To the Graduate Council:

I am submitting herewith a thesis written by Robert Scott Harris entitled Approximate String Matching With Pumping. I have examined the final copy of this thesis for form and content and recommend that it is accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jens Gregor, Major Professor

We have read this thesis and recommend its acceptance:

Heather Booth

Michael G Thomason

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature: Robert S. Han

Date: 5/19/93

Approximate String Matching With Pumping

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

ROBERT SCOTT HARRIS

AUGUST 1993

Acknowledgments

I gratefully thank my advisor Dr. Jens Gregor of the University of Tennessee for the suggestion of this project, and for guidance through the completion of it. I also thank my committee members Dr. Michael Thomason and Dr. Heather Booth. All three have greatly contributed to my education during the past year through the courses they have taught.

I would also like to take this opportunity to express my gratitude toward others who helped me get this far. To Ken Barto, Dr. Jon Bentley, Dave Vetter, and Sam Overton, thanks.

Finally, I thank my parents. For all those times you said I'd thank you some day—this is the day.

Abstract

The classic dynamic programming algorithm for computing edit distances between two strings is extended to compute the distance between a string and a class of strings represented by a regular expression. The result allows arbitrary pumping of symbols and substrings. The extended algorithm has the same complexity as the original—O(mn) where m and n are the lengths of the regular expression and the string.

Preface

The author began work that led to this thesis in November 1992, at the suggestion of Dr. Jens Gregor. The initial focus was to add single symbol pumping to approximate string matching. The broader goal was to then find a reasonably efficient algorithm allowing pumping of arbitrarily nested substrings. The result is that there is such an algorithm, and it has the same O(mn) computational complexity as the classic algorithm for the problem without pumping.

The paper by Miller & Myers [10], which the author did not discover until February 1993, demonstrates a similar algorithm and makes the same complexity claim. The author had already completed much independent work, had realized that the prototype should be formulated as a regular expression, and had created an algorithm of the same general form as [10].

The important difference is that the author had not yet determined a bound on the number of passes required for the computation of each column. As presented in [10], two passes are enough regardless of the complexity of the prototype (as long as the prototype is constructed from a regular expression). The author had discovered that two passes were enough for limited nesting of cycles (one cycle nested inside another), but had not shown (or even suspected) that two passes would be enough for the general case.

This thesis offers some improvements over [10]. Chapter 3 demonstrates that single symbol pumping can be implemented in a simple way. While [10] claims that two-passes are enough for substring pumping, the proof, which refers to [4], is not clear. Chapter 4 gives an independent proof (admittedly tedious) that is more closely related to the problem domain. Chapter 5 discusses representing regular expressions like **`a+b+c`** by a single row in the matrix.

Table of Contents

1. Introduction 1

1.1. Approximate String Matching 1

1.1.1. Background 1

1.1.2. Definitions 1

1.2. String Notation 3

1.3. Regular Expressions and Pumping 4

1.4. Related Problems 5

2. The Classic Algorithm 6

2.1. Algorithm 6

2.2. Example 8

2.3. Complexity 10

3. Pumping of Single Symbols 11

3.1. Definition 11

3.2. Algorithm 12

3.3. Example 13

3.4. Proof 15

3.5. Complexity 20

4. Pumping of Substrings 22

4.1. Definition 22

4.1.1. Prototypes from Regular Expressions 24

4.1.2. Properties of Prototypes 28

4.2. Algorithm 31

4.3. Examples 36

4.4. Proof 41

4.5. Complexity 52

5. Extensions 53

5.1. More Primitives 53

5.2. More Construction Rules 54

5.3. Simplifying Prototypes 54

5.4. Limiting Pumping Range 56

6. Conclusion 59

List of References 60

Appendix. Equivalence of Regular Expressions and Arbitrary Prototypes 63

A.1. Equivalence 64

A.2. Construction 64

Vita 68

List of Figures

Figure 1. Algorithm A: Classic Algorithm for Approximate String Matching..... 8

Figure 2. Algorithm B: Back-Tracing Optimal Alignments..... 9

Figure 3. Example Distance Matrix for Algorithm A..... 9

Figure 4. Algorithm C: Approximate String Matching in Linear Space.....10

Figure 5. Algorithm D: Approximate String Matching with Single Symbol Pumping.....14

Figure 6. Distance Matrix with Pumpable Symbol.....14

Figure 7. Arbitrary Prototypes.....23

Figure 8. Regular Expression and Corresponding Prototype.....25

Figure 9. Prototype Constructions for Primitive Regular Expressions 26

Figure 10. Prototype Construction for Concatenation of Regular Expressions.....26

Figure 11. Prototype Construction for Union of Regular Expressions.....27

Figure 12. Prototype Construction for Kleene Closure of a Regular Expression.....28

Figure 13. Algorithm E: Approximate String Matching with Substring Pumping.....35

Figure 14. Distance Matrix with Substring Pumping, and Optimal Alignments.....37

Figure 15. Another Distance Matrix with Substring Pumping.....39

Figure 16. Hierarchy of Subnetworks in Prototype.....41

Figure 17. Prototype Constructions for Lemmas 7 through 10.....43

Figure 18. Prototype Construction for Optional Regular Expression.....55

Figure 19. Prototype Construction for Positive Closure of a Regular Expression 55

Figure 20. Eliminating ε-Nodes 57

Figure 21. Equivalent Prototypes.....65

Figure 22. Algorithm F: Constructing Regular Expression from an NFA.....66

Nomenclature

Σ	An alphabet of symbols.
a, **b**, etc.	Specific symbols from Σ, shown in boldface to distinguish them from the surrounding text.
ε	The empty symbol or empty string (depending on context).
S	Some string of symbols from Σ.
$\|S\|$	The number of symbols in S.
S_j	The symbol in the j^{th} position of S. S_1 is the first symbol. S_0 is considered to be the empty symbol ε.
$S_{i:j}$	The substring consisting of the symbols in the i^{th} through j^{th} positions of S. $S_{0:i}$ is a prefix of S, while $S_{i:\|S\|}$ is a suffix of S. $S_{0:0}$ is considered to be the empty string ε.
$S^{\bullet}$	The longest proper prefix of S, $S_{0:\|S\|-1}$.
T	The test string.
t	An unknown symbol from the test string, possibly empty.
n	The length of the test string $\|T\|$.
R	A regular expression.
$\hat{p}$	The labeled prototype string or prototype network.
P	The prototype string.
p	An unknown symbol from the prototype, possibly empty.
m	The length of the prototype, $\|P\|$.

1. Introduction

1.1. Approximate String Matching

1.1.1. Background

Given strings of symbols from some alphabet, an *edit distance* can be defined as a measure of similarity between strings. Given any two strings, there is some sequence of *edit operations* (an *edit sequence*) that transforms one string into the other. Intuitively, strings that are similar to each other will have shorter edit sequences available than strings that are dissimilar. Strings that are very similar are said to *approximately match* (in contrast with an identical match).

This paper relaxes edit distance to permit *pumping* of symbols or substrings in one of the strings. Pumping a non-empty substring represents a regular class of strings. The similarity of some string T to this class is determined by the class member most similar to T.

1.1.2. Definitions

For the purposes of this paper, the approximate string matching problem (without pumping) can be phrased as follows. We are given a prototype string P and test string T, of lengths m and n, from (possibly infinite) alphabet Σ. We are to determine the similarity of P and T (in a manner to be defined shortly), and an alignment demonstrating that similarity.

We define a set of edit operations represented as pairs of symbols from $\Sigma \cup \{\varepsilon\}$, each of which transforms a (possibly empty) symbol p in P into a (possibly empty) symbol t in T. Allowable operations are (p,ε)—delete symbol p, (ε,t)—insert symbol t, and

(p,t)—substitute t for p[1]. A given linear-time computable cost function d(p,t) associates a non-negative real cost with each operation. Typically d(p,t) = 0 whenever p = t, so that exact matches are treated as a special case of substitutions, having no cost.

Levenshtein [9] defines a simple class of cost functions where one cost is associated with all deletions, another cost with all insertions, and another with all substitutions. We denote these as $Lev(d_{del}, d_{ins}, d_{sub})$, and define them by

$$d(p,t) = \begin{cases} d_{del} \text{ if } t=\varepsilon \\ d_{ins} \text{ if } p=\varepsilon \\ 0 \quad \text{if } p=t \\ d_{sub} \text{ otherwise} \end{cases}$$

While the results of this paper are not this restrictive, the examples all assume Lev(1,1,1) which simplifies to

$$d(p,t) = \begin{cases} 0 \text{ if } p=t \\ 1 \text{ otherwise} \end{cases}$$

To measure similarity, we extend the cost function to create a distance function over strings. The cost of a sequence of operations E that transforms P to T is the sum of the costs of the individual operations,

$$d(E) = \sum_{k=1}^{k=|E|} d(P_k, T_k).$$

The edit distance d(P,T) is the smallest such cost[2].

As an example, consider the strings P=**bccac** and T=**cacbc**, and the distance function Lev(1,1,1). One way to transform P into T is to delete the **b**, insert an **a** after the first **c**, substitute a **b** for the existing **a**, and match all other symbols. This corresponds to the edit sequence ⟨(**b**,ε), (**c**,**c**), (ε,**a**), (**c**,**c**), (**a**,**b**), (**c**,**c**)⟩. The total cost is d(**b**,ε) +

---

1 Since (ε,ε) is generally useless, we consider it an invalid operation.

2 The reader should be aware that d is used as the name for the cost function of edit operations, the distance function of strings, and the cost function of edit sequences. This is the traditional usage found in the literature. In general, the usage is not ambiguous.

d(ε,**a**) + d(**a**,**b**), which is 3. From this we can conclude that d(**bccac**,**cacbc**) is no more than 3.

Wagner and Fischer [16] show a direct equivalence between edit sequences and *alignments* of the two strings. An alignment is a list of pairs of positions from the two strings corresponding to the match and substitute operations in the edit sequence. For example, an alignment of **bccac** and **cacbc** for the previous example is ⟨(2,1),(3,3),(4,4),(5,5)⟩, indicating that position 2 of **bccac** is matched with position 1 of **cacbc** , and so on. An advantage of alignments is that they can be displayed graphically, with the cost of each operation often attached. For example, the alignment of **bccac** and **cacbc** in the previous example is as follows[1]:

```
P: bc-cac
T: -cacbc
d: 1 1 1
```

The remainder of this paper uses alignments exclusively.

1.2. String Notation

This paper uses the following string notation. A string S is a series of symbols $S_1S_2...S_k$. |S| denotes k, the length of S. S is indexed by the integers 1 through |S|, so S_1 is the first symbol in S, $S_{|S|}$ is last, and S_i is the symbol in the i^{th} position. We may have occasion to refer to S_0, which we consider to be the empty symbol ε.

Substrings of S are denoted by $S_{i:j} = S_iS_{i+1}...S_j$. Prefixes are usually denoted by $S_{0:i}$ rather than $S_{1:i}$, as the former notation allows for the empty prefix $S_{0:0}$. We introduce the special notation $S^\bullet$ to indicate the *longest proper prefix* of S. $S^\bullet$ is S with the last symbol removed, $S_1S_2...S_{k-1}$.

---

[1] In alignments, the empty character ε is shown as a hyphen for visual convenience.

Strings of repeated symbols are denoted by superscripts. S_i^k is a string of k copies of the symbol S_i. k may be zero, in which case S_i^k is empty.

1.3. Regular Expressions and Pumping

A frequent use of approximate string matching is in pattern analysis systems. A class of strings is represented by some examples from the class, and test strings are then classified as class members if they are close to some string in the class. In many cases it is desirable to allow pumping of symbols or substrings, since the structure of the class may contain cycles. We permit pumping by using a regular expression as the prototype.

For example, d(**bc*ac**,**bcacbac**) is 2. A corresponding alignment is shown below. **c***, representing any number of consecutive **c**'s, is matched with **cac** (equivalent to substituting an **a** for a **c**) and a **b** is inserted before the **a**.

```
P: bccc-ac
T: bcacbac
d:   1 1
```

The problem is approached in two stages. Chapter 3 generalizes the prototype to allow pumping of single symbols (such as **c*** in the preceding example). Chapter 4 further allows pumping of substrings using regular expressions[1]. Chapter 5 discusses improvements to the regular expression matching.

For each problem, algorithms are developed based on the classic dynamic programming algorithm of Wagner and Fischer [16]. In both cases, the time and space complexity remain O(mn) as in the original algorithm (where m is the length of the regular expression). A significant practical contribution of this paper is the ability to allow single-symbol pumping with very little overhead.

---

[1] The appendix discusses arbitrary prototype networks and shows that regular expressions are sufficient to allow any arbitrary substring pumping.

1.4. Related Problems

Wang and Pavlidis [17] also match regular expressions, but represent the prototype by an arbitrary finite automaton. They show a worst case computational complexity of $O(s^3n)$ where s is the number of states in the automaton. This may compare favorably to the O(mn) algorithm when the number of states is small and the equivalent regular expression is large.

There are different viewpoints as to what constitutes a solution to the approximate string matching problem. This paper considers the case where both the distance and corresponding alignment(s) between any two strings is desired. The problem can be simplified if only the distance is needed. This is discussed briefly in this paper, as well as in [5]. Still another simplification is to simply report whether or not the distance is less than some threshold, as considered in [14].

Several authors have made use of restricted distance functions, typically Lev(1,1,1) or Lev(1,1,2). Special algorithms that are faster than the general case are then possible, as shown in [8], [14], [18], and [19]. For Lev(1,1,2), the problem can be solved by finding the longest common subsequence, as in [7]. [2] generalizes this result.

Another variation involves finding a substring in T which closely matches the prototype P. This is discussed in [12] and [19]. The algorithms presented herein can easily be modified along the lines in [12].

[1] shows an algorithm that takes advantage of strings consisting of runs of identical symbols.

A overview of several algorithms for the problem can be found in [3].

2. The Classic Algorithm

2.1. Algorithm

The distance d(P,T) and corresponding optimal alignments can be found by the classic dynamic programming algorithm due to Wagner and Fischer [16], Sellers [11], and others.

In the classic algorithm we iteratively compute the distance between longer and longer prefixes of P and T, until we have exhausted the entire strings. The premise of the algorithm is that the final operation in an optimal alignment of P and T has to correspond to a delete, insert, or substitute. The cost of such an alignment ending with a delete would be the cost of an optimal alignment of the prefix $P_{0:m-1}$ and T, plus the cost of the delete ($d(P_m,\varepsilon)$). The costs for alignments ending with an insert or a substitute can be similarly computed, and the minimum of these three costs is the desired distance.

Applying this logic to all prefixes of both strings leads to the recurrence relation due to Wagner and Fischer [16]:[1]

$$d(P_{0:i},T_{0:j}) = \min \begin{cases} \text{Ⓓ}\ d(P_{0:i-1},T_{0:j}) + d(P_i,\varepsilon) \\ \text{Ⓘ}\ d(P_{0:i},T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{Ⓢ}\ d(P_{0:i-1},T_{0:j-1}) + d(P_i,T_j) \end{cases} \qquad (2.1)$$

with boundary values defined by

$$\begin{aligned} d(P_{0:0},T_{0:0}) &= d(\varepsilon,\varepsilon) = 0 \\ d(P_{0:i},T_{0:0}) &= d(P_{0:i},\varepsilon) = \text{Ⓓ}\ d(P_{0:i-1},T_{0:0}) + d(P_i,\varepsilon) \\ d(P_{0:0},T_{0:j}) &= d(\varepsilon,T_{0:j}) = \text{Ⓘ}\ d(P_{0:0},T_{0:j-1}) + d(\varepsilon,T_j) \end{aligned} \qquad (2.2)$$

---

[1] The circled letters D, I, and S are not part of the formula, but are reminders of the edit operations justifying the formula— delete, insert, and substitute, respectively.

