Algorithmica (1990) 5: 313–323

Algorithmica

A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings[1]

Esko Ukkonen[2]

Abstract. Approximate shortest common superstrings for a given set R of strings can be constructed by applying the greedy heuristics for finding a longest Hamiltonian path in the weighted graph that represents the pairwise overlaps between the strings in R. We develop an efficient implementation of this idea using a modified Aho–Corasick string-matching automaton. The resulting common superstring algorithm runs in time $O(n)$ or in time $O(n \min(\log m, \log|\Sigma|))$ depending on whether or not the goto transitions of the Aho–Corasick automaton can be implemented by direct indexing over the alphabet Σ. Here n is the total length of the strings in R and m is the number of such strings. The best previously known method requires time $O(n \log m)$ or $O(n \log n)$ depending on the availability of direct indexing.

Key Words. Shortest common superstring, Approximation algorithm, Linear-time algorithm, Greedy heuristics, Hamiltonian path.

1. Introduction. Let R be a finite set of strings over some (finite or infinite) alphabet Σ. A string w is a *common superstring* of strings R if each string x in R is a substring of w, that is, w can be written as uxv for some u and v. The *shortest common superstring* (SCS) problem is to find a shortest w. An obvious application area for SCS algorithms is data compression. Molecular biology is another such area. DNA sequencing leads to a fragment assembly problem whose natural abstract formulation comes close to the SCS problem [5], [7]–[9].

As the SCS problem is NP-complete [3], [4], we are interested in polynomial-time approximation algorithms. Such algorithms construct a common superstring w which is not necessarily a shortest one. In [10] and [11] such an approximation algorithm is developed and analysed.

The algorithm is based on the following "greedy" idea [2]. Find and remove two strings in R that have the longest mutual overlap among all possible pairs of strings in R. Then form the overlapped string from the removed two strings and replace it back in R. Repeat until there is only one string in R or no two strings have a nonempty overlap. This and some other methods are also studied in [12].

The quality of the approximation algorithm could be evaluated in terms of the length $|w|$ of the resulting common superstring w. Another possibility is to use

[1] This work was supported by the Academy of Finland.

[2] Department of Computer Science, University of Helsinki, Teollisuuskatu 23, SF-00510 Helsinki, Finland.

Received May 1, 1987; revised May 2, 1988. Communicated by Robert Sedgewick.

the total overlap inherent in w which we call the *compression.* If n denotes the total length of the strings in R, then the compression in w is defined as $n-|w|$. Obviously, compression is the largest possible when w is the shortest possible.

The following performance guarantee is proved in [10]-[12]: the compression in the common superstring w constructed by the greedy heuristics is at least half of the compression in an SCS. On the other hand, bounding the more natural performance measure, namely the ratio between the length of w and the length of an SCS, remains open. In other words, it is not known whether or not the greedy method is an ε-approximation algorithm.

An implementation for the greedy heuristics, running in time $O(mn)$, is described by Tarhio and Ukkonen in [11]. Here m is the number of strings in R. They used the Knuth-Morris-Pratt string-matching algorithm [6] for finding the pairwise overlaps between strings in R and a bucket sort for sorting them. Turner [12] develops asymptotically faster implementation achieving $O(n \log m)$ time for alphabets Σ that are small enough to allow direct indexing over Σ and $O(n \log n)$ otherwise, based on suffix trees and lexicographic splay trees.

The purpose of the present paper is to give an asymptotically optimal implementation, based on the Aho-Corasick string-matching machine [1]. The algorithm runs in linear time $O(n)$ for small alphabets, such that the Aho-Corasick trie can be implemented by direct indexing, and in time $O(n \cdot \min(\log m, \log|\Sigma|))$ for arbitrary alphabets.

This paper is organized as follows. Section 2 formulates the greedy method as a shortest-path heuristics in certain graphs. Our idea of using the Aho-Corasick construction is sketched in Section 3. Detailed algorithms are given in Section 4 and their analysis is given in Section 5.

2. The Greedy Heuristics.

There is an *overlap* v between strings x and x', if v is both a suffix of x and a prefix of x', that is, $x=uv$ and $x'=vu'$. An overlap v is *maximal* if it is longest possible. An overlap is *self-overlap* if $x=x'$.

