must match with one of the bracket pairs, and thus (i_1, i_2, i_3) must be one of the sequences $s_{i,k}$.

Claim 2.2. The set of clauses $\{C_i\}$ is satisfiable if and only if there is a simultaneous matching of P_V into T_V and each P_{C_i} into T_{C_i}.

Proof. A matching of P_V into T_V corresponds to a truth-value assignment to the variables. By the above construction, this matching can be extended to a matching of P_{C_i} into T_{C_i} if and only if the corresponding assignment satisfies C_i. ∎

Two issues remain: how to put guards between the segments of P and of T to ensure that they remain separate; and how to eliminate the need for repeated elements in P and T.

We consider the guards first. Recall that m is the number of clauses and l the number of variables. Let P be the sequence $(l + m + 2, P_V, l + m + 2, P_{C_1}, l + m + 2, P_{C_2}, l + m + 2, \ldots, l + m + 2, P_{C_m})$, followed by $3m + 1$ copies of $l + m + 2$. Let T be the sequence $(2l + 7m + 2, T_V, 2l + 7m + 2, T_{C_1}, 2l + 7m + 2, T_{C_2}, 2l + 7m + 2, \ldots, T_{C_m})$, followed by $3m + 1$ copies of $2l + 7m + 2$. Observe that any matching of P into T must match the element $l + m + 2$ of P to the element $2l + 7m + 2$

of T, because there are $4m+2$ copies of $l+m+2$ in P, and no value except $2l+7m+2$ appears more than $4m+1$ times in T. Since there are the same number of occurrences of $l+m+2$ in P as there are of $2l+7m+2$ in T, they do perform as guards, ensuring that P_V matches into T_V and each P_{C_i} matches into T_{C_i}.

Finally, let us turn to the question of eliminating repeated elements. It is easier to describe the construction if we allow the pattern and the text to each be a sequence of distinct numbers, not necessarily integers. Then to obtain a permutation, simply replace each element by its ordinal.

Instead of using a singleton in P_V for each variable, which must map to one of two possible values in T_V, we will use a pair in P_V which must map to one of two possible pairs in T_V. Occurrences of the variable in clauses will then be modelled as values intermediate to the corresponding pair, thus avoiding the need to duplicate values. Construct $P_V = (1, 2, \ldots, 2l-1, 2l, 2l+1)$, and $T_V = (3, 4, 1, 2, 7, 8, 5, 6, \ldots, 4l-1, 4l, 4l-3, 4l-2, 4l+1)$. Observe that if P_V matches into T_V, then the pair $2i-1, 2i$ must match with either $4i-1, 4i$ or $4i-3, 4i-2$, which we will interpret as setting variable v_i TRUE or FALSE respectively.

Consider a clause C_i containing variables $v_{i_1}, v_{i_2}, v_{i_3}$ in either positive or negated form, with $i_1 < i_2 < i_3$. Construct $P_{C_i} = (2l+2i+1, u_{i,1}, u_{i,2}, u_{i,3}, 2l+2i)$, where $u_{i,j}$ is a number strictly between $2i_j - 1$ and $2i_j$, larger than all values previously chosen in that interval. For $j = 1, 2, 3$, let $\hat{t}_j$ be the open interval $(4i_j - 1, 4i_j)$ if variable v_{i_j} appears in C_i positively, and the interval $(4i_j - 3, 4i_j - 2)$ otherwise; and let $\hat{f}_j$ be the other of the two intervals. Let $t_{j,1}$, $t_{j,2}$, $t_{j,3}$ and $t_{j,4}$ be values in the interval $\hat{t}_j$, and let $f_{j,1}$, $f_{j,2}$ and $f_{j,3}$ be values in the interval $\hat{f}_j$; all these values are to be distinct from each other and larger than all values previously chosen in their respective intervals. Let $b_i = 4l + 14(i-1) + 1$.