Lemma 1: The recurrence relations in (2.1) and (2.2) are correct.

$$\forall P,T,\ d(P,T) = \min \begin{cases} \text{(D)} & d(P^{\bullet},T) + d(P_m,\varepsilon) \\ \text{(I)} & d(P,T^{\bullet}) + d(\varepsilon,T_n) \\ \text{(S)} & d(P^{\bullet},T^{\bullet}) + d(P_m,T_n) \end{cases}$$

Proof: The premise of the proof is described above. Proof is given in [16]. ▫

Algorithm A, shown in figure 1, is the classic algorithm. A matrix D[0..m,0..n] is filled in so that D[i,j] = $d(P_{0:i},T_{0:j})$, the distance between prefixes $P_{0:i}$ and $T_{0:j}$. Lines 3 through 10 initialize the top row and left column of the matrix as per formula (2.2). Lines 11 through 21 fill the interior of the matrix using formula (2.1), determining each element from the three neighboring elements already determined. Upon completion, the desired distance d(P,T) is found at D[m,n].

Lines 7, 10, and 16 through 21 save links between elements in a matrix E[0..m,0..n] so that optimal alignments can be found by a simple back tracing algorithm. E[i,j] is a set of edit operations, each of which can be used as the last operation in some optimal alignment of $P_{0:i}$ and $T_{0:j}$.

The back-tracing algorithm (algorithm B, figure 2) recursively generates all possible optimal sequences of edit operations to a given element of D[i,j]. BackTrace(m,n,emptylist) will produce all the optimal edits that transform P to T. If only one such alignment is desired, lines 10 and 12 can be changed to else-ifs so that only one path is traced.

```
1)  D is a matrix of non-negative real numbers over [0..m,0..n]
2)  E is a matrix of subsets of {delete, insert, substitute} over [0..m,0..n]
3)  D[0,0] = 0
4)  E[0,0] = {}
5)  for 1≤i≤m
6)      D[i,0] = D[i-1,0] + d(Pi,ε)                    // delete
7)      E[i,0] = {delete}
8)  for 1≤j≤n
9)      D[0,j] = D[0,j-1] + d(ε,Tj)                    // insert
10)     E[0,j] = {insert}
11) for 1≤i≤m
12)     for 1≤j≤n
13)         D[i,j] = min {D[i-1,j  ] + d(Pi,ε ),       // delete
14)                       D[i  ,j-1] + d(ε ,Tj),       // insert
15)                       D[i-1,j-1] + d(Pi,Tj)}       // substitute
16)         if D[i,j] is D[i-1,j]+d(Pi,ε)
17)             E[i,j] = E[i,j] U {delete}
18)         if D[i,j] is D[i,j-1]+d(ε,Tj)
19)             E[i,j] = E[i,j] U {insert}
20)         if D[i,j] is D[i-1,j-1]+d(Pi,Tj)
21)             E[i,j] = E[i,j] U {substitute}
```

Figure 1. Algorithm A: Classic Algorithm for Approximate String Matching

2.2. Example

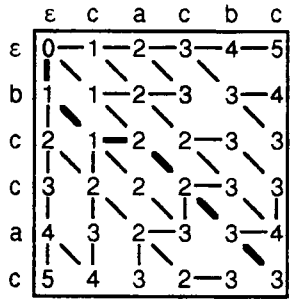
Figure 3(a) shows the distance matrix and links resulting from the computation of d(**bccac**,**cacbc**). Vertical links (|) indicate deletions, Horizontal links (—) insertions, and diagonal links (\) substitutions or matches. The thicker links trace the optimal alignment shown in figure 3(b)— "delete **b**, match **c**, insert **a**, match **c**, substitute **b** for **a**, and match **c**."

Any path of links between elements D[0,0] and D[m,n] corresponds to an optimal alignment. In this example the only other such path follows the diagonal and represents 3 substitutions and two matches. The corresponding alignment is shown in figure 3(c).

```
1)  E has been filled by the classic algorithm
2)  list is an ordered list of {delete, insert, substitute}
3)  ∘ is the concatenation operator on lists

4)  BackTrace (i,j,list):
5)      if E[i,j] is empty
6)          print list
7)      else
8)          if delete ∈ E[i,j]
9)              BackTrace (i-1,j,delete ∘ list)
10)         if insert ∈ E[i,j]
11)             BackTrace (i,j-1,insert ∘ list)
12)         if substitute ∈ E[i,j]
13)             BackTrace (i-1,j-1,substitution ∘ list)
```

Figure 2. Algorithm B: Back-Tracing Optimal Alignments



(a) Distance Matrix

P: **bc-cac**
T: **-cacbc**
d: 1 1 1

(b)

P: **bccac**
T: **cacbc**
d: 11 1

(c)

Figure 3. Example Distance Matrix for Algorithm A

```
1)  D is an array of non-negative real numbers over [0..m]
2)  diag, del, ins, and sub are non-negative real numbers
3)  D[0] = 0
4)  for 1≤i≤m
5)      D[i] = D[i-1] + d(P_i,ε)                          // delete
6)  for 1≤j≤n
7)      diag = D[0]
8)      D[0] = D[0] + d(ε,T_j)                            // insert
9)      for 1≤i≤m
10)         del = D[i-1] + d(P_i,ε )                      // delete
11)         ins = D[i]   + d(ε ,T_j)                      // insert
12)         sub = diag   + d(P_i,T_j))                    // substitute
13)         diag = D[i]
14)         D[i] = min {del, ins, sub}
```

Figure 4. Algorithm C: Approximate String Matching in Linear Space

2.3. Complexity

Algorithm A has O(mn) time complexity and O(mn) space complexity, since each of the (m+1)(n+1) matrix cells can be computed in linearly bounded time. Without multiple traces, algorithm B has O(m+n) complexity, as m+n is the longest possible path.

If alignments are not needed, Hirschberg [5] shows that we can eliminate the E matrix and reduce D to two columns (or two rows, whichever is smallest) resulting in linear space complexity O(min(m,n)). In fact, we can reduce D to a single row and a few temporary variables. The result is algorithm C, shown in figure 4.

3. Pumping of Single Symbols

3.1. Definition

We now extend the prototype to allow pumping of any of its symbols, indicated by marking the symbol with the Kleene closure indicator *. $\hat{P}$ is a string P over Σ with a labeling function that tells us whether a symbol P_i is allowed to pump. Each symbol P_i is labeled as *simple* or *pumpable*. Such a prototype represents a regular language, denoted $L(\hat{P})$.

We first extend the distance function to allow for languages of strings. The distance between two languages is the minimum cost of an alignment between a string from one language and a string from the other:

$$d(L_1,L_2) = \min \{ d(S_1,S_2) \mid S_1 \in L_1,\ S_2 \in L_2 \}$$

Then for a prototype $\hat{P}$ we treat T as a language containing only T and define

$$\begin{aligned} d(\hat{P},T) &= d(L(\hat{P}),L(T)) \\ &= \min \{ d(S,T) \mid S \in L(\hat{P}) \} \end{aligned} \qquad (3.1)$$

We also find it useful to define the series of languages representing the prefixes of $\hat{P}$. For any prefix $\hat{P}_{0:i}$ of $\hat{P}$, we define the language $L_i(\hat{P}) = L(\hat{P}_{0:i})$. This definition is consistent with a more general definition that will be introduced in chapter 4.

As an example, consider the prototype $\hat{P}$=**bc*ac**. The language is

$$L(\hat{P}) = \{\mathbf{bac}, \mathbf{bcac}, \mathbf{bccac}, \mathbf{bcccac}, \mathbf{bccccac}, \mathbf{bcccccac}, \ldots\}$$

and

$$d(\mathbf{bc^*ac}, \mathbf{bcacbac})$$

is the minimum of

$$\begin{aligned} d(\mathbf{bac},\mathbf{bcacbac}) &= 4, \\ d(\mathbf{bcac},\mathbf{bcacbac}) &= 3, \\ d(\mathbf{bccac},\mathbf{bcacbac}) &= 2, \\ d(\mathbf{bcccac},\mathbf{bcacbac}) &= 2, \\ d(\mathbf{bccccac},\mathbf{bcacbac}) &= 2, \\ d(\mathbf{bcccccac},\mathbf{bcacbac}) &= 3, \\ \cdots\ \cdots\ \cdots & \quad \cdots \end{aligned}$$

We will first present an algorithm to compute distances and alignments allowing pumping of single symbols, show an example, then prove the correctness of the recurrence relations at the heart of the algorithm.

3.2. Algorithm

The algorithm for single symbol pumping is the same as the classic algorithm, except that a new recurrence relation is used to compute D[i,j] when i represents a pumpable row (when P_i is a pumpable symbol).

To compute the distance between $\hat{P}$ and T, we iteratively match longer and longer prefixes of $\hat{P}$ and T. For simple rows we can use the same recurrence relations (2.1) and (2.2) as algorithm A:

$$d(\hat{P}_{0:i},T_{0:j}) = \min \begin{cases} \textcircled{D}\ d(\hat{P}_{0:i-1},T_{0:j}) + d(P_i,\varepsilon) \\ \textcircled{I}\ d(\hat{P}_{0:i},T_{0:j-1}) + d(\varepsilon,T_j) \\ \textcircled{S}\ d(\hat{P}_{0:i-1},T_{0:j-1}) + d(P_i,T_j) \end{cases} \qquad (3.2)$$

and

$$\begin{aligned} d(\hat{P}_{0:0},T_{0:0}) &= 0 \\ d(\hat{P}_{0:i},T_{0:0}) &= \textcircled{D}\ d(\hat{P}_{0:i-1},T_{0:0}) + d(P_i,\varepsilon) \\ d(\hat{P}_{0:0},T_{0:j}) &= \textcircled{I}\ d(\hat{P}_{0:0},T_{0:j-1}) + d(\varepsilon,T_j) \end{aligned} \qquad (3.3)$$

For pumpable rows we use a different recurrence relation. The new recurrence allows an *empty-match* into a pumping row, replacing the usual delete operation. Substitute, normally a diagonal move, is replaced by *pump-substitute*, which is horizontal. The insert operation remains. The pumpable recurrence relations are[1]

---

[1] The circled letters E and P stand for empty match and pump substitute, respectively.

$$d(\hat{P}_{0:i},T_{0:j}) = \min \begin{cases} \text{(E)} & d(\hat{P}_{0:i-1},T_{0:j}) \\ \text{(I)} & d(\hat{P}_{0:i},T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(P)} & d(\hat{P}_{0:i},T_{0:j-1}) + d(P_i,T_j) \end{cases} \quad (3.4)$$

and

$$d(\hat{P}_{0:i},T_{0:0}) = \text{(E)}\ d(\hat{P}_{0:i-1},T_{0:0}) \quad (3.5)$$

Algorithm D, shown in figure 5, computes the distance between $\hat{P}$ and T. For the sake of clarity, the algorithm as shown does not produce the E matrix of links that algorithm A did. This could easily be added. Back tracing would be performed in the same manner as in algorithm B, with appropriate consideration for the new operations.

The D[0..m,0..n] matrix is filled so that D[i,j] = $d(\hat{P}_{0:i},T_{0:j})$, the distance between the prefix language $L(\hat{P}_{0:i})$ and the prefix string $T_{0:j}$. Lines 2 through 4 initialize the top row as per formula (3.3). Lines 5 through 9 initialize the left column, using formula (3.3) or (3.5) as appropriate. Lines 10 through 19 fill the interior using formula (3.2) or (3.4). The desired distance $d(\hat{P},T)$ is found at D[m,n].

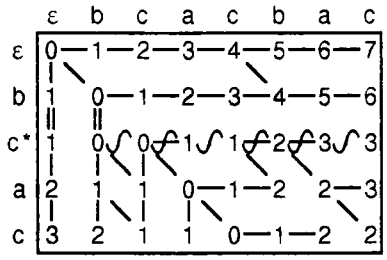
3.3. Example

The D matrix for the computation of d(**bc*ac**,**bcacbac**) is shown in figure 6(a). A ∫ link is used to indicate substitution in a pumping row, and a ‖ link indicates an empty-match. ∫ and — links are overlaid in situations where a substitution and an insert have equivalent cost.

Figures 6(b) through 6(e) show the optimal alignments, all of which have a cost of 2. All four are traced from essentially the same path through the D matrix. However, three

```
1)  D is a matrix of non-negative real numbers over [0..m,0..n]
2)  D[0,0] = 0
3)  for 1≤j≤n do
4)     |D[0,j] = D[0,j-1] + d(ε,Tj)                    // insert
5)  for 1≤i≤m do
6)     |if label(i) is simple                          // simple row
7)     |   |D[i,0] = D[i-1,0] + d(Pi,ε)                // delete
8)     |else                                           // pumpable row
9)     |   |D[i,0] = D[i-1,0]                          // empty-match
10) for 1≤i≤m do
11)    |for 1≤j≤n do
12)    |   |if label(i) is simple                      // simple row
13)    |   |   |D[i,j] = min{D[i-1,j]+d(Pi,ε),        // delete
14)    |   |   |           D[i,j-1]+d(ε,Tj),          // insert
15)    |   |   |           D[i-1,j-1]+d(Pi,Tj)}       // substitute
16)    |   |else                                      // pumpable row
17)    |   |   |D[i,j] = min{D[i-1,j],                // empty-match
18)    |   |   |           D[i,j-1]+d(ε,Tj),          // insert
19)    |   |   |           D[i,j-1]+d(Pi,Tj)}         // pump-substitute
```

Figure 5. Algorithm D: Approximate String Matching with Single Symbol Pumping



(a) Distance Matrix

(b)	(c)	(d)	(e)
P: **bccccac**	P: **bc-ccac**	P: **bccc-ac**	P: **bc-c-ac**
T: **bcacbac**	T: **bcacbac**	T: **bcacbac**	T: **bcacbac**
d: 1 1	d: 1 1	d: 1 1	d: 1 1

Figure 6. Distance Matrix with Pumpable Symbol

different prototype strings (**bccac**, **bcccac**, and **bccccac**) are represented. The algorithm not only determines the distance and alignment but also produces the corresponding prototype strings. The alignments in 6(d) and 6(e) each have two interpretations— the inserted **b** can be viewed as being part of **c***, or as being between **c*** and **a**.

In some applications it may be undesirable to allow insertions while pumping. This can easily be accommodated by eliminating line 18. The process still has a mathematical basis if the pumpable symbol is considered as a new, distinct, member of the alphabet, having an infinite insertion cost.

3.4. Proof

Algorithm C is correct if $D[m,n] = d(\hat{P},T)$. Just as the classic algorithm establishes the property $D[i,j] = d(P_{0:i},T_{0:j})$, we define a prefix-matching property for algorithm C. We say that D[i,j] is *correct* if $D[i,j] = d(L_i(\hat{P}),T_{0:j})$. We will show that all assignments to D in algorithm C establish and preserve the correctness property. If so, D[m,n] is correct,

$$D[m,n] = d(L_m(\hat{P}),T_{0:n}) = d(\hat{P},T)$$

hence algorithm D is correct.