It is convenient to represent the pairwise overlaps between strings in R as an overlap graph, defined as follows. The *overlap graph* for $R=\{x_1,\ldots,x_m\}$ is a directed, weighted graph with node set R, and with directed arcs (x_i,x_j), $i\neq j$. Each arc (x_i,x_j) has weight w_{ij} where w_{ij} is the length of the maximal overlap between strings x_i and x_j.

Besides using the overlaps defined above there is another possibility of merging strings in R, namely that a string x is a substring of another string y. Then $y=uxv$ for some u, v. We call R *reduced* if there is no such strings x in R.

It is immediate that R and reduced R have the same SCS. Therefore we will remove substrings at the beginning of the common superstring construction.

Common superstrings for a reduced R and Hamiltonian paths in the overlap graph are closely related. Let H be some directed Hamiltonian path in the overlap graph. Hence H is a (acyclic) path that goes through each node exactly once. From H we may construct a common superstring for R: just place strings in R above each other in the order they appear on path H. The successive x_i's must be maximally overlapped, that is, the length of the overlap equals the weight of

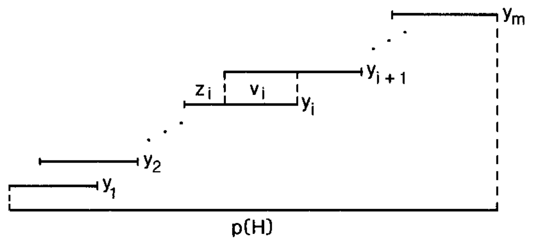

Fig. 1. Construction of a common superstring.

the corresponding arc on the path. Let $y_1, \ldots, y_m$ be the strings in R, written in the order they appear in H. The situation is as shown in Figure 1. If v_i is the maximal overlap between y_i and y_{i+1}, then y_i can be written as $y_i = z_i v_i$ for some z_i. The common superstring $p(H)$ for R constructed from H is then $z_1 z_2 \cdots z_{m-1} y_m$.

The length of $p(H)$ is $(|x_1| + \cdots + |x_m|) - |H|$ where $|H|$ denotes the sum of the weights on path H and $|x_i|$ is the length of x_i. Thus the larger $|H|$ is, the shorter is the corresponding $p(H)$. In fact, the following is true.

THEOREM 1 [11]. *An SCS for a reduced R is string p(H) where H is a longest Hamiltonian path in the overlap graph for R.*

The NP-hardness of the SCS problem [3] implies, noting Theorem 1, that also finding maximal Hamiltonian paths in overlap graphs is NP-hard. We have to look at approximation algorithms. The greedy heuristics delineated in Section 1 is equivalent to the following well-known greedy heuristics for longest Hamiltonian paths.

To construct a Hamiltonian path, select an arc e from the remaining arcs of the overlap graph such that

(i) e has the largest weight, and
(ii) e together with the arcs selected earlier forms a subgraph which can be expanded to a Hamiltonian path.

Repeat this until a Hamiltonian path H containing all nodes has been constructed.

This procedure can be implemented by processing the arcs in decreasing order of weight. The next arc (x, y) is rejected if there is an arc already in H starting from x or ending at y, or (x, y) together with the arcs already in H forms an oriented cycle. Otherwise (x, y) is selected to H.

3. Outline of the Method.

In the greedy heuristics it is essential to find and process the overlaps in decreasing order. In this section we explain, using an example, how this can be done very fast using the Aho–Corasick string-matching machine.

The Aho–Corasick string-matching machine [1], or the *AC machine*, generalizes the Knuth–Morris–Pratt algorithm for a set of key words. Let us assume that we

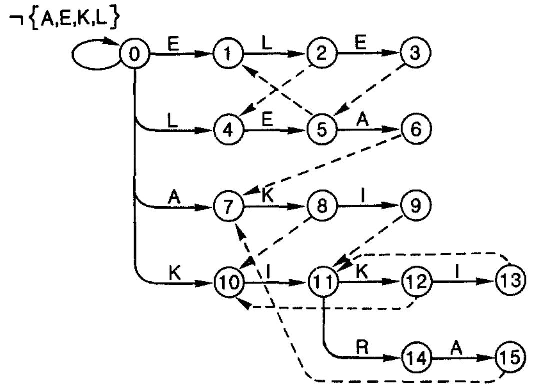

Fig. 2. The AC machine.

have to find all occurrences of key words

1. AKI
2. ELE
3. KIKI
4. KIRA
5. LEA

in some text string over the ordinary alphabet. This can be done by scanning the text only once with the AC machine shown in Figure 2.