For the seven satisfying assignments of C_i we choose the seven sequences $s_{i,1} = (t_{1,1}, t_{2,1}, t_{3,1})$, $s_{i,2} = (t_{1,2}, t_{2,2}, f_{3,1})$, $s_{i,3} = (t_{1,3}, f_{2,1}, t_{3,2})$, ..., $s_{i,7} = (f_{1,3}, f_{2,3}, t_{3,4})$. The text segment T_{C_i} is the sequence

$$(b_i+2, s_{i,1}, b_i+1, b_i+4, s_{i,2}, b_i+3, b_i+6, s_{i,3}, b_i+5, \ldots, b_i+14, s_{i,7}, b_i+13).$$

In any matching of P_{C_i} into T_{C_i} that respects a matching of P_V into T_V, the occurrences of $2l+2i+1$ and $2l+2i$ must match with one of the bracket pairs, and thus (i_1, i_2, i_3) must be one of the sequences $s_{i,k}$.

Claim 2.3. The set of clauses $\{C_i\}$ is satisfiable iff there is a matching of P_V into T_V and each P_{C_i} into T_{C_i}.

It remains to implement the guards without the use of duplicate elements. Note that the parts of P so far, $(P_V, P_{C_1}, \ldots, P_{C_m})$, form a permutation of $1, \ldots, 2l+5m+1$, and the corresponding parts of T form a permutation of $1, \ldots, 4l+35m+1$. Augment P to a permutation of $1, \ldots, 4l+36m+3$ by inserting the decreasing subsequence of $2l+31m+2$ values $4l+36m+3, \ldots, 2l+5m+2$, using the first

$m+1$ values as guards—the first before P_V, the second before P_{C_1}, etc.—and putting the remaining $2l+31m+2$ values at the end of P. Similarly, augment T to a permutation of $1, \ldots, 6l+66m+3$ by inserting the decreasing subsequence of $2l+31m+2$ values $6l+66m+3, \ldots, 4l+35m+2$, using the first $m+1$ values as guards, and putting the remaining $2l+31m+2$ values at the end of T.

If the last element of P maps to the last element of T, then, since there are an equal number of larger elements in P as in T, these large elements act as guards. If the last element of P maps to a different one of the large trailing elements of T, then there are insufficiently many larger elements in T to accommodate the larger elements of P. The final possibility is that the last element of P maps to one of the smaller $4l+35m+1$ values of T. But then the trailing $2l+31m+2$ values of T cannot be used in the match, leaving only $4l+36m+2$ values—not enough to match P, which has $4l+36m+3$ elements. ∎

Corollary 2.4. Counting the number of matchings of a pattern into a text is #P-complete.

Proof. In the reduction above, the matchings of P into T are in one-to-one correspondence with the satisfying assignments of the original formula. ∎

3. A Good Algorithm for Separable Permutations

A pattern P is said to be *separable* if it contains neither the subpattern $(3,1,4,2)$ nor the subpattern $(2,4,1,3)$. We begin with a lemma that clarifies the sense of the term "separable", and a discussion of the relationship between separable permutations and a class of graphs called P_4-free graphs.

3.1. Separable Permutations

A *separating tree* for a permutation $(p_1, \ldots, p_k)$ of $1, \ldots, k$ is an ordered binary tree $\mathcal{T}$ with leaves $(p_1, \ldots, p_k)$ in that order, such that for each node V, if the leaves of the subtree rooted at V are $p_i, p_{i+1}, \ldots, p_{i+j}$, then the set of numbers $\{p_i, p_{i+1}, \ldots, p_{i+j}\}$ is a *subrange* of the range $1, \ldots, k$—i.e. is of the form $\{l, l+1, \ldots, l+j\}$, $1 \le l, l+j \le k$. This subrange is called *the range* of the node V. If V is a node of the tree with left child V_L and right child V_R, then the above condition implies that either the range of V_L just precedes the range of V_R, or vice versa. In the first case V is called a *positive* node, and in the second case a *negative* node.

Lemma 3.1. A pattern $P = (p_1, \ldots, p_k)$ is separable iff it has a separating tree.

The separating tree need not be unique. For example, the permutation $(4,5,3,1,2)$ has the two separating trees $(((4,5),3),(1,2))$ and $((4,5),(3,(1,2)))$. See Figure 3.1 for an example of a separating tree.