Lemma 2: Every cell in D is correct. That is,

$\forall i,j,\ 0 \le i \le m,\ 0 \le j \le n,\ D[i,j] = d(L_i(\hat{P}), T_{0:j})$

Proof: By induction on the rows. We will first show that the first row is correct. Then the remaining rows can be split into two types, simple and pumpable. We show that either type of row will be correct if the row immediately preceding it is. Within each row, the proof is by induction on the columns.

basis: $i = 0$

Consider the first row. Line 2 initializes D[0,0] to 0, which is correct, since

$d(L_0(\hat{P}), T_{0:0}) = d(\varepsilon,\varepsilon) = 0$

Line 4 evaluates the remainder of row 0. Suppose j is a column for which D[0,j-1] is correct. Then

$$\begin{aligned} D[0,j] &= D[0,j-1] + d(\varepsilon, T_j) \\ &= d(L_0(\hat{P}), T_{0:j-1}) + d(\varepsilon, T_j) \\ &= d(\varepsilon, T_{0:j-1}) + d(\varepsilon, T_j) \\ &= d(\varepsilon, T_{0:j}) \\ &= d(L_0(\hat{P}), T_{0:j}) \end{aligned}$$

So every cell in row 0 is correct.

case 1: $i > 0$, row i-1 is correct, and row i is a simple row

Assume i is a simple row (P_i is labeled as simple), and that the previous row is correct. Then line 7 assigns

$$\begin{aligned} D[i,0] &= D[i-1,0] + d(P_i,\varepsilon) \\ &= d(L_{i-1}(\hat{P}), T_{0:0}) + d(P_i,\varepsilon) && \text{(row i-1 correct)} \\ &= \min \{\, d(S,T_{0:0}) \mid S \in L_{i-1}(\hat{P}) \,\} + d(P_i,\varepsilon) && \text{(definition of } L_{i-1}) \\ &= \min \{\, d(S^{\bullet},T_{0:0}) \mid S \in L_i(\hat{P}) \,\} + d(P_i,\varepsilon) && \text{(relation of } L_{i-1} \text{ to } L_i) \\ &= \min \{\, d(S^{\bullet},T_{0:0}) + d(P_i,\varepsilon) \mid S \in L_i(\hat{P}) \,\} && (d(P_i,\varepsilon) \text{ is constant}) \\ &= d(L_i(\hat{P}), T_{0:0}) && \text{(definition of } L_i) \end{aligned}$$

So the leftmost cell in row i is correct.

Further suppose that j is a column for which D[i,j-1] is correct. Then by lines 13 through 15,

$$D[i,j] = \min \begin{cases} \text{Ⓓ } D[i-1,j] + d(P_i,\varepsilon) \\ \text{Ⓘ } D[i,j-1] + d(\varepsilon,T_j) \\ \text{Ⓢ } D[i-1,j-1] + d(P_i,T_j) \end{cases}$$

Each of these three matrix elements has already been computed and is correct, so,

$$D[i,j] = \min \begin{cases} \text{Ⓓ } d(L_{i-1}(\hat{P}),T_{0:j}) + d(P_i,\varepsilon) \\ \text{Ⓘ } d(L_i(\hat{P}),T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{Ⓢ } d(L_{i-1}(\hat{P}),T_{0:j-1}) + d(P_i,T_j) \end{cases}$$

Each of the three distances is the minimum over some set. Expanding gives us

$$D[i,j] = \min \begin{cases} \text{Ⓓ } d(S,T_{0:j}) + d(P_i,\varepsilon) \mid S \in L_{i-1}(\hat{P}) \\ \text{Ⓘ } d(S,T_{0:j-1}) + d(\varepsilon,T_j) \mid S \in L_i(\hat{P}) \\ \text{Ⓢ } d(S,T_{0:j-1}) + d(P_i,T_j) \mid S \in L_{i-1}(\hat{P}) \end{cases}$$

These can be rewritten in terms of $L_i(\hat{P})$.

$$D[i,j] = \min \begin{cases} \text{Ⓓ } d(S^{\bullet},T_{0:j}) + d(P_i,\varepsilon) \mid S \in L_i(\hat{P}) \\ \text{Ⓘ } d(S,T_{0:j-1}) + d(\varepsilon,T_j) \mid S \in L_i(\hat{P}) \\ \text{Ⓢ } d(S^{\bullet},T_{0:j-1}) + d(P_i,T_j) \mid S \in L_i(\hat{P}) \end{cases}$$

Combining the three cases for each S, and applying the recurrence from (2.1), we can reduce to

$$\begin{aligned} D[i,j] &= \min \{ d(S,T_{0:j}) \mid S \in L_i(\hat{P}) \} \\ &= d(L_i(\hat{P}),T_{0:j}) \end{aligned}$$

So lines 13 through 15 compute correct values of D[i,j], and every cell in the simple row i is correct.

<u>case 2</u>: i>0, row i-1 is correct, and row i is a pumpable row

First, recall that when P_i is pumpable,

$$\begin{aligned} L_i(\hat{P}) &= L_{i-1}(\hat{P})\, P_i^* \\ &= \{ SP_i^k \mid S \in L_{i-1}(\hat{P}),\ k \geq 0 \} \\ &= L_{i-1}(\hat{P}) \cup L_{i-1}(\hat{P})\, P_i^+ \end{aligned}$$

Line 9 assigns

$$\begin{aligned} D[i,0] &= D[i-1,0] \\ &= d(L_{i-1}(\hat{P}), T_{0:0}) && \text{(row i-1 correct)} \\ &= \min \{ d(S,T_{0:0}) \mid S \in L_{i-1}(\hat{P}) \} && \text{(definition of } L_{i-1}) \end{aligned}$$

We then note that regardless of P_i, the only possible operations are series of deletes, since $T_{0:0}$ is empty. So we have, for any $k \geq 0$,

$$d(SP_i^k, T_{0:0}) = d(S,T_{0:0}) + k\,d(P_i,\varepsilon).$$

Since $d(P_i,\varepsilon) \geq 0$, $d(SP_i^k,T_{0:0}) \geq d(S,T_{0:0})$. So we can add members of the form $d(SP_i^k,T_{0:0})$ to the set without changing the minimum.

$$\begin{aligned} D[i,0] &= \min \{ d(SP_i^k,T_{0:0}) \mid S \in L_{i-1}(\hat{P}),\ k \geq 0 \} \\ &= \min \{ d(S,T_{0:0}) \mid S \in L_{i-1}(\hat{P})P_i^k,\ k \geq 0 \} && \text{(changing S)} \\ &= \min \{ d(S,T_{0:0}) \mid S \in L_i(\hat{P}) \} && \text{(relation of } L_{i-1} \text{ to } L_i) \\ &= d(L_i(\hat{P}),T_{0:0}) && \text{(definition of } L_i) \end{aligned}$$

Further suppose that j is a column for which D[i,j-1] is correct. Then by lines 17 through 19,

$$D[i,j] = \min \begin{cases} \text{(E)}\ D[i-1,j] \\ \text{(I)}\ D[i,j-1] + d(\varepsilon,T_j) \\ \text{(P)}\ D[i,j-1] + d(P_i,T_j) \end{cases}$$

$$= \min \begin{cases} \text{(E)}\ d(L_{i-1}(\hat{P}),T_{0:j}) \\ \text{(I)}\ d(L_i(\hat{P}),T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(P)}\ d(L_i(\hat{P}),T_{0:j-1}) + d(P_i,T_j) \end{cases}$$

$$= \min \begin{cases} \text{(E)}\ d(S,T_{0:j}) & \mid S \in L_{i-1}(\hat{P}) \\ \text{(I)}\ d(S,T_{0:j-1}) + d(\varepsilon,T_j) & \mid S \in L_i(\hat{P}) \\ \text{(P)}\ d(S,T_{0:j-1}) + d(P_i,T_j) & \mid S \in L_i(\hat{P}) \end{cases}$$

As before, any new member as large as D[i,j] can be added to the set without affecting the minimum. We can show that any value in the set $\{d(S,T_{0:j}) + d(P_i,\varepsilon) \mid S \in L_i(\hat{P})\}$ is at least as large as D[i,j], so we can add this entire set. We use the fact that $L_i(\hat{P}) = \{SP_i^k \mid S \in L_{i-1}(\hat{P}), k \geq 0\}$ and prove by induction on k.

The basis step is that, for any $S \in L_{i-1}(\hat{P})$, $d(S,T_{0:j})$ is one of the (E) members in the set, so

$$d(SP_i^0,T_{0:j}) + d(P_i,\varepsilon) \geq d(SP_i^0,T_{0:j}) = d(S,T_{0:j}) \geq D[i,j]$$

The inductive step assumes that $k>0$ and $d(SP_i^{k-1},T_{0:j}) + d(P_i,\varepsilon) \geq D[i,j]$. By recurrence (2.1),

$$d(SP_i^k,T_{0:j}) + d(P_i,\varepsilon) \geq \min \begin{cases} \text{(D)} & d(SP_i^{k-1},T_{0:j}) + d(P_i,\varepsilon) \\ \text{(I)} & d(SP_i^k,T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(S)} & d(SP_i^{k-1},T_{0:j-1}) + d(P_i,T_j) \end{cases}$$

The (D) item is known to be as large as D[i,j] (by our inductive hypothesis). The (I) and (S) items are also:

$$\begin{aligned} d(SP_i^k,T_{0:j-1}) + d(\varepsilon,T_j) &\geq \min \{ d(S,T_{0:j-1}) + d(\varepsilon,T_j) \mid S \in L_i(\hat{P}) \} \\ &\geq D[i,j] \\ d(SP_i^{k-1},T_{0:j-1}) + d(P_i,T_j) &\geq \min \{ d(S,T_{0:j-1}) + d(P_i,T_j) \mid S \in L_i(\hat{P}) \} \\ &\geq D[i,j] \end{aligned}$$

So

$$d(SP_i^k,T_{0:j}) + d(P_i,\varepsilon) \geq D[i,j]$$

So we can include the set $\{d(S,T_{0:j}) + d(P_i,\varepsilon) \mid S \in L_i(\hat{P})\}$ without altering the minimum (we mark it with (D) since it is of the same form as a delete):

$$D[i,j] = \min \begin{cases} \text{(E)}\ d(S,T_{0:j}) & \mid S \in L_{i-1}(\hat{P}) \\ \text{(D)}\ d(S,T_{0:j}) + d(P_i,\varepsilon) & \mid S \in L_i(\hat{P}) \\ \text{(I)}\ d(S,T_{0:j-1}) + d(\varepsilon,T_j) & \mid S \in L_i(\hat{P}) \\ \text{(P)}\ d(S,T_{0:j-1}) + d(P_i,T_j) & \mid S \in L_i(\hat{P}) \end{cases}$$

$$= \min \begin{cases} \text{(E)}\ d(S,T_{0:j}) & \mid S \in L_{i-1}(\hat{P}) \\ \text{(D)}\ d(S,T_{0:j}) + d(P_i,\varepsilon) & \mid S \in L_{i-1}(\hat{P})P_i^* \\ \text{(I)}\ d(S,T_{0:j-1}) + d(\varepsilon,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^* \\ \text{(P)}\ d(S,T_{0:j-1}) + d(P_i,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^* \end{cases}$$

We can also remove a member from the set if we can show another member that is no larger. Note that for any $S \in L_{i-1}(\hat{P})$, $d(S,T_{0:j-1}) + d(\varepsilon,T_j)$ is one of the (I) members in the set. But $d(S,T_{0:j})$ is no larger, and is one of the (E) members. So removing $d(S,T_{0:j-1}) + d(\varepsilon,T_j)$ will not change the minimum. Doing so, we get

$$D[i,j] = \min \begin{cases} \text{(E)}\ d(S,T_{0:j}) & \mid S \in L_{i-1}(\hat{P}) \\ \text{(D)}\ d(S,T_{0:j}) + d(P_i,\varepsilon) & \mid S \in L_{i-1}(\hat{P})P_i^* \\ \text{(I)}\ d(S,T_{0:j-1}) + d(\varepsilon,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^+ \\ \text{(P)}\ d(S,T_{0:j-1}) + d(P_i,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^* \end{cases}$$

The (D) and (P) members can be rewritten by using

$$\{ S \mid S \in L_{i-1}(\hat{P})P_i^* \} = \{ S^\bullet \mid S \in L_{i-1}(\hat{P})P_i^+ \}$$

This gives us

$$D[i,j] = \min \begin{cases} \text{(E)}\ d(S,T_{0:j}) & \mid S \in L_{i-1}(\hat{P}) \\ \text{(D)}\ d(S^\bullet,T_{0:j}) + d(P_i,\varepsilon) & \mid S \in L_{i-1}(\hat{P})P_i^+ \\ \text{(I)}\ d(S,T_{0:j-1}) + d(\varepsilon,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^+ \\ \text{(P)}\ d(S^\bullet,T_{0:j-1}) + d(P_i,T_j) & \mid S \in L_{i-1}(\hat{P})P_i^+ \end{cases}$$

The (D), (I), and (P) members can be combined and reduced by recurrence (2.1) to yield

$$D[i,j] = \min \begin{cases} d(S,T_{0:j}) \mid S \in L_{i-1}(\hat{P}) \\ d(S,T_{0:j}) \mid S \in L_{i-1}(\hat{P})P_i^+ \end{cases}$$

Finally, since $L_i(\hat{P}) = L_{i-1}(\hat{P}) \cup L_{i-1}(\hat{P})\ P_i^+$, we can combine the two remaining items to see that D[i,j] is correct:

$$\begin{aligned} D[i,j] &= \min \{ d(S,T_{0:j}) \mid S \in L_i(\hat{P}) \} \\ &= d(L_i(\hat{P}),T_{0:j}) \end{aligned}$$

So every cell in the pumpable row i is correct.

We have shown that every cell in row 0 is correct, and that, regardless of the type of row (simple or pumpable), if the previous row is correct, then every cell in the current row is correct. By induction on rows, then, every cell in D is correct. ▣

3.5. Complexity

Algorithm D has the same time and space complexity as algorithm A— O(mn). m is the length of $\hat{P}$, the number of symbols from Σ in the regular expression. As before, each of the (m+1)(n+1) matrix cells can be computed in linearly bounded time. An algorithm like algorithm C can be constructed, using O(min(m,n)) space, if alignments and corresponding prototype strings are not needed.

4. Pumping of Substrings

4.1. Definition

We now turn our attention to the problem of allowing pumping of substrings. This is more complicated than single symbol pumping because such a substring introduces a cycle of arbitrary length. The convenient computational order of algorithms A and D is disturbed—the recurrence relation for $d(\hat{P}_{0:i}, T_{0:j})$ may depend upon symbols in the suffix $P_{i+1:m}$. Further, these cycles can nest and overlap in ways that further contort the computational dependencies.

It is no longer convenient to represent the arbitrary prototype by a simple text string, so we turn to a network representation. The networks used are non-deterministic finite automata, with ε-transitions. For convenience, the transition labels are shown in the nodes rather than on the arcs. Figure 7 shows several prototypes in this representation, with some sample strings from the corresponding languages.