The AC machine behaves like a finite-state automaton. There are two types of transitions: *goto transitions* and *failure transitions*. The goto transitions, shown as solid arrows in Figure 2, are normal finite-state automaton moves. There can be a goto transition from each state for each symbol in the alphabet. We write $g(s, a) = t$ if there is a goto transition from state s to state t for symbol a. If there is no applicable goto transition in some state, then the machine uses a failure transition. Except for the start state 0 (which has a goto transition for every symbol), there is exactly one failure transition starting from each state. The failure transitions that do not go to the start state 0 are shown by dashed arrows in Figure 2 while those pointing to 0 are omitted. A failure transition is a so-called ε-move which does not consume any symbol from the string scanned. We write $f(s) = t$ if there is a failure transition from s to t.

When the example machine scans text AKIRAT, the state transitions are as shown in Figure 3.

When entering state 9 the machine announces recognition of the key word number 1 (AKI) and when entering state 15 recognition of the key word number 4 (KIRA).

```
  A   K   I   R   A   T
0   7   8   9  14  15   0
           11       7
                    0
```

Fig. 3. State transitions for AKIRAT.

We say that string u *represents* state s of the AC machine if the shortest path in the goto graph from the start state to state s spells out u. For example, string KIRA in Figure 2 represents state 15.

Aho and Corasick [1] prove the following lemma.

LEMMA 2. *Suppose that state s is represented by string u and state t by string v. Then, $f(s) = t$ if and only if v is the longest proper suffix of u that is also a prefix of some key word.*

This means that if $f(s) = t$, then there is an overlap v between u and each key word x_j such that the path spelling out x_j contains state t. By repeatedly applying the failure function f from state s we can find all overlaps between u and the key words. In particular, let u itself be some key word. Then we can see how the AC machine encodes the pairwise overlaps: the overlaps can be found by following the failure paths from the states represented by the key words. This observation can be done precise as follows.

For each state s, let $L(s)$ be the set of indices i such that the goto path, that spells out the ith key word x_i, contains state s. We call $L(s)$ the set of the *supporters* of s. In Figure 2, $L(11) = \{3, 4\}$ and $L(0) = \{1, 2, 3, 4, 5\}$. Also let $d(s)$ denote the *depth* of state s, that is, the length of the string that represents s.

LEMMA 3. *Let string u represent state s. For all strings x_j in R, there is an overlap of length k between u and x_j if and only if, for some $h \geq 0$, state $t = f^h(s)$ is such that j is in $L(t)$ and $k = d(t)$.*

PROOF. By induction on $d(s)$.

Let first $d(s) = 0$, that is, u is the empty string and s is the start state. There is an empty overlap between u and all strings x_j. On the other hand, t must be the start state. Then every j is in $L(t)$ and $d(t) = 0$. Hence the lemma is true.

Assume then $d(s) \geq 1$. Let v be an overlap of length k between u and some x_j. We have three cases:

Case 1: $v = u$. The lemma is true with $h = 0$, since j is in $L(s)$ when $v = u$.

Case 2. String v is the longest proper suffix of u that is also a prefix of some string in R. By Lemma 2, this is true if and only if $f(s) = t$ where v represents state t and hence j is in $L(t)$. Since $d(t) = k$, the lemma is true with $h = 1$.

Case 3. String v is properly shorter than the longest proper suffix v' of u that is also a prefix of some string in R. Let t' be the state represented by v'. By Lemma 2,

$$f(s) = t'. \tag{1}$$

Since $|v| < |v'|$, there is overlap v also between v' and x_j. Since $d(t') = |v'| < |u| = d(s)$, it follows by the induction hypothesis that the overlap v between v' and x_j exists if and only if, for some $h \geq 1$, state $t = f^h(t')$ is such that j is in $L(t)$ and $d(t) = k$. Noting (1), this is true if and only if, for some $h > 1$, state $t = f^h(s)$ is such that j is in $L(t)$ and $d(t) = k$. □

The depth of the states on a failure transition path is monotonically decreasing by Lemma 2. Then if u is replaced by x_i in Lemma 3, the next result follows:

THEOREM 4. *Let strings x_i and x_j be in R and let x_i represent state s. The maximal overlap between x_i and x_j is of length k if and only if $k = d(t)$ where t is the first state on the failure transition path from s such that j is in $L(t)$.*

Theorem 4 suggests how the greedy heuristics can be implemented using the AC machine. To find the pairwise overlaps in descending order, follow all failure paths starting from the states represented by the strings in R. All paths have to be followed simultaneously and the longest available overlap has to be selected at each stage. This suggests an implementation as follows. Traverse the states of the AC machine in reversed breadth-first order. At each state we have to know which of the failure paths pass through that state. Each passing path represents an overlap. Choose any overlap that is not forbidden by earlier selections.