A graph-theoretic formulation of this result is well-known: From any permutation T of $1, \ldots, n$, a graph can be defined with vertices $1, \ldots, n$, and with an edge (i,j) for $i < j$ iff i appears after j in the permutation. The graphs formed this way are *permutation graphs*. It is easy to see that the

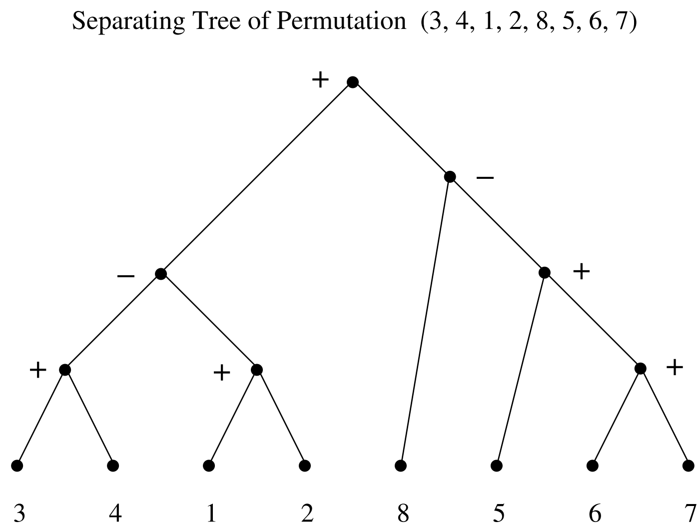

Figure 3.1

permutation containing the pattern $(3,1,4,2)$ or $(2,4,1,3)$ is equivalent to the graph containing an induced path on 4 vertices—a P_4. In general, graphs that do not contain an induced path on 4 vertices are called P_4-*free* graphs. Lemma 3.1 follows from the result that a graph is a P_4-free graph iff it is a *cograph*—a graph constructible from single vertices by the operations of disjoint union and complementation [CLS]. The use of a tree to represent this construction is explicit in [CLS]. This characterization of P_4-free graphs also implies that any P_4-free graph is a permutation graph. A linear time algorithm to recognize P_4-free graphs was given in [CPS]. This provides a linear time algorithm to recognize separable permutations.

Since we begin with a permutation rather than a graph, the results we need are significantly simpler, and thus it seems worthwhile—especially for those not already familiar with cographs—to give below a direct proof of Lemma 3.1 and a direct algorithm to recognize separable permutations.

It is tempting to think that the permutation pattern matching problem translates to the subgraph isomorphism problem for a cograph in a permutation graph. This is not the case, since a permutation graph does not uniquely determine the permutation. For example, the permutations $(3,1,4,2)$ and $(2,4,1,3)$ yield the same permutation graph.

An alternative graphical interpretation is to form the *poset* of the permutation, where i *precedes* j if i appears before j in the permutation and i is smaller than j. The poset of a permutation retains more information than the permutation graph, but still not enough to recover the permutation, and thus the permutation pattern matching problem is not equivalent to the poset sub-isomorphism problem.

Proof of Lemma 3.1. If the permutation does contain the subpattern $(3,1,4,2)$ or $(2,4,1,3)$

then there is no separating tree, since no two consecutive elements of these patterns form a range.

It remains to prove that any permutation either has a separating tree or has a subpattern $(3,1,4,2)$ or $(2,4,1,3)$. Suppose some permutation has neither of the subpatterns. By induction, it suffices to prove that there is a consecutive pair p_i, p_{i+1} forming a range.

Look at p_1 and p_2. Suppose without loss of generality that $p_1 < p_2$. If they form a range, fine; otherwise there is some $i > 2$ for which $p_1 + 1 = p_i$. Let j be the smallest index such that the substring $R = (p_2, p_3, \ldots, p_j)$ contains all the values in the range $p_i, p_i + 1, \ldots, p_2$. It is possible that $i = j$. We have $p_1 < p_i \le p_j < p_2$. If the set of values in R forms a range then by induction R contains a consecutive pair forming a range, since the patterns $(3,1,4,2)$ and $(2,4,1,3)$ are excluded.