The languages represented by such networks are the same as the languages represented by regular expressions[1], so we can limit our discussion to prototypes constructed from a restricted[2] regular expression. Doing so lets us avoid overlapping cycles[3], and we are able to produce an algorithm that is still O(mn).

---

1 This is discussed in more detail in the appendix.

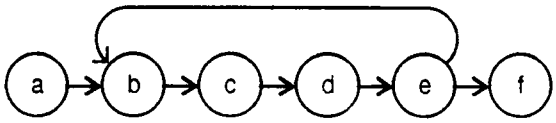
2 We restrict regular expressions to those using only concatenation, union, and Kleene closure, which is sufficient to describe any regular language.

3 For a cycle C, let min-index(C) be the lowest index of any node in C, and max-index(C) the highest. Two cycles C and D are said to overlap if and only if

$$\text{min-index}(C) < \text{min-index}(D) \leq \text{max-index}(C) < \text{max-index}(D)$$

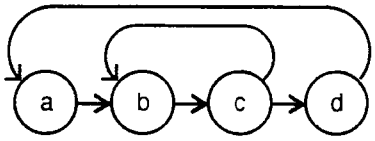
or

$$\text{min-index}(D) < \text{min-index}(C) \leq \text{max-index}(D) < \text{max-index}(C).$$



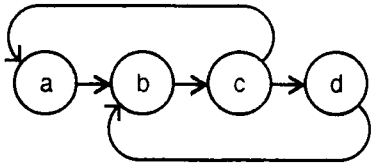
samples
abcdef
abcdebcdef
abcdebcdebcdef

(a) Single Cycle



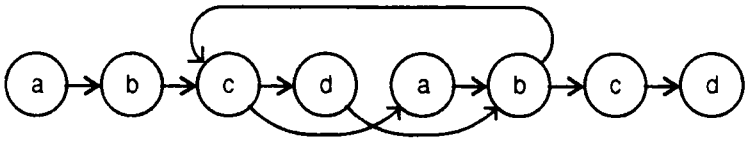
samples
abcd
abcbcd
abcdabcbcd

(b) Nested Cycles



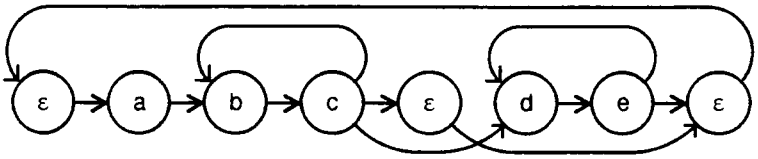
samples
abcd
abcabcd
abcdbcd
abcdbcabcdbcd

(c) Overlapping Cycles



samples
abcd
abcabcd
abcdbcd
abcdbcabcdbcd

(d) Cycle Containing Alternatives



samples
abc
abcbc
abcdede
abcdeabcabcdede

(e) Nested Cycles with ε-Nodes

Figure 7. Arbitrary Prototypes

4.1.1. Prototypes from Regular Expressions

We extend the prototype to be a network constructed from a regular expression (by a process we will describe shortly). In order to make use of a non-empty regular expression R (over alphabet Σ), we first determine any single symbols that may be pumped and label them as pumpable. The remaining symbols are labeled as either simple or empty (when the symbol is ε). The operators in R are unlabeled.

We then convert a regular expression to a prototype network $\hat{P}$ that recognizes the same language (given a regular expression R, we call the prototype built by this process $\hat{P}(R)$). We represent $\hat{P}$ by the pair (P,pred), where

- P is string of symbols over $\Sigma \cup \{\varepsilon\}$ labeled as empty, simple, or pumpable.
- pred() is a function mapping each position in P to a set of positions, the *predecessors* of the position.

If $\hat{P}$ has properties (4.1) through (4.3) we define it to be *proper*.

- pred(1) is empty, and $|pred(i)| \leq 2$ for $i>1$. (4.1)
- Except for i=1, pred(i) contains some $k<i$.[1] (4.2)
- $\hat{P}$ includes no overlapping cycles. (4.3)

We will show that our construction always produces proper networks. Figure 8 shows an example of a regular expression, a corresponding (proper) network, and the formal definition of the prototype.

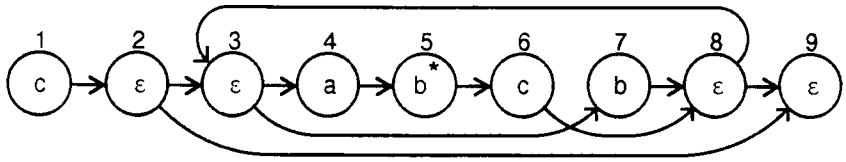
We now present a technique for constructing a prototype from a regular expression. The construction is based on a proof in [6] (pages 29-32), and similar constructions are used in [10] and [13]. The unique aspect of this presentation is the numbering of nodes during the construction.

---

[1] The predecessor set pred(i) usually contains i-1, but not always, as exhibited by node 7 of the network in figure 8.

c(ab*c+b)*

(a) Regular Expression



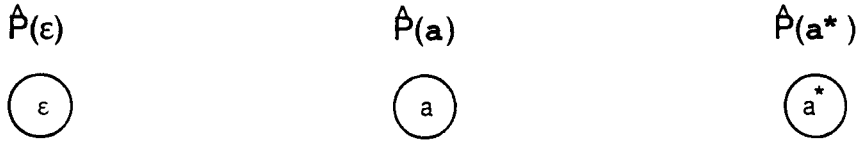
(b) Prototype Network

i	P_i	label(i)	pred(i)
1	c	simple	{}
2	ε	empty	{1}
3	ε	empty	{2,8}
4	a	simple	{3}
5	b	pumpable	{4}
6	c	simple	{5}
7	b	simple	{3}
8	ε	empty	{6,7}
9	ε	empty	{2,8}

(c) Prototype Definition

Figure 8. Regular Expression and Corresponding Prototype

Figures 9 through 12 diagram the construction process. Primitive regular expressions produce a single node prototype. Then we combine prototypes by concatenation, union, or closure to build successively larger ones. For example, if R is **c(ab*c+b)***, we first build prototypes for the primitives **a**, **b**, **b***, and **c**. We can combine these by concatenation to get prototypes for first **ab*** then **ab*c**. We combine this by union with the prototype for **b** to get a prototype for **ab*c+b**. We use the closure rules to get a prototype for **(ab*c+b)***, then concatenate with **c** to get a prototype for **c(ab*c+b)***.



$P_1 = R_1$, labeled empty
pred(1) = {}

(a) Empty Symbol

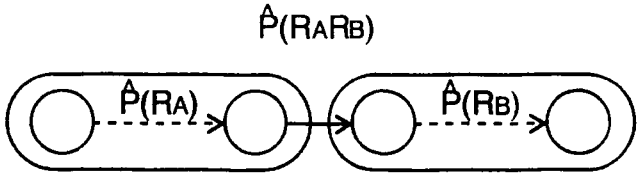
$P_1 = R_1$, labeled simple
pred(1) = {}

(b) Simple Symbol

$P_1 = R_1$, labeled pumpable
pred(1) = {}

(c) Pumpable Symbol

Figure 9. Prototype Constructions for Primitive Regular Expressions

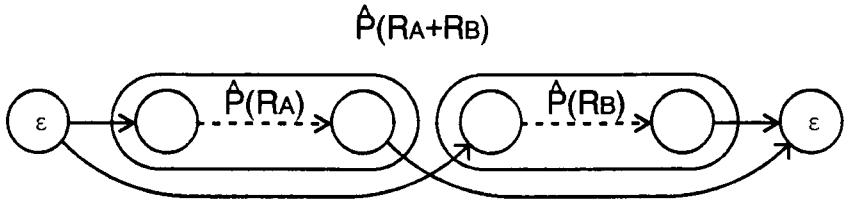


for $1 \le i \le |P_A|$,
$P_i = (P_A)_i$
$pred(i) = pred_A(i)$

for $i = |P_A|+1$,
$P_i = (P_B)_{i-|P_A|}$
$pred(i) = \{|P_A|\}$

for $|P_A|+2 \le i \le |P_A|+|P_B|$,
$P_i = (P_B)_{i-|P_A|}$
$pred(i) = \{ j+|P_A| \mid j \in pred_B(i-|P_A|) \}$

Figure 10. Prototype Construction for Concatenation of Regular Expressions



for i = 1,
P_i = ε, labeled empty
pred(i) = {}

for i = 2,
P_i = $(P_A)_{i-1}$
pred(i) = {1}

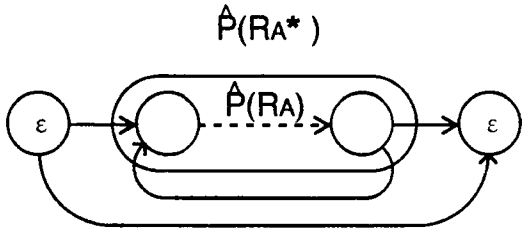
for $3 \le i \le |P_A|+1$,
P_i = $(P_A)_{i-1}$
pred(i) = $\{ j+1 \mid j \in pred_A(i-1) \}$

for $i = |P_A|+2$,
P_i = $(P_B)_{i-|P_A|-1}$
pred(i) = {1}

for $|P_A|+3 \le i \le |P_A|+|P_B| + 1$,
P_i = $(P_B)_{i-|P_A|-1}$
pred(i) = $\{ j+|P_A|+1 \mid j \in pred_B(i-|P_A|-1) \}$

for $i = |P_A|+|P_B|+2$,
P_i = ε, labeled empty
pred(i) = $\{|P_A|+1, |P_A|+|P_B|+1\}$

Figure 11. Prototype Construction for Union of Regular Expressions



for i = 1,
P_i = ε, labeled empty
pred(i) = {}

for i = 2,
P_i = $(P_A)_{i-1}$
pred(i) = {1, $|P_A|+1$}

for $3 \le i \le |P_A|+1$,
P_i = $(P_A)_{i-1}$
pred(i) = { j+1 | j ∈ $pred_A(i-1)$ }

for $i = |P_A|+2$,
P_i = ε, labeled empty
pred(i) = {1, $|P_A|+1$}

Figure 12. Prototype Construction for Kleene Closure of a Regular Expression

4.1.2. Properties of Prototypes

We list several properties of prototypes constructed in this manner, and offer proofs, sketches of proofs, or references for each.

Lemma 3: $\hat{P}(R)$, with the first node considered as the initial state and the last node as the only accepting state, recognizes the same language as R.

Proof: A proof can be found in [6]. ◻

Lemma 4: $\hat{P}(R)$ is proper.

Proof: By induction on the number of nodes in the network. The properties we must prove are (4.1) through (4.3).

basis: $\hat{P}(R)$ has one node.

These are the primitive prototypes, so one of the construction rules in figure 9(a), (b), or (c) is used. Since there is only one node, (4.2) and (4.3) are trivially true. (4.1) is true because pred(1) is empty. Therefore $\hat{P}(R)$ is proper.

case 1: $\hat{P}(R)$ has more than one node, $R=R_AR_B$, and all networks with fewer nodes built by this process are proper.

The construction follows figure 10. $\hat{P}(R_A)$, represented by P_A and $pred_A$, is combined with $\hat{P}(R_B)$, represented by P_B and $pred_B$.

pred(1) is copied directly from the first node of $\hat{P}(R_A)$, and since R_A is shorter (and is therefore proper), this is empty. The first node copied from $\hat{P}(R_B)$ will have only the single predecessor we add. All other nodes will have the same number as they have in $\hat{P}(R_A)$ or $\hat{P}(R_B)$, which is two or less. So (4.1) is true.

The first node copied from $\hat{P}(R_B)$ (node i) will have the predecessor i-1. The act of copying nodes preserves the predecessor ordering, so all other nodes (except node 1) will continue to have at least one lower indexed predecessor. So (4.2) is true.

No overlapping cycles are introduced, so (4.3) is true.

Therefore $\hat{P}(R)$ is proper.

case 2: $\hat{P}(R)$ has more than one node, $R=R_A+R_B$, and all networks with fewer nodes built by this process are proper.

The construction follows figure 11. This case is similar to case 1. The reader can verify that (4.1) through (4.3) are preserved (noting that the final node added will have two predecessors), and that $\hat{P}(R)$ is proper.

case 3: $\hat{P}(R)$ has more than one node, $R=R_A^*$, and all networks with fewer nodes built by this process are proper.

The construction follows figure 12. This case is also similar to case 1. The reader can verify that (4.1) through (4.3) are preserved (noting that the first node copied from $\hat{P}(R_A)$ has two new predecessors, but none before), and that $\hat{P}(R)$ is proper.

case 4: $\hat{P}(R)$ has more than one node, $R=(R_A)$, and all networks with fewer nodes built by this process are proper.

There is no construction for parenthetical grouping. $\hat{P}(R)$ is simply $\hat{P}(R_A)$. $\hat{P}(R_A)$ is not a shorter network, but, after we strip off all enclosing parenthesis from R_A, we will have a regular expression R_B that must be one of the forms covered by cases 1, 2, or 3, or the basis. Then $\hat{P}(R) = \hat{P}(R_B)$, which is proper.

We have shown that every network with one node is proper, and that the construction rules preserve this property if all shorter networks exhibit it. Therefore all networks constructed by this process are proper. ☒

Lemma 5: The number of nodes in $\hat{P}(R)$ (the length of P) is no more than twice the length of R (counting both operators and labeled symbols in R).

Proof: By induction on the length of the regular expression.

basis: R is of length 1.

R is a primitive regular expression, and we use one of the construction rules in figure 9. There is only one node, which is less than twice the length of R.

case 1: $R=R_AR_B$, and all networks built by this process from shorter regular expressions have fewer nodes than twice the length of the regular expression.

The construction follows figure 10. $|R|=|R_A|+|R_B|$, and $|P| = |P_A|+|P_B|$. So

$$|P| = |P_A|+|P_B| \leq 2|R_A|+2|R_B| = 2(|R_A|+|R_B|) = 2|R|$$

case 2: $R=R_A+R_B$, and all networks built by this process from shorter regular expressions have fewer nodes than twice the length of the regular expression.

The construction follows figure 11. $|R|=|R_A|+|R_B|+1$, and $|P| = |P_A|+|P_B|+2$. So

$$|P| = |P_A|+|P_B|+2 \leq 2|R_A|+2|R_B|+2 = 2(|R_A|+|R_B|+1) = 2|R|$$

case 3: $R=R_A^*$, and all networks built by this process from shorter regular expressions have fewer nodes than twice the length of the regular expression.

The construction follows figure 12. $|R|=|R_A|+1$, and $|P| = |P_A|+2$. So

$$|P| = |P_A|+2 \leq 2|R_A|+2 = 2(|R|-1)+2 = 2|R|$$

case 4: $R=(R_A)$, and all networks built by this process from shorter regular expressions have fewer nodes than twice the length of the regular expression.

$\hat{P}(R)$ is simply $\hat{P}(R_A)$. So $|R|=|R_A|+2$, and

$$|P| = |P_A| \leq 2|R_A| = 2(|R|-2) \leq 2|R|$$

By induction, then, every network constructed by this process have fewer nodes than twice the length of the regular expression. ▣

4.2. Algorithm

The algorithm for substring pumping uses the same general recurrence relations as algorithms A and D. A new recurrence relation is used for D[i,j] when row i is labeled empty. Further, where the previous algorithms assumed the previous row as a predecessor, we must now generalize the recurrence relation to account for the predecessor sets. More importantly, we have to account for predecessors whose values cannot be known at the time a cell's value is computed.