In our example the form of the resulting common superstring depends on the order in which equally long overlaps are selected. The result can be string

ELEAKIRAKIKI

where the compression is 5, or a string (which is the SCS)

ELEAKIKIRA

where the compression is 7.

4. The Method.

We now give a precise formulation of the method delineated in the previous section. The presentation is divided into two parts: Algorithm 1 is a preprocessing phase that augments the usual AC machine with the necessary additional functions; Algorithm 2 implements the actual greedy heuristics.

Given a set $R = \{x_1, \dots, x_m\}$ of strings, we first construct the AC machine for it. This is done as described in [1]; as a result we get the goto function g and the failure function f. In addition, we have to compute the following items; note that now (item 5) we explicitly eliminate substrings to get a reduced R:

1. Depth $d(s)$ for each state s of the AC machine.
2. List $L(s)$ of the supporters for each state s.
3. The state $F(i)$ representing string x_i for each x_i in R. States $F(i)$ are the starting points of the failure paths which will be followed in Algorithm 2.
4. Reversed breadth-first ordering of the states. For each state s, this is represented by a link $b(s)$ giving the successor of s in this order. Pointer B will give the first state in this chain.
5. Elimination of the substrings to get reduced R. Two situations can indicate a substring. First, if there is a goto transition from some state s represented by a string x_i in R (i.e., $F(i) = s$), then x_i must be a prefix of some other string in R and can be removed. Second, assume there is a failure transition $f(s) = t$ such that $t = F(i)$, for some i and s, is supported by index j. Then by Lemma

1, x_i must be a substring of x_j and can be removed. A convenient way of representing the removal of x_i is to set $F(i)=0$. (State 0 is the root of the AC trie.)

Algorithm 1 performs the above computations. For brevity, we assume that functions g and f have already been constructed using the algorithms of [1]. Hence Algorithm 1 can use them as input.

ALGORITHM 1. Preprocessing.
Input. Set $R=\{x_1,\dots,x_m\}$ of strings, and the AC machine consisting of the goto function g and the failure function f for R.
Output. Depth $d(s)$, list $L(s)$, and link $b(s)$ for each state s of the AC machine; pointer B to the first state in the b-link chain; state $F(i)$ representing string x_i for each x_i with the exception that if x_i is a substring of another string in R then $F(i)=0$.
Notation. Operator $\cdot$ denotes list catenation. An inverse of F is represented by E: if $F(i)=s$, then $E(s)=i$; initially $E(s)=0$ for every state s.
Method.

```
1.   for $i \leftarrow 1$ to $m$ do
2.     let $x_i = a_1 \cdots a_k$
3.     $s \leftarrow 0$
4.     for $j \leftarrow 1$ to $k$ do
5.       $s \leftarrow g(s, a_j)$
6.       $L(s) \leftarrow L(s) \cdot \{j\}$
7.       if $j = k$ then
8.         $F(i) \leftarrow s$
9.         $E(s) \leftarrow i$
10.        if $s$ is not a leaf of the AC machine
             then $F(i) \leftarrow 0$
           fi
         fi
       od
     od
11.  queue $\leftarrow 0$
12.  $d(0) \leftarrow 0$
13.  $B \leftarrow 0$
14.  while queue $\neq$ empty do
15.    let $r$ be the next state in queue
16.    queue $\leftarrow$ queue $- r$
17.    for each $s$ such that $g(r, a) = s$ for some $a$ do
18.      queue $\leftarrow$ queue $\cdot s$
19.      $d(s) \leftarrow d(r) + 1$
20.      $b(s) \leftarrow B$
21.      $B \leftarrow s$
22.      $F(E(f(s))) \leftarrow 0$
       od
     od
```

The next step after Algorithm 1 is the greedy selection of the overlaps. The selected overlaps should form a Hamiltonian path H in the overlap graph. We traverse the states of the AC machine in reversed breadth-first order. This is done by following the b-links. Moreover, we maintain lists $P(s)$ at each state s. When the traversal comes to state s, list $P(s)$ will contain all indices i such that

1. s is on the failure path that starts from $F(i)$, and
2. path H does not yet contain an overlap (x_i, x_j) for any j.