Thus we may assume that R does not form a range. There are two possibilities: (1) R contains a small value—i.e. there is an l in $2, 3, \ldots, j$ with $p_l < p_1$; or (2) R contains a large value, but not all values before it—i.e. there is an l in $2, 3, \ldots, j$ and an $m > j$ with $p_2 < p_m < p_l$. In case (1) the subsequence p_1, p_2, p_l, p_j has the pattern $2, 4, 1, 3$; and in case (2) the subsequence p_2, p_l, p_j, p_m has the pattern $2, 4, 1, 3$. ∎

Separable permutations can be viewed as permutations sortable in a particular way. This is somewhat analogous to the characterization by Knuth [K, Section 2.2.1] that permutations without the pattern $2, 3, 1$ are exactly the permutations sortable using one stack; and to the further characterizations by Tarjan [T], by Even and Itai [EI], and by Pratt [P] (see also [G, chapter 7], and for a more recent result see [W]).

Separable permutations are exactly the permutations that can be sorted in the following way. Imagine a straight segment of railway track, with the elements of the permutation lined up on the track, uncoupled, so that any element can be moved back and forth, though without changing the relative ordering of elements. Imagine a segment in the middle of the track that can be rotated $180°$, thus reversing the order of the elements on the middle segment. Using this device, we may move any consecutive subsequence of elements onto the middle section of track and reverse their order. There is one restriction: a sequence of elements to be rotated on the middle section must first be coupled together and must remain coupled forever afterwards. Separable permutations are exactly the permutations that can be sorted using this method. See Figure 3.2 for an example. An efficient algorithm for sorting in this way is described below. It provides a test for whether a permutation is separable. (Large railway switching yards usually have these devices, implemented as a huge rotatable disc, though they generally have many segments of track converging radially on the disc, and of course do not permit an unlimited number of cars on the disc at one time.)

Lemma 3.1 provides a linear time algorithm to test if a permutation P is separable: Use a stack S whose elments are subranges $l, l+1, \ldots, l+m$ of the range $1, \ldots, k$. In general, to add a range r to S, see if the top element of S forms a larger range together with r—i.e. the union

Sorting Permutation (2, 1, 4, 5, 3)

Add 2 : Push 2	Stack : 2
Add 1 : Pop 2, [Add (1, 2) : Push (1, 2)]	Stack : (1, 2)
Add 4 : Push 4	Stack : (1, 2) 4
Add 5 : Pop 4, [Add (4, 5) : Push (4, 5)]	Stack : (1, 2) (4, 5)
Add 3 : Pop (4, 5), [Add (3, 4, 5) : Pop (1, 2), [Add (1, 2, 3, 4, 5) : Push (1, 2, 3, 4, 5)]]	Stack : (1, 2, 3, 4, 5)

Railway Sort : 2 1 4 5 3 $\rightarrow$ (1 - 2) 4 5 3 $\rightarrow$ 1 - 2 (5 - 4) 3 $\rightarrow$
1 - 2 (3 - 4 - 5) $\rightarrow$ 1 - 2 3 - 4 - 5

( ) indicates reversal

\- indicates coupling

Figure 3.2

of the two sets of numbers is a range. If so, pop the top element of S, form the combined range, and recursively add it to S; and if not, then push the range r onto S. The algorithm proceeds by scanning through P once, making each element of S into a singleton range and adding that range to S, as just described. If, at the end of P, the stack S contains a single range then the permutation is separable, and otherwise it is not. See Figure 3.2 for an example. P's separating tree $\mathcal{T}$ can easily be recovered.

To end this section we mention one other result about separable permutations. Shapiro and Stephens [SS1] consider the permutation matrices which completely fill up under "bootstrap percolation" (the process of filling in a 1 in a position where at least 2 of the 4 neighbours are 1). They show that the permutations whose matrices fill up are exactly the permutations with a separating tree (hence separable, by the above), and they show that the number of such n-permutations is the $(n-1)^{\text{st}}$ Schröder number.