As in algorithms A and D, we iteratively match longer and longer prefixes of $\hat{P}$ and T, but now we must scan the cells column by column. Consider the computation of D[i,j] when one of its predecessors is in the suffix $\hat{P}_{i+1:m}$—when pred(i) contains some k>i. As we scan down a column (as in Algorithm D), we will not be able to completely determine the value at D[i,j], since it depends upon D[k,j], which is not yet known.

The solution is to ignore such predecessors[1] in a first pass through the column j, producing an upper bound on each D[i,j]. Then a second pass adjusts each D[i,j] based on the first pass results from all cells in the column. If row i has a suffix predecessor k, the first pass bound for D[k,j] can be used to reduce the bound on D[i,j]. While it may appear that additional passes would be necessary to determine a solution, we will prove in section 4.4 that the second pass is sufficient when the prototype network is proper.

The concept of a prefix language of $\hat{P}$ also changes. $\hat{P}$ is a network, so a prefix $\hat{P}_{0:i}$ of $\hat{P}$ is likely to contain arcs from nodes that are not in the prefix. If so, $\hat{P}_{0:i}$ is not a valid network. So we define the prefix language $L_i(\hat{P})$ to be the language recognized by $\hat{P}$, modified so that state i is the only accepting state.

We now have separate recurrence relations for first and second passes, as well as for empty, simple, and pumpable rows[2]. To distinguish the first and second pass results, we use d_1 and d_2, respectively. To handle the recurrence boundary conditions, we augment the network by adding an initial state $\hat{P}_0$, labeled empty and with no predecessors, and set pred(1) equal to {0}.

The top row uses the same recurrence relations as algorithms A and D for the first pass, and the second pass values are the same as the first:

$$\begin{aligned} d_1(L_0(\hat{P}),T_{0:0}) &= 0 \\ d_1(L_0(\hat{P}),T_{0:j}) &= \textcircled{I}\ d_1(L_0(\hat{P}),T_{0:j-1}) + d(\varepsilon,T_j) \\ d_2(L_0(\hat{P}),T_{0:0}) &= d_1(L_0(\hat{P}),T_{0:0}) \\ d_2(L_0(\hat{P}),T_{0:j}) &= d_1(L_0(\hat{P}),T_{0:j}) \end{aligned} \qquad (4.4)$$

---

1 Note that (when k>i) we only ignore D[k,j], not D[k,j-1]. The latter is in the previous column, so its value is known.

2 Even though the computation is performed column by column, we describe the recurrence relations primarily row by row.

All values in the left column remain the same in the second pass:

$$d_2(L_i(\hat{P}),T_{0:0}) \quad = \quad d_1(L_i(\hat{P}),T_{0:0}) \tag{4.5}$$

For rows labeled empty, the new recurrence permits empty-matches into the row. This amounts to taking the minimum value of all predecessors in the same column. So we have

$$\begin{aligned}
d_1(L_i(\hat{P}),T_{0:0}) &= \min \{ \text{(E)}\ d_1(L_k(\hat{P}),T_{0:0}) \mid k<i \in \text{pred}(i) \} \\
d_1(L_i(\hat{P}),T_{0:j}) &= \min \{ \text{(E)}\ d_1(L_k(\hat{P}),T_{0:j}) \mid k<i \in \text{pred}(i) \} \\
d_2(L_i(\hat{P}),T_{0:j}) &= \min \begin{cases} d_1(L_i(\hat{P}),T_{0:j}) \\ \text{(E)}\ d_2(L_k(\hat{P}),T_{0:j}) \mid k \in \text{pred}(i) \end{cases}
\end{aligned} \tag{4.6}$$

For simple rows, we have

$$\begin{aligned}
d_1(L_i(\hat{P}),T_{0:0}) &= \min \{ \text{(D)}\ d_1(L_k(\hat{P}),T_{0:0}) + d(P_i,\varepsilon) \mid k<i \in \text{pred}(i) \} \\
d_1(L_i(\hat{P}),T_{0:j}) &= \min \begin{cases} \text{(D)}\ d_1(L_k(\hat{P}),T_{0:j}) + d(P_i,\varepsilon) \mid k<i \in \text{pred}(i) \\ \text{(I)}\ d_2(L_i(\hat{P}),T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(S)}\ d_2(L_k(\hat{P}),T_{0:j-1}) + d(P_i,T_j) \mid k \in \text{pred}(i) \end{cases} \\
d_2(L_i(\hat{P}),T_{0:j}) &= \min \begin{cases} d_1(L_i(\hat{P}),T_{0:j}) \\ \text{(D)}\ d_2(L_k(\hat{P}),T_{0:j}) + d(P_i,\varepsilon) \mid k<i \in \text{pred}(i) \\ \text{(D)}\ d_1(L_k(\hat{P}),T_{0:j}) + d(P_i,\varepsilon) \mid k>i \in \text{pred}(i) \end{cases}
\end{aligned} \tag{4.7}$$

Finally, for pumpable rows we have

$$\begin{aligned}
d_1(L_i(\hat{P}),T_{0:0}) &= \min \{ \text{(E)}\ d_1(L_k(\hat{P}),T_{0:0}) \mid k<i \in \text{pred}(i) \} \\
d_1(L_i(\hat{P}),T_{0:j}) &= \min \begin{cases} \text{(E)}\ d_1(L_k(\hat{P}),T_{0:j}) \mid k<i \in \text{pred}(i) \\ \text{(I)}\ d_2(L_i(\hat{P}),T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(S)}\ d_2(L_i(\hat{P}),T_{0:j-1}) + d(P_i,T_j) \end{cases} \\
d_2(L_i(\hat{P}),T_{0:j}) &= \min \begin{cases} d_1(L_i(\hat{P}),T_{0:j}) \\ \text{(E)}\ d_2(L_k(\hat{P}),T_{0:j}) \mid k<i \in \text{pred}(i) \\ \text{(E)}\ d_1(L_k(\hat{P}),T_{0:j}) \mid k>i \in \text{pred}(i) \end{cases}
\end{aligned} \tag{4.8}$$

Figure 13 shows algorithm E, which computes the distance between $\hat{P}$ and T, allowing substring pumping. $\hat{P}$ is a proper prototype, represented by the string P and the predecessor sets. The D matrix holds the distances between any prefix language $L_i(\hat{P})$ and any prefix $T_{0:j}$. As in algorithms A and D, $d(\hat{P},T)$ is found at D[m,n].

The V array holds the cost of a vertical move into a row. This is either the cost of a delete or of an empty match, depending upon the type of row. Each value of V is determined during computation of column zero and is not changed thereafter. The value H holds the cost of a standard horizontal move into the current row—the cost of an insert. H is changed with each column.

Line 8 initializes the top left entry of the D matrix as in algorithms A and D. Lines 9 through 15 fill the V array and initialize the left column of D. Lines 16 through 20 fill the D matrix column by column. First H is determined for the current column, the top value in the column is computed, then procedures FirstPass and SecondPass are invoked to fill the rest of column. Each procedure visits every cell in the column.

The procedure FirstPass (lines 21 through 35) sets an upper bound on each cell in column j of D, considering the previous column and any previous cells within the column. np, nd, and nl are the minimum values of neighbors preceding, diagonally preceding, and to the left, respectively. Diagonally preceding neighbors are comprised of the cells in the previous column corresponding to predecessors of the cell.

The procedure SecondPass (lines 36 through 39) takes a second pass through the column, lowering any D values that have a better match involving a backward arc within the column. It will be shown that this second pass results in $D[i,j] = d(L_i(\hat{P}),T_{0:j})$.

```
1)  pred is an array of sets of integers over [1..m] satisfying...
2)      pred[i] contains exactly 1 or 2 members
3)      all k∈pred[i] are such that 0≤k≤m
4)      pred[i] contains at least one k such that k<i
5)  D is a matrix of non-negative real numbers over [0..m,0..n]
6)  V is an array of non-negative real numbers over [1..m]
7)  H is a non-negative real number
8)  D[0,0] = 0
9)  for 1≤i≤m do
10)     if label(i) is simple                          // simple row
11)        V[i] = d(P_i,ε)                             // delete
12)     else                                           // empty or pumping row
13)        V[i] = 0                                    // empty-match
14)     np = min {D[k,0] | k∈pred(i) and k<i}
15)     D[i,0] = np + V[i]                             // delete or empty-match
16) for 1≤j≤n do
17)     H = d(ε,T_j)                                   // insert
18)     D[0,j] = D[0,j-1] + H
19)     FirstPass(j,H)
20)     SecondPass(j)
```

Figure 13. Algorithm E: Approximate String Matching with Substring Pumping

(continues)

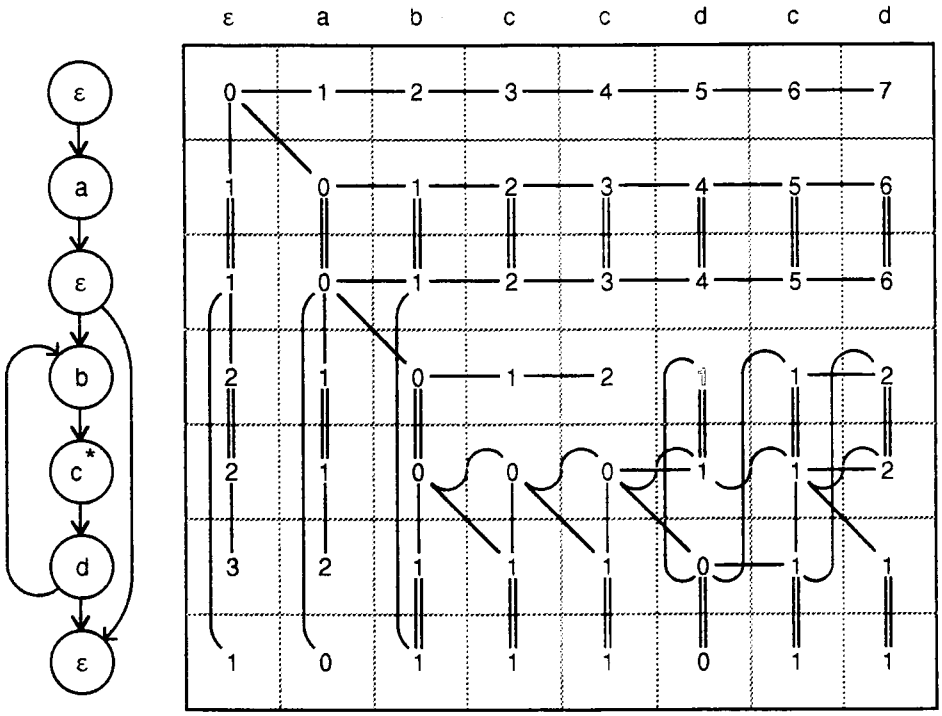
```
21) to perform FirstPass (j, H):
22)     for 1≤i≤m do
23)         np = min {D[k,j] | k∈pred(i) and k<i}
24)         nd = min {D[k,j-1] | k∈pred(i)}
25)         nl = D[i,j-1]
26)         if label(i) is 'empty'                  // empty row
27)             D[i,j] = np                         // empty-match
28)         else if label(i) is 'simple'            // simple row
29)             D[i,j] = min(np + V[i],             // delete
30)                          nl + H,                // insert
31)                          nd + d(Pi,Tj))         // substitute
32)         else                                    // pumping row
33)             D[i,j] = min(np,                    // empty-match
34)                          nl + H,                // insert
35)                          nl + d(Pi,Tj))         // pump-sub
36) to perform SecondPass (j):
37)     for 1≤i≤m do
38)         np = min {D[k,j] | k∈pred(i)}
39)         D[i,j] = min (D[i,j], np+V[i])          // delete or empty-match
```

Figure 13. (continued)

4.3. Examples

Figure 14(a) shows the D matrix and links for the computation of d(**a(bc*d)***, **abccdcd**). Substring pumping links are indicated by a tall ↶, and a delete or empty match along a back arc is indicated by a tall C.

In this example, only one cell was improved during the second pass (the fourth row in the sixth column, indicated by a hollow number 1). During the first pass, the best we can do is a substitution of **d** for **b**, from node **d** in the previous column, for a total distance of 2. On the second pass, we discover that we can delete the **b**, from node **d** in the current column, for an improved distance of 1.



(a) Distance Matrix

(b)	(c)	(d)
P: **abccccd**	P: **abccdbd**	P: **abccdbcd**
T: **abccdcd**	T: **abccdcd**	T: **abccd-cd**
d: 1	d: 1	d: 1

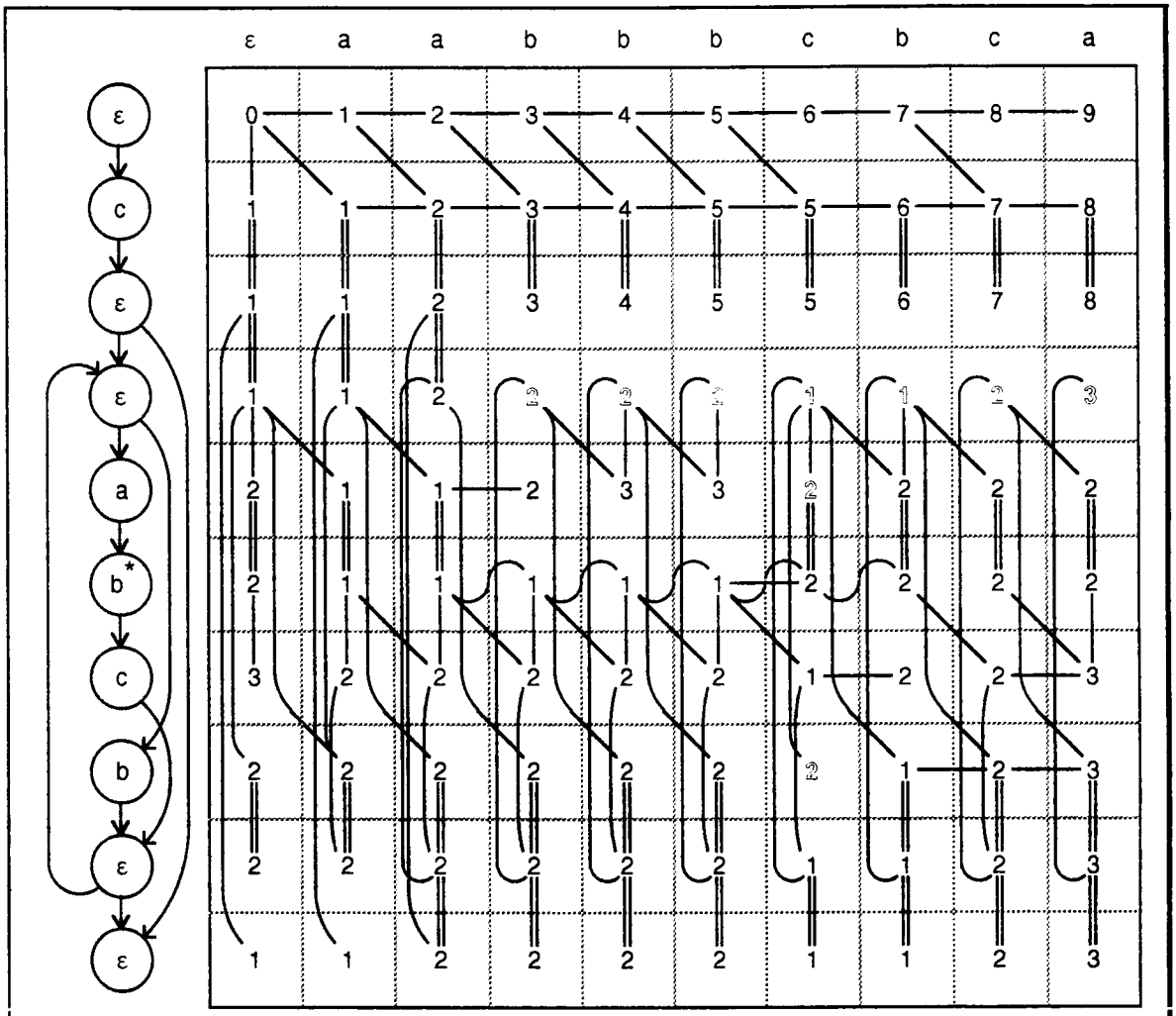
Figure 14. Distance Matrix with Substring Pumping, and Optimal Alignments

Tracing the links, we can find three alignments with the optimal distance 1. Figure 14(b) shows an alignment with the substring **bc*d** pumped only once. A **d** has been substituted for one of the **c**'s (or equivalently for this cost function, the **d** was inserted). The alignment in 14(c) pumps **bc*d** twice, once as **bccd**, then as **bd**. A **c** is substituted for the **b** in the second instance. In 14(d), the alignment again pumps twice, once as **bccd**, then as **bcd**. In the second instance, the **b** is deleted.