Hence we know at this stage that, for each i in $P(s)$ and for each j in $L(s)$, there is an overlap (x_i, x_j) of length $d(s)$ and that, for each such i, H does not yet contain an overlap starting from x_i. Next we check if some overlap (x_i, x_j) is such that

(i) H does not contain an overlap ending at x_j, and
(ii) (x_i, x_j) together with the overlaps already in H does not create a cycle.

For each j in $L(s)$ satisfying (i) (this will be indicated by flag forbidden(j)) we traverse list $P(s)$ until the first i is found such that (x_i, x_j) satisfies (ii). Fortunately, for each j we have to examine only at most two indices i in $P(s)$. If the first i creates a cycle, the second i cannot do it with the same j. Then (x_i, x_j) is added to H, and forbidden(j) is made true; initially forbidden(j) is true only if x_j is a substring. Moreover, index i is removed from $P(s)$.

We also need a data structure that allows us to check condition (ii) in constant time. We maintain tables FIRST(i) and LAST(i), $i = 1, \ldots, m$. At any moment, if H contains a path from x_i to x_j and no arc in H ends at x_i or starts from x_j, then FIRST$(j) = i$ and LAST$(i) = j$. Initially FIRST$(i) =$ LAST$(i) = i$ for all i.

Assume that we next try to add (x_i, x_j) to H. If now FIRST$(i) = j$, we know (x_i, x_j) would create a cycle and should be rejected. Otherwise (x_i, x_j) satisfies (ii) and can be accepted.

When (x_i, x_j) is added to H, the path ending at x_i becomes catenated with the path starting from x_j. Hence FIRST and LAST must be updated in an obvious way.

Finally, when all overlaps at s have been processed, $P(s)$ is catenated with $P(f(s))$ so that processing of the still active failure paths that pass s can be continued when the traversal reaches $f(s)$. Now we are finished with s and can move to $b(s)$.

Algorithm 2 gives the resulting procedure in detail.

ALGORITHM 2. Construction of H.
Input. Augmented AC machine for R, as constructed by Algorithm 1.
Output. A Hamiltonian path H in the overlap graph of reduced R. As explained in Section 2, a common superstring for R can then be constructed by forming $p(H)$.

Method. Let initially each $P(s)$ be empty.

```
1.   for $j \leftarrow 1$ to $m$ do
       if $F(j) \neq 0$ then $P(f(F(j))) \leftarrow P(f(F(j))) \cdot \{j\}$
                    $\text{FIRST}(j) \leftarrow \text{LAST}(j) \leftarrow j$
                 else forbidden$(j) \leftarrow$ true
       fi
     od
2.   $s \leftarrow b(B)$
3.   while $s \neq 0$ do
4.     if $P(s)$ is not empty then
5.       for each $j$ in $L(s)$ such that forbidden$(j) =$ false do
6.         $i \leftarrow$ the first element of $P(s)$
7.         if $\text{FIRST}(i) = j$ then
8.           if $P(s)$ has only one element then goto next
             else $i \leftarrow$ the second element of $P(s)$
           fi
9.         $H \leftarrow H \cdot \{(x_i, x_j)\}$
10.        forbidden$(j) \leftarrow$ true
11.        $P(s) \leftarrow P(s) - \{i\}$
12.        $\text{FIRST}(\text{LAST}(j)) \leftarrow \text{FIRST}(i)$
13.        $\text{LAST}(\text{FIRST}(i)) \leftarrow \text{LAST}(j)$
         next:
         od
14.      $P(f(s)) \leftarrow P(f(s)) \cdot P(s)$
       fi
15.    $s \leftarrow b(s)$
     od
```

5. Time Analysis. In the analysis we assume that every $L(s)$ and $P(s)$ is represented as a linked list such that catenations and deletions can be performed in constant time. Also recall that n denotes the total length of the strings in R.

THEOREM 5. *The AC machine of R can be constructed and preprocessed with Algorithm* 1 *in time $O(n)$ if alphabet Σ is small enough such that the goto function g can be implemented by direct indexing, and in time $O(n \cdot \min(\log m, \log|\Sigma|))$ if direct indexing on elements of Σ is not applicable.*

PROOF. The innermost loop (steps 5–10) of the first part of Algorithm 1 is performed n times. Each takes constant time except possibly the time of applying function g in step 5 and the time of testing whether s is a leaf in step 10.