3.2. Pattern Matching with Separable Permutations

We will now describe the algorithm to match a separable permutation pattern $P = (p_1, \ldots, p_k)$ into a text permutation $T = (t_1, \ldots, t_n)$. In fact, the number of such matches will be computed. The technique is dynamic programming, and the problem is subdivided based on P's separating tree, $\mathcal{T}$. For each node V of $\mathcal{T}$, try to match the substring corresponding to the subtree rooted at V—say, $(p_l, \ldots, p_m)$—into the text T. More specifically, solve one subproblem for each node V as above, and for each substring $(t_i, \ldots, t_j)$ of T, and each subrange $a, a+1, \ldots, b$ of the range $1, \ldots, n$. For each such subproblem, count the number of matches, $M(V, i, j, a, b)$, of the pattern $(p_l, \ldots, p_m)$ into the text $(t_i, \ldots, t_j)$, *using* t_i, and using text values in the range $a, \ldots, b$ *including*

a. Note that there are $O(kn^4)$ such subproblems, since the tree $\mathcal{T}$ has $O(k)$ nodes, and there are $O(n)$ choices for each of i, j, a, b. After solving the subproblems we can recover the number of matchings by summing $M(\text{Root}, i, n, a, n)$ over all values of i and a.

We will show how to compute $M(V, i, j, a, b)$ based on the values of M for the children of node V. (Note that solutions to the subproblems for the leaves of the tree $\mathcal{T}$ are immediately obtainable.) Let V_L and V_R be the left and right children of V, respectively. If node V is a positive node, in the sense defined above—i.e. the range of values for V_L precedes the range of values for V_R, then

$$M(V, i, j, a, b) = \sum\{M(V_L, i, h-1, a, c-1) \cdot M(V_R, h, j, c, b) : i < h \leq j, a < c \leq b\}$$

On the other hand, if V is a negative node then

$$M(V, i, j, a, b) = \sum\{M(V_L, i, h-1, c, b) \cdot M(V_R, h, j, a, c-1) : i < h \leq j, a < c \leq b\}$$

Thus it takes $O(n^2)$ arithmetic steps to compute $M(V, i, j, a, b)$ from previously computed values. Altogether, the running time of the dynamic programming algorithm is $O(kn^6)$.

Rather than just computing the number of matchings, the algorithm can easily be augmented to compute a matching (if there is one) or to compute a list of all matchings—in the latter case the running time will have an additional factor of the number of matchings. For the case of recovering one matching, it is enough to record for each $M(V, i, j, a, b)$ that is non-zero, values of h and c in the above formulas that cause it to be non-zero.

References

[CLS] D.G. Corneil, H. Lerchs, L. Stewart Burlingham, Complement-reducible graphs, Discrete Applied Math. 3, 163–174, 1981.

[CPS] D.G. Corneil, Y. Perl, L.K. Stewart, A linear recognition algorithm for cographs, SIAM J. Computing 14, 926–934, 1985.

[CW] M.-S. Chang and F.-H. Wang, Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs, Information Processing Letters 43, 293–295, 1992.

[EI] S. Even and A. Itai, Queues, stacks and graphs, in *Theory of Machines and Computations*, ed. Z. Kohavi and A. Paz, Academic Press, New York, 71–86, 1971.

[GJ] M.R. Garey and D.S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[G] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[K] D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd edition, Addison-Wesley, 1973.

[L] L. Lovász, *Combinatorial Problems and Exercises*, North-Holland, 1979.

[P] V.R. Pratt, Computing permutations with double-ended queues, parallal stacks, and parallel queries, Fifth ACM Symposium on Theory of Computing, 268–277, 1973.

[R] D. Rotem, Stack-sortable permutations, Discrete Math, 33, 185–196, 1981.

[SS1] L. Shapiro and A.B. Stephens, Bootstrap percolation, the Schröder number, and the N-kings problem, SIAM J. Discrete Math. 2, 275–280, 1991.

[SS2] R. Simion and F.W. Schmidt, Restricted permutations, European J. Combinatorics 6, 383–405, 1985.

[S] Z. Stankova, Forbidden subsequences, to appear, Discrete Math.

[T] R. Tarjan, Sorting using networks of queues and stacks, J. Assoc. Computing Machinery 19, 341–346, 1972.

[W] J. West, Generating trees and the Catalan and Schröder numbers, to appear, Discrete Math.

[W2] J. West, Generating trees and forbidden subseguences, extended abstract, poster session, 6th Conference Formal Power Series, DIMACS, 1994.