Our second example, shown in figure 15, involves the network from figure 8. 15(a) shows the D matrix and links for d(**c(ab*c+b)***, **aabbbcbca**). Several cells were altered by the second pass, as indicated by the hollow numbers.

15(b) shows the trace of one alignment in the matrix, and 15(c) shows the alignment (with the pumpings of substring **ab*c+b** separated). **ab*c+b** is pumped three times, first as **abbbc**, then as **abc**, and finally as **b**. The reader is encouraged to trace the following similar alignments, which pump the substring two, three, and four times, through figure 15(a):

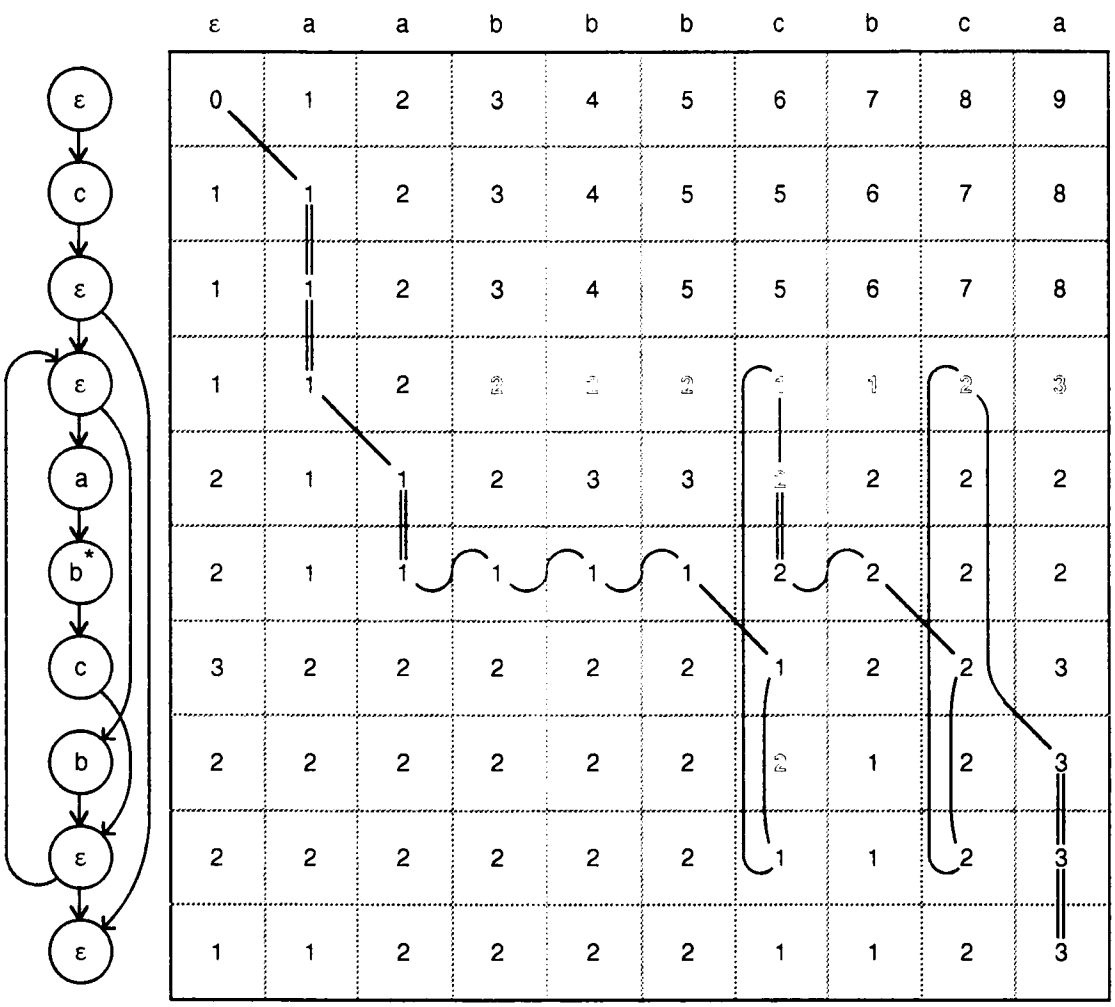
```
P: cabbbcb--      P: cabbbcb-b      P: cabbbcbbb
T: aabbbcbca      T: aabbbcbca      T: aabbbcbca
d: 1      11      d: 1      11      d: 1      11
```



(a) Distance Matrix

Figure 15. Another Distance Matrix with Substring Pumping

(continues)



(b) Trace in Distance Matrix

P: **c abbbc abc b**
T: **a abbbc -bc a**
d: 1 1 1

(c) Alignment

Figure 15. (continued)

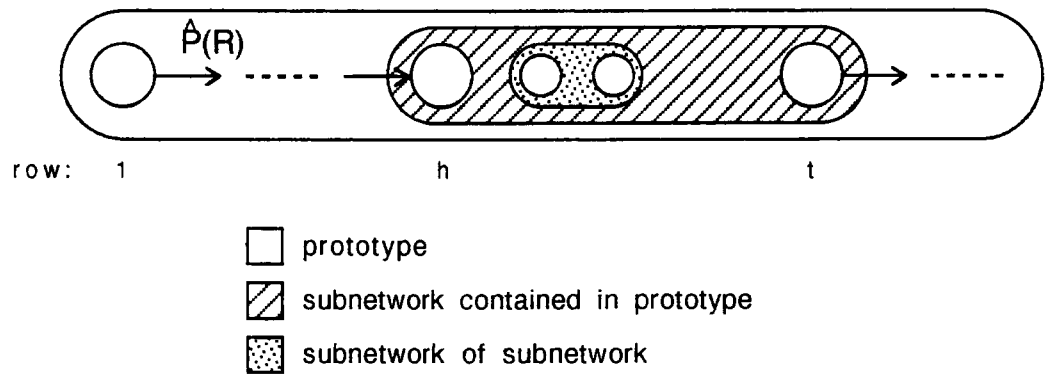


Figure 16. Hierarchy of Subnetworks in Prototype

4.4. Proof

We will show that algorithm E produces, for each column j, a solution to the set of equations enforcing the appropriate recurrence relations for $D[i,j] = d(L_i(\hat{P}),T_{0:j})$. Then, as in algorithm D, $D[m,n] = d(L_m(\hat{P}),T_{0:n}) = d(\hat{P},T)$.

We observe the behavior of the algorithm for a particular column $j \geq 1$ of the D matrix, assuming that column j-1 has been computed correctly. We will show that one application of the second pass recurrence relations is enough to compute each value in the column—additional passes would not change any value in the column.

We make use of the construction of the prototype and look at each subnetwork. Figure 16 shows how a subnetwork may be contained within the prototype and may itself contain other subnetworks. Since a subnetwork may contain a cycle, some of its values may change during the second pass. This change may cause changes in the values of the subnetworks contained within. Further, a subnetwork may be contained in a cycle, so its first value may change in the second pass. However, we can show that a change introduced within a subnetwork will not affect the networks that contain it.

A more compact notation is needed for the discussion of subnetworks. We isolate the subnetwork and consider only those rows between the head (row h) and tail (row t), and denote this network by h...t. For any row i ($h \leq i \leq t$), we denote $d_1(L_i(\hat{P}),T_{0:j})$ by i_1 and $d_2(L_i(\hat{P}),T_{0:j})$ by i_2. Note that we drop the column index j. We generalize the second

pass recurrence relations in (4.4) through (4.8) to allow for a third pass through the column (where d_2 depends on d_1 and d_2, d_3 depends on d_2 and d_3), and denote $d_3(L_i(\hat{P}),T_{0:j})$ by i_3. We also denote the cost of a vertical move (delete or empty-match) into row i by V_i (this is the same as V[i] in algorithm E).

We examine how a subnetwork h...t behaves (how its internal values change) relative to the value at its head node during the second pass. The head value may change ($h_2 \neq h_1$) or not ($h_2 = h_1$). If h...t contained no cycles, then every path from h to t would be feed-forward. As lines 37 through 39 of algorithm E go through h...t on the second pass, each new value considered is simply a series of vertical moves (deletes and empty-matches) from the head. Clearly no value in the subnetwork will change unless the head value does. But since a series of vertical moves cannot have negative cost, any value that changes will not become less than the head value.

If h...t contains a cycle, the analysis above does not apply. However, even if a subnetwork contains a cycle, its head and tail values will behave as though they were part of a feed-forward network. For example, the tail value will change only if the head value does, but will not become less than the head[1]. This is the motivation for the properties below.

If a network $\hat{P}$, indexed by h...t, satisfies property (4.12) below, it is *two-pass computable.* Clearly Algorithm E will produce the correct values for such a network. If $\hat{P}$ satisfies (4.9) through (4.11) as well, it is said to be *strongly two-pass computable.* We will show that every network we use is strongly two-pass computable, by induction on the size of the network.

---

[1] This does <u>not</u> mean that the tail will never be less than the head, only that for this to occur, it must happen during the first pass.

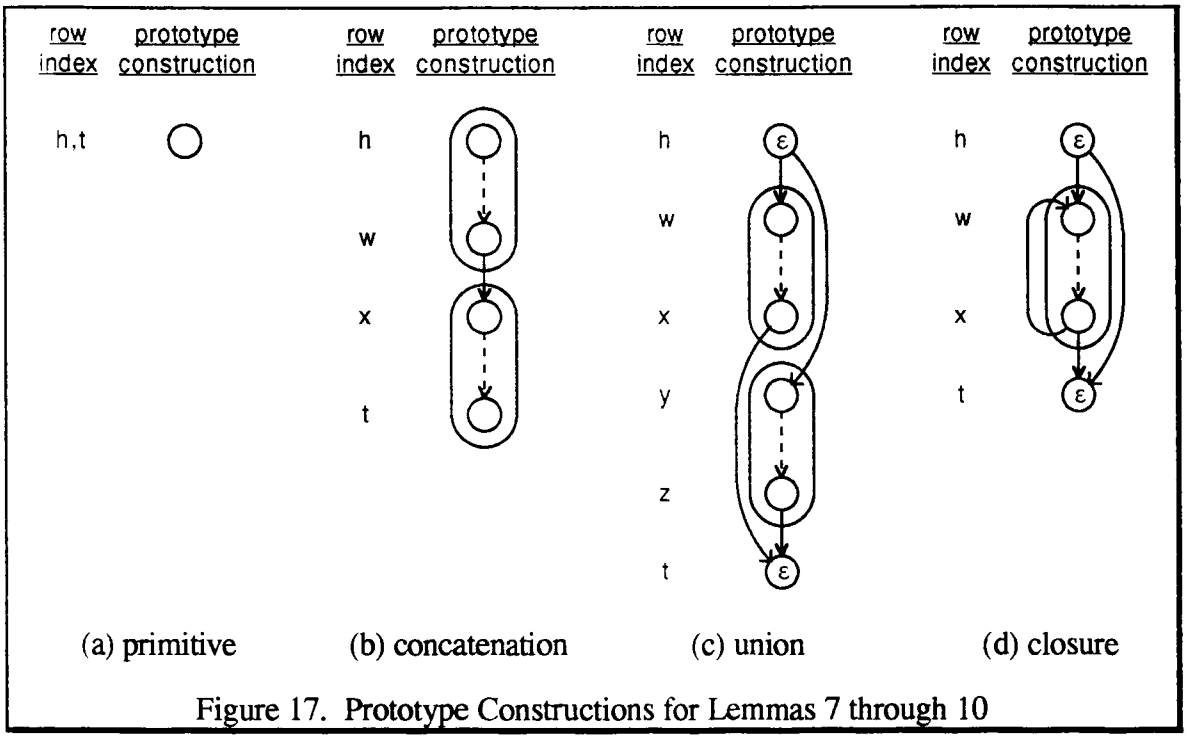


Figure 17. Prototype Constructions for Lemmas 7 through 10

- if $h_2 = h_1$ then $t_2 = t_1$ (4.9)
- if $h_2 \geq t_1$ then $t_2 = t_1$ (4.10)
- if $h_2 < t_1$ then $h_2 \leq t_2$ (4.11)
- if $h_3 = h_2$ then $\forall i$, $h \leq i \leq t$, $i_3 = i_2$ (4.12)

Lemma 6: No value is increased by the second pass. That is,

$\forall i\ 0 \leq i \leq m,\ i_2 \leq i_1$

Proof: Recurrence relations (4.4) and (4.5) have $i_2 = i_1$, while (4.6), (4.7), and (4.8) have $i_2 = \min \{i_1, \ldots\}$. Therefore i_2 cannot be greater than i_1. ▨

Lemma 7: If $\hat{P}$ is constructed from a primitive regular expression, then $\hat{P}$ is strongly two-pass computable.

Proof: Refer to figure 17(a). $\hat{P}$ has only one node, so $h_1 = t_1$, $h_2 = t_2$, and $h_3 = t_3$. We prove the properties (4.9) through (4.12) separately.

7.1: $\hat{P}$ satisfies (4.9): Assume $h_2 = h_1$. Then

$h_2 = h_1 \Rightarrow t_2 = t_1$ (h, t index same row)

7.2: $\hat{P}$ satisfies (4.10): Assume $h_2 \geq t_1$. Then

$h_2 \geq t_1 \Rightarrow h_2 \geq h_1$ (h, t index same row)
$\Rightarrow h_2 = h_1$ (Lemma 6: $x_2 \leq x_1$)
$\Rightarrow t_2 = t_1$ (4.9 on $\hat{P}$)

7.3: $\hat{P}$ satisfies (4.11): $h_2 = t_2$ regardless of t_1. So $h_2 < t_1 \Rightarrow h_2 \leq t_2$ trivially.