The latter can be done in constant time if we make the assumption that there are flags indicating leaves. Such flags can be set during the construction of g.

Step 5 needs time $O(1)$ if direct indexing is used and time $O(\min(\log m, \log|\Sigma|))$ if using a search tree. Hence the total time so far is $O(n)$ or

$$O(n \cdot \min(\log m, \log|\Sigma|)).$$

The innermost loop (steps 18-22) of the second part of Algorithm 1 is performed once for each goto transition, hence $O(n)$ times. Each obviously takes constant time provided that each new s in step 17 is found in constant time. Here we may assume that each state r has a list containing all its children, that is, all states s such that $s = g(r, a)$ for some a. Such links can be set during the construction of g without increasing the asymptotic time complexity. Then the total time for the second part is $O(n)$, independently of $|\Sigma|$.

To complete the proof we recall from [1] that g and f can be constructed in time $O(n)$ if direct indexing is possible, and in time $O(n \cdot \min(\log m, \log|\Sigma|))$ otherwise. Finally, we point out that it is clearly possible to merge Algorithm 1 with the construction of g and f with no increase in asymptotic time requirement. Hence, basically, the total time bound depends on the time needed to construct g and f. □

THEOREM 6. *Algorithm 2 runs in time $O(n)$.*

PROOF. The innermost loop starting at step 5 dominates in the total time. The loop is performed at most once for each j in each $L(s)$, hence altogether $O(\sum_s |L(s)|) = O(n)$ times. The body of the loop clearly takes constant time which completes the proof. That it is correct to examine only at most two elements of $P(s)$ for each j has already been shown in the discussion of Algorithm 2 in Section 4. □

Theorems 5 and 6 give our conclusion.

COROLLARY. *The greedy heuristics for the SCS problem can be implemented in time $O(n)$ or in time $O(n \cdot \min(\log m, \log|\Sigma|))$ depending on whether or not direct indexing on elements of Σ is applicable in implementation of the goto transitions of the AC machine.*

6. Conclusion.

The greedy heuristics seems to be the only polynomial-time approximation algorithm for the SCS problem for which a uniform performance guarantee is known [10]-[12]: the algorithm constructs common superstrings with at least half of the maximal compression. Finding efficient implementations is therefore important.

We developed in this paper a relatively simple new implementation, which in most applications will run in linear time. First we exhibited an equivalent formulation for the SCS problem as a problem of finding a maximal Hamiltonian path in the overlap graph. Then we showed in detail how this path problem can be approximately solved very swiftly using a slightly modified Ahc Corasick string-matching automaton.

Acknowledgment. The author is indebted to the referee whose comments were helpful in considerably improving the presentation of this paper.

References

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

[2] J. K. Gallant: String Compression Algorithms. Ph.D. Thesis, Princeton University, Princeton, NJ, 1982.

[3] J. Gallant, D. Maier, and J. A. Storer: On finding minimal length superstrings. *J. Comput. System Sci.* **20** (1980), 50-58.

[4] M. R. Garey and D. S. Johnson: *Computers and Intractability*. Freeman, San Francisco, 1979.

[5] T. R. Gingeras, J. P. Milazzo, D. Sciaky, and R. J. Roberts: Computer programs for the assembly of DNA sequences. *Nucleic Acids Res.* **7** (1979), 529-545.

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

[7] H. Peltola, J. Söderlund, J. Tarhio, and E. Ukkonen: Algorithms for some string-matching problems arising in molecular genetics, in *Information Processing* (R. E. A. Mason, ed.). Elsevier Science, Amsterdam, 1983, pp. 59-64.

[8] H. Peltola, H. Söderlund, and E. Ukkonen: SEQAID: a DNA sequence assembling program based on a mathematical model. *Nucleic Acids Res.* **12** (1984), 307-321.

[9] R. Staden: Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequencing. *Nucleic Acids Res.* **10** (1982), 4731-4751.

[10] J. Tarhio and E. Ukkonen: A greedy algorithm for constructing shortest common superstrings, in *Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, Vol. 233. Springer-Verlag, Berlin, 1986, pp. 602-610.

[11] J. Tarhio and E. Ukkonen: A greedy approximation algorithm for constructing shortest common superstrings. *Theoret. Comput. Sci.* **57** (1988), 131-145.

[12] J. S. Turner: Approximation Algorithms for the Shortest Common Superstring Problem. Technical Report WUCS-86-16, Department of Computer Science, Washington University, Saint Louis, MO, 1986.