7.4: $\hat{P}$ satisfies (4.12): Assume $h_3 = h_2$. Since h is the only row in $\hat{P}$,

$\forall i,\ h \leq i \leq t,\ i_3 = i_2$

Since $\hat{P}$ satisfies properties (4.9) through (4.12), it is strongly two-pass computable. ▨

Lemma 8: If $\hat{P}$ is constructed by concatenating two shorter prototypes that are strongly two-pass computable, then $\hat{P}$ is also strongly two-pass computable.

Proof: Refer to figure 17(b). We first consider the combination of nodes w and x and show that it behaves as a strongly two-pass computable network. We then use this to prove the general case of concatenated prototypes.

8.1: w∘x satisfies (4.9): Since x...t was proper before concatenation, x has only one predecessor—w. So we know

$x_1 = w_1 + V_x$ and $x_2 = \min \{x_1, w_2 + V_x\}$

Now assume $w_2 = w_1$. Then

$w_2 = w_1 \Rightarrow x_2 = \min \{x_1, w_1 + V_x\}$ ($x_2 = \min \{x_1, w_2 + V_x\}$)
$\Rightarrow x_2 = x_1$ ($x_1 = w_1 + V_x$)

8.2: w∘x satisfies (4.10): Assume $w_2 \geq x_1$. Then

$w_2 \geq x_1 \Rightarrow x_1 \leq w_2 + V_x$ ($w_2 \leq w_2 + V_x$)
$\Rightarrow x_2 = x_1$ ($x_2 = \min \{x_1, w_2 + V_x\}$)

8.3: $w \circ x$ satisfies (4.11): Assume $w_2 < x_1$. Then

$$\begin{aligned} w_2 < x_1 &\Rightarrow w_2 \le \min\{x_1, w_2+V_x\} && (w_2 \le w_2+V_x) \\ &\Rightarrow w_2 \le x_2 && (x_2 = \min\{x_1, w_2+V_x\}) \end{aligned}$$

8.4: $w \circ x$ satisfies (4.12): Assume $h_3 = h_2$. Then, by an argument similar to that used in 8.1, $x_3 = x_2$. Therefore,

$$\forall i,\ w \le i \le x,\ i_3 = i_2$$

So $w \circ x$ behaves as a strongly two-pass computable network. We now prove the properties (4.9) through (4.12) on $\hat{P}$ separately.

8.5: $\hat{P}$ satisfies (4.9): Assume $h_2 = h_1$. Then

$$\begin{aligned} h_2 = h_1 &\Rightarrow w_2 = w_1 && (4.9 \text{ on } h \ldots w) \\ &\Rightarrow x_2 = x_1 && (4.9 \text{ on } w \circ x) \\ &\Rightarrow t_2 = t_1 && (4.9 \text{ on } x \ldots t) \end{aligned}$$

8.6: $\hat{P}$ satisfies (4.10): Assume $h_2 \ge t_1$.

There are three subcases— (i) $h_2 \ge w_1$, (ii) $w_2 \ge x_1$, or (iii) both $h_2 < w_1$ and $w_2 < x_1$.

case 8.6.i: $h_2 \ge w_1$

$$\begin{aligned} h_2 \ge w_1 &\Rightarrow w_2 = w_1 && (4.10 \text{ on } h \ldots w) \\ &\Rightarrow x_2 = x_1 && (4.9 \text{ on } w \circ x) \\ &\Rightarrow t_2 = t_1 && (4.9 \text{ on } x \ldots t) \end{aligned}$$

case 8.6.ii: $w_2 \ge x_1$

$$\begin{aligned} w_2 \ge x_1 &\Rightarrow x_2 = x_1 && (4.10 \text{ on } w \circ x) \\ &\Rightarrow t_2 = t_1 && (4.9 \text{ on } x \ldots t) \end{aligned}$$

case 8.6.iii: $h_2 < w_1$ and $w_2 < x_1$

$$\begin{aligned} w_2 < x_1 &\Rightarrow w_2 \le x_2 && (4.11 \text{ on } w \circ x) \\ h_2 < w_1 &\Rightarrow h_2 \le w_2 && (4.11 \text{ on } h \ldots w) \\ &\Rightarrow h_2 \le x_2 && (w_2 \le x_2) \\ &\Rightarrow x_2 \ge t_1 && (\text{assumption 8.6: } h_2 \ge t_1) \\ &\Rightarrow t_2 = t_1 && (4.10 \text{ on } x \ldots t) \end{aligned}$$

8.7: $\hat{P}$ satisfies (4.11): Assume $h_2 < t_1$.

There are four subcases— (i) $h_2 \geq w_1$, (ii) $w_2 \geq x_1$, (iii) $x_2 \geq t_1$, or (iv) none of them ($h_2 < w_1$, $w_2 < x_1$, and $x_2 < t_1$).

case 8.7.i: $h_2 \geq w_1$

$h_2 \geq w_1$	$\Rightarrow t_2 = t_1$	(see case 8.6.i)
	$\Rightarrow h_2 < t_2$	(assumption 8.7: $h_2 < t_1$)
	$\Rightarrow h_2 \leq t_2$	

case 8.7.ii: $w_2 \geq x_1$

$w_2 \geq x_1$	$\Rightarrow t_2 = t_1$	(see case 8.6.ii)
	$\Rightarrow h_2 \leq t_2$	(see case 8.7.i)

case 8.7.iii: $x_2 \geq t_1$

$x_2 \geq t_1$	$\Rightarrow t_2 = t_1$	(4.9 on x...t)
	$\Rightarrow h_2 \leq t_2$	(see case 8.7.i)

case 8.7.iv: $h_2 < w_1$, $w_2 < x_1$, and $x_2 < t_1$

$x_2 < t_1$	$\Rightarrow x_2 \leq t_2$	(4.11 on x...t)
$w_2 < x_1$	$\Rightarrow w_2 \leq x_2$	(4.11 on wox)
$h_2 < w_1$	$\Rightarrow h_2 \leq w_2$	(4.11 on h...w)
	$\Rightarrow h_2 \leq x_2$	($w_2 \leq x_2$)
	$\Rightarrow h_2 \leq t_2$	($x_2 \leq t_2$)

8.8: $\hat{P}$ satisfies (4.12): Assume $h_3 = h_2$. Then

$h_3 = h_2$	$\Rightarrow \forall i,\ h \leq i \leq w,\ i_3 = i_2$	(4.12 on h...w)
	$\Rightarrow x_3 = x_2$	($w_3 = w_2$, 4.12 on wox)
	$\Rightarrow \forall i,\ x \leq i \leq t,\ i_3 = i_2$	(4.12 on x...t)
	$\Rightarrow \forall i,\ h \leq i \leq t,\ i_3 = i_2$	

$\hat{P}$ satisfies properties (4.9) through (4.12), so it is strongly two-pass computable. ⧅

Lemma 9: If $\hat{P}$ is constructed from the union of two shorter prototypes that are strongly two-pass computable, then $\hat{P}$ is also strongly two-pass computable.

Proof: Refer to figure 17(c). The subnetworks w...x and y...z are both strongly two-pass computable, and we could form prototypes equivalent to how...x and hoy...z that would be strongly two-pass computable (according to lemma 8).

9.1: $\hat{P}$ satisfies (4.9): Assume $h_2 = h_1$. Then

$h_2 = h_1$	$\Rightarrow x_2 = x_1$ and $z_2 = z_1$	(4.9 on how...x and hoy...z)
	$\Rightarrow t_2 = \min \{x_1, z_1\}$	($t_2 = \min \{x_2, z_2\}$)
	$\Rightarrow t_2 = t_1$	($t_1 = \min \{x_1, z_1\}$)

9.2: $\hat{P}$ satisfies (4.10): Assume $h_2 \geq t_1$. Without loss of generality we can assume that $t_1 = x_1$ ($t_1 = \min \{x_1, z_1\}$, so a similar argument would then apply to the case where $t_1 = z_1$). We first derive $x_2 = x_1$:

$h_2 \geq t_1$	$\Rightarrow h_2 \geq x_1$	($t_1 = x_1$)
	$\Rightarrow x_2 = x_1$	(4.10 on how...x)

Then there are two subcases— (i) $h_2 \geq z_1$ or (ii) $h_2 < z_1$.

case 9.2.i: $h_2 \geq z_1$

$h_2 \geq z_1$	$\Rightarrow z_2 = z_1$	(4.10 on hoy...z)
	$\Rightarrow \min \{x_2, z_2\} = x_1$	($x_2 = x_1$, $z_2 = z_1$, $\min\{x_1, z_1\} = x_1$)
	$\Rightarrow t_2 = x_1$	($t_2 = \min \{x_2, z_2\}$)
	$\Rightarrow t_2 = t_1$	($t_1 = x_1$)

case 9.2.ii: $h_2 < z_1$

$h_2 < z_1$	$\Rightarrow h_2 \leq z_2$	(4.11 on hoy...z)
	$\Rightarrow z_2 \geq t_1$	(assumption 9.2: $h_2 \geq t_1$)
	$\Rightarrow z_2 \geq x_2$	($t_1 = x_1$ and $x_2 = x_1$)
	$\Rightarrow t_2 = x_2$	($t_2 = \min \{x_2, z_2\}$)
	$\Rightarrow t_2 = t_1$	($x_2 = x_1$ and $x_1 = t_1$)

9.3: $\hat{P}$ satisfies (4.11): Assume $h_2 < t_1$. Then

$h_2 < t_1$	$\Rightarrow h_2 < x_1$	(t_1=min $\{x_1, z_1\} \Rightarrow t_1 \leq x_1$)
	$\Rightarrow h_2 \leq x_2$	(4.11 on how...x)
$h_2 < t_1$	$\Rightarrow h_2 < z_1$	(t_1=min $\{x_1, z_1\} \Rightarrow t_1 \leq z_1$)
	$\Rightarrow h_2 \leq z_2$	(4.11 on hoy...z)
	$\Rightarrow h_2 \leq \min \{x_2, z_2\}$	($h_2 \leq x_2$ and $h_2 \leq z_2$)
	$\Rightarrow h_2 \leq t_2$	($t_2 = \min \{x_2, z_2\}$)

9.4: $\hat{P}$ satisfies (4.12): Assume $h_3 = h_2$. Then

$h_3 = h_2$	$\Rightarrow \forall i,\ w \leq i \leq x,\ i_3 = i_2$	(4.12 on how...x)
$h_3 = h_2$	$\Rightarrow \forall i,\ y \leq i \leq z,\ i_3 = i_2$	(4.12 on hoy...z)
	$\Rightarrow \min \{x_3, z_3\} = \min \{x_2, z_2\}$	($x_3 = x_2$ and $z_3 = z_2$)
	$\Rightarrow t_3 = t_2$	(t_3 =min $\{x_3, z_3\}$, t_2 =min $\{x_2, z_2\}$)
	$\Rightarrow \forall i,\ h \leq i \leq t,\ i_3 = i_2$	

$\hat{P}$ satisfies properties (4.9) through (4.12), so it is strongly two-pass computable. ☒

Lemma 10: If $\hat{P}$ is constructed from the closure of a shorter prototype that is strongly two-pass computable, then $\hat{P}$ is also strongly two-pass computable.

Proof: Refer to figure 17(d). We first show (in 10.1-10.4) that if $x_2 = x_1$, then $\hat{P}$ satisfies (4.9)-(4.12). Then we consider the three possibilities for the assignment of w_2— either w_1, x_1+V_w, or h_2+V_w. The first two of these are taken care of immediately by 10.1-10.4. The third case can also be quickly resolved by 10.1-10.4 whenever $w_2 \geq x_1$.

10.1: if $x_2 = x_1$, $\hat{P}$ satisfies (4.9): Assume $x_2 = x_1$ and $h_2 = h_1$. Then

$h_2 = h_1$	$\Rightarrow t_2 = \min \{h_1, x_1\}$	($t_2 = \min \{h_2, x_2\}$ and $x_2 = x_1$)
	$\Rightarrow t_2 = t_1$	($t_1 = \min \{h_1, x_1\}$)

10.2: if $x_2 = x_1$, $\hat{P}$ satisfies (4.10): Assume $x_2 = x_1$ and $h_2 \geq t_1$.

Then there are two subcases— (i) $h_1 > t_1$ or (ii) $h_1 \leq t_1$.

case 10.2.i: $h_1 > t_1$

$h_1 > t_1$	$\Rightarrow t_1 = x_1$	($t_1 = \min \{h_1, x_1\}$)
	$\Rightarrow t_1 = x_2$	(assumption 10.2: $x_2 = x_1$)
	$\Rightarrow t_2 = \min \{h_2, t_1\}$	($t_2 = \min \{h_2, x_2\}$)
	$\Rightarrow t_2 = t_1$	(assumption 10.2: $h_2 \geq t_1$)

case 10.2.ii: $h_1 \leq t_1$

$h_1 \leq t_1$	$\Rightarrow h_1 \leq h_2$	(assumption 10.2: $h_2 \geq t_1$)
	$\Rightarrow h_1 = h_2 = t_1$	(lemma 6: $h_2 \leq h_1$)
	$\Rightarrow x_1 \geq h_1$	($t_1 = \min \{h_1, x_1\}$)
	$\Rightarrow x_2 \geq h_2$	($h_1 = h_2$ and $x_2 = x_1$)
	$\Rightarrow t_2 = h_2$	($t_2 = \min \{h_2, x_2\}$)
	$\Rightarrow t_2 \geq t_1$	(assumption 10.2: $h_2 \geq t_1$)
	$\Rightarrow t_2 = t_1$	(lemma 6: $t_2 \leq t_1$)

<u>10.3: if $x_2 = x_1$, $\hat{P}$ satisfies (4.11)</u>: Assume $x_2 = x_1$ and $h_2 < t_1$. Then

$h_2 < t_1$	$\Rightarrow h_2 < x_1$	($t_1 = \min \{h_1, x_1\} \Rightarrow t_1 \leq x_1$)
	$\Rightarrow h_2 < x_2$	(assumption 10.3: $x_2 = x_1$)
	$\Rightarrow t_2 = h_2$	($t_2 = \min \{h_2, x_2\}$)
	$\Rightarrow h_2 \leq t_2$	

<u>10.4: if $x_2 = x_1$, $\hat{P}$ satisfies (4.12)</u>: Assume $x_2 = x_1$ and $h_3 = h_2$. Then

$h_3 = h_2$	$\Rightarrow w_3 = \min \{h_2, x_2\} + V_w$	($w_3 = \min \{h_3, x_2\} + V_w$)
	$\Rightarrow w_3 = \min \{h_2, x_1\} + V_w$	($x_2 = x_1$)
	$\Rightarrow w_3 = w_2$	($w_2 = \min \{h_2, x_1\} + V_w$)
	$\Rightarrow \forall i,\ w \leq i \leq x,\ i_3 = i_2$	(4.12 on w...x)
	$\Rightarrow \min \{h_3, x_3\} = \min \{h_2, x_2\}$	($h_3 = h_2$ and $x_3 = x_2$)
	$\Rightarrow t_3 = t_2$	($t_3 = \min \{h_3, x_3\}$, $t_2 = \min \{h_2, x_2\}$)
	$\Rightarrow \forall i,\ h \leq i \leq t,\ i_3 = i_2$	

We now consider the three possibilities for the assignment of w_2— either w_1, x_1+V_w, or h_2+V_w. The first two of these are taken care of immediately.

<u>10.5: if $w_2 = w_1$, $\hat{P}$ satisfies (4.9)-(4.12)</u>: Assume $w_2 = w_1$. Then

$w_2 = w_1$	$\Rightarrow x_2 = x_1$	(4.9 on w...x)
	$\Rightarrow \hat{P}$ satisfies (4.9)-(4.12)	(10.1-10.4)

10.6: if $w_2 = x_1+V_w$, $\hat{P}$ satisfies (4.9)-(4.12): Assume $w_2 = x_1+V_w$. Then

$w_2 = x_1+V_w \Rightarrow w_2 \geq x_1$ ($V_w \geq 0$)
$\Rightarrow x_2 = x_1$ (4.10 on w...x)
$\Rightarrow \hat{P}$ satisfies (4.9)-(4.12) (10.1-10.4)

The remaining possibility for w_2 is h_2+V_w. We can resolve this quickly whenever $w_2 \geq x_1$.

10.7: if $w_2 \geq x_1$, $\hat{P}$ satisfies (4.9)-(4.12): Assume $w_2 \geq x_1$. Then

$w_2 \geq x_1 \Rightarrow x_2 = x_1$ (4.9 on w...x)
$\Rightarrow \hat{P}$ satisfies (4.9)-(4.12) (10.1-10.4)

So we can proceed with the assumption that $w_2 = h_2+V_w < x_1$. We can show that

$w_2 < x_1 \Rightarrow w_2 \leq x_2$ (4.11 on w...x)
$\Rightarrow h_2 \leq x_2$ ($w_2 = h_2+V_w \Rightarrow h_2 \leq w_2$)
$\Rightarrow t_2 = h_2$ ($t_2 = \min\{h_2, x_2\}$)

10.8: $\hat{P}$ satisfies (4.9): Assume $h_2 = h_1$. As noted above, we can also assume that $w_2 = h_2+V_w$. Then

$h_2 = h_1 \Rightarrow w_2 = h_1+V_w$ (assumption: $w_2 = h_2+V_w$)
$\Rightarrow w_2 \geq w_1$ ($w_1 \leq h_1+V_w$)
$\Rightarrow w_2 = w_1$ (lemma 6: $w_2 \leq w_1$)
$\Rightarrow x_2 = x_1$ (4.9 on w...x)
$\Rightarrow t_2 = t_1$ (see case 10.1)

10.9: $\hat{P}$ satisfies (4.10): Assume $h_2 \geq t_1$. As noted above, we can also assume that $t_2 = h_2$. Then

$h_2 \geq t_1 \Rightarrow t_2 \geq t_1$ (assumption: $t_2 = h_2$)
$\Rightarrow t_2 = t_1$ (lemma 6: $t_2 \leq t_1$)

10.10: $\hat{P}$ satisfies (4.11): As noted above, we can assume that $t_2 = h_2$, regardless of t_1. So $h_2 < t_1 \Rightarrow h_2 \leq t_2$ trivially.

10.11: $\hat{P}$ satisfies (4.12): Assume $h_3 = h_2$. As noted above, we can also assume that $w_2 = h_2+V_w < x_1$. Then

$w_2 < x_1 \Rightarrow w_2 \leq x_2$ (4.11 on w...x)

$\Rightarrow w_2 \leq x_2+V_w$ ($V_w \geq 0$)

$\Rightarrow w_3 = \min\{w_2, h_3+V_w\}$ ($w_3=\min\{w_2, h_3+V_w, x_2+V_w\}$)

$\Rightarrow w_3 = \min\{w_2, h_2+V_w\}$ (assumption: $h_3 = h_2$)

$\Rightarrow w_3 = w_2$ (assumption: $w_2 = h_2+V_w$)

$\Rightarrow \forall i,\ h \leq i \leq t,\ i_3 = i_2$ (see case 10.4)

$\hat{P}$ satisfies properties (4.9) through (4.12), so it is strongly two-pass computable. ▨

Theorem 1: All networks constructed by the process described by figures 9 through 12 are strongly two-pass computable.

Proof: The proof is by induction on the length of the prototype. Lemma 7 shows that all single-node prototypes are strongly two-pass computable. Lemmas 8, 9, and 10 show that the construction rules preserve strong two-pass computability. Thus all prototypes are strongly two-pass computable. ▨

Theorem 2: Algorithm E is correct.

$D[m,n] = d(\hat{P},T)$

Proof: By induction on the columns. A cell D[i,j] is correct if it is $d(L_i(\hat{P}),T_{0:j})$.

basis) j=0.

$d(L_0(\hat{P}),T_{0:0}) = d(\varepsilon,\varepsilon) = 0$, and line 8 (of algorithm E) assigns D[0,0] correctly. Each of the remaining cells D[i,0] is computed from the minimum of its predecessors in the prefix. Due to the topology of $\hat{P}$, no backward arc can reduce the cost. So D[i,0] is correct.

inductive step) $j \geq 1$ and column j-1 is correct.

By (2.2), $d(L_0(\hat{P}),T_{0:j}) = d(\varepsilon,T_{0:j}) = d(\varepsilon,T_{0:j-1})+d(\varepsilon,T_j)$. Since D[0,j-1] is correct, $D[0,j] = d(\varepsilon,T_{0:j-1})+d(\varepsilon,T_j)$, so it is correct.

Each cell in column j-1 is correct, so, by theorem 1, rows 1...m of column j of D form a solution to the set of equations enforcing the appropriate recurrence relations for $d(L_i(\hat{P}),T_{0:j})$, so D[i,j] is correct for each i. ⧅

4.5. Complexity

Algorithm E still has time and space complexity O(mn). m is now the length of the regular expression from which $\hat{P}$ was constructed[1]. The number of symbols in $\hat{P}$ is bounded by 2m (lemma 5), so the D matrix is bounded by (2m+1)(n+1). Since no row has more than two predecessors (lemma 4, property (4.1)), the computation per cell is bounded by a constant (at most 5 neighbors on the first pass, and 2 on the second).

---

1 Note that m counts symbols and operators in the regular expression, except for parentheses and any * attached to a single symbol.

5. Extensions

This chapter informally presents results and ideas that are directly related to the work of earlier chapters. We include the results without proof—in most cases they can be proved in a manner similar to those used in earlier chapters. The ideas present possibilities for future work in the area.

5.1. More Primitives

Chapter 3 introduced recurrence relations that permitted pumpable symbols to be represented by a single row in the D matrix. We now describe single row recurrence relations for two other common primitives, $\mathbf{a}^?$ and $\mathbf{a}^+$, and for single symbol unions such as **a+b+c**. These can be added to (3.2) and (3.4) to form a complete set.

An *optional* symbol p, represented by $p^?$, is defined as a single p or the empty string. This primitive can be implemented by the recurrence (5.1). We allow the normal insert and substitute operations, but replace the normal delete with an empty match.

$$d(\hat{P}_{0:i},T_{0:j}) = \min \begin{cases} \text{(E)} & d(\hat{P}_{0:i-1},T_{0:j}) \\ \text{(I)} & d(\hat{P}_{0:i},T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(S)} & d(\hat{P}_{0:i-1},T_{0:j-1}) + d(P_i,T_j) \end{cases} \qquad (5.1)$$

The *positive closure* of a symbol p, represented by p^+, is defined as a non-empty string of p's, of any length. p^+ is equivalent to pp^*, so we could implement it in two rows using (3.2) and (3.4). However, (5.2) gives a single row recurrence for such rows. We allow the normal delete, insert, and substitute operations, along with pump-substitute.

$$d(\hat{P}_{0:i},T_{0:j}) = \min \begin{cases} \text{(D)} & d(\hat{P}_{0:i-1},T_{0:j}) + d(P_i,\varepsilon) \\ \text{(I)} & d(\hat{P}_{0:i},T_{0:j-1}) + d(\varepsilon,T_j) \\ \text{(S)} & d(\hat{P}_{0:i-1},T_{0:j-1}) + d(P_i,T_j) \\ \text{(P)} & d(\hat{P}_{0:i},T_{0:j-1}) + d(P_i,T_j) \end{cases} \qquad (5.2)$$

A series of unions of single symbols, such as **a+b+c+...**, can also be treated as a primitive. Simply define a new symbol x in the prototype alphabet, with the cost function defined by

$$d(x,t) = \min \{ d(\mathbf{a},t),\ d(\mathbf{b},t),\ d(\mathbf{c},t),\ \ldots \}$$

then use recurrence (3.2) for the row. Actually, we can also treat **(a+b+c+...)***, **(a+b+c+...)?**, and **(a+b+c+...)+** as primitives by using one of the recurrences (3.4), (5.1), or (5.2).

This will offer savings in computation time (versus repeated applications of the construction in figure 11) for small finite alphabets, since the single symbol cost function can be precomputed and saved in a two-dimensional table. For large or infinite alphabets, there may be no (or only little) advantage. For example, if the alphabet is all real numbers, and d(p,t)=|p-t|, then d(x,t) = min { |**a**-t|, |**b**-t|, |**c**-t|, ... } which may be no easier to compute than by using the equivalent network construction.

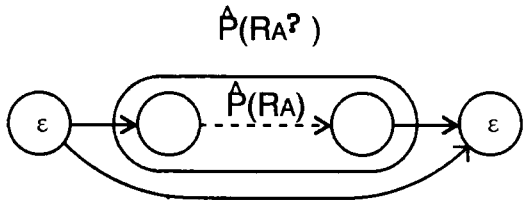
5.2. More Construction Rules

Chapter 4 showed how to build, from a regular expression, a prototype suitable for algorithm A. This section describes construction rules for $R^?$ and R^+.

Figures 18 and 19 show the construction of $\hat{P}(R^?)$ and $\hat{P}(R^+)$. Note that the construction for positive closure is identical to Kleene closure (figure 12), except that the final node is not used. These networks can be shown to be proper and strongly two-pass computable.

5.3. Simplifying Prototypes

It is evident from some of the examples that the prototype construction process in chapter 4 often creates nodes that have no computational purpose. Such nodes can be eliminated without affecting the correctness of the computation.



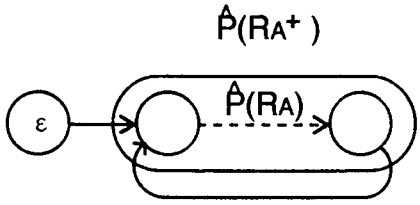
for i = 1,
$P_i = \varepsilon$, labeled empty
pred(i) = {}

for i = 2,
$P_i = (P_A)_{i-1}$
pred(i) = {1}

for $3 \le i \le |P_A|+1$,
$P_i = (P_A)_{i-1}$
pred(i) = $\{ j+1 \mid j \in pred_A(i-1) \}$

for $i = |P_A|+2$,
$P_i = \varepsilon$, labeled empty
pred(i) = $\{1, |P_A|+1\}$

Figure 18. Prototype Construction for Optional Regular Expression



for i = 1,
$P_i = \varepsilon$, labeled empty
pred(i) = {}

for i = 2,
$P_i = (P_A)_{i-1}$
pred(i) = $\{1, |P_A|+1\}$

for $3 \le i \le |P_A|+1$,
$P_i = (P_A)_{i-1}$
pred(i) = $\{ j+1 \mid j \in pred_A(i-1) \}$

Figure 19. Prototype Construction for Positive Closure of a Regular Expression

For example, consider the network used in the example in figure 14. The second and third rows of the D matrix are identical. The ε-node corresponding to the third row is redundant. Removing it, and making the **a**-node a predecessor of the **b**-node and final ε-node, will not affect the results of the algorithm (but make it faster). Figure 20(a) and (b) show how to remove similar ε-nodes. An ε-node at position i that has a single predecessor j is removed by locating all the successors of i—those k such that $i \in pred(k)$—removing i from pred(k) and adding j. After all such nodes are removed from the network, the network can be reindexed to remove the gaps.

It may also be possible to remove an ε-node that has more than one predecessor, if it has only one successor, as shown in figure 20(c) and (d). If the successor has another predecessor, the removal will create a node with more than two predecessors. While this violates (4.1), the number of comparisons performed by algorithm E would not be increased[1].

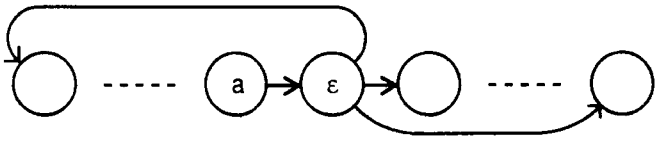
Of course, all ε-nodes could be removed as long as the predecessors are correctly inherited by successors or the removed nodes. However, doing so may actually *increase* the number of comparisons performed by algorithm E. Figure 20(e) shows such a case. As the network stands, one comparison is required to evaluate the ε-node. If the ε-node were removed, this comparison would have to be performed as part of the evaluation of each successor.

5.4. Limiting Pumping Range

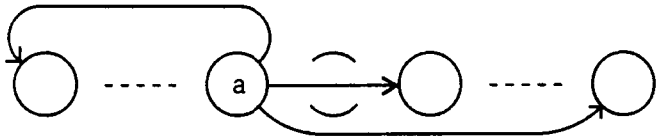
In some cases it may be unrealistic to allow a symbol to pump any number of times—it may be necessary to limit it. We may also insist that a symbol is pumped some minimum number of times as well. We use the notation $p^{f:g}$ to indicate a string of p's no shorter than

---

1 Only the overhead, if any, associated with allowing more than two members in a set.



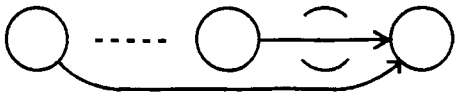
(a) Before Removal



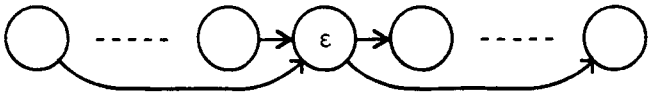
(b) After Removal



(c) Before Removal



(d) After Removal



(e) Defies Removal

Figure 20. Eliminating ε-Nodes

f and no longer than g. Note that we have already addressed the special cases $p^{0:\infty}$ (p^*), $p^{1:\infty}$ (p^+), $p^{0:1}$ ($p^?$), and $p^{1:1}$ (p).

One way of implementing a pumping range $p^{f:g}$ is with f copies of p followed by g-f copies of $p^?$. For example, $\mathbf{a}^{3:5}$ would be replaced by $\mathbf{aaa}^?\mathbf{a}^?$. This will obviously produce correct answers, but has several disadvantages. First, g may be large, and we are not really saving anything. Second, if we are pumping a substring (or subnetwork), we will have many copies of the same subnetwork. Finally, the technique produces many redundant alignments.

Another implementation would involve maintaining a list of g different distances (and edit links) to each cell in the row. The h^{th} entry would contain the best distance ending with the symbol (or substring) pumped h times[1]. Vertical moves into and out of the cell would have to account for the possibility of multiple deletes or insertions. For example, $\mathbf{a}^{3:5}$ can match **aa** with the same cost as deleting an **a**, and $\mathbf{a}^7$ with the same cost as inserting two **a**'s.

If we have a substring with limited pumping $s^{f:g}$, we would have to maintain the lists of size g for each cell within s. This is still manageable so long as s contains no other limited pumping strings or cells.

---

[1] Note that we cannot simply keep those entries that have the best distance, since some with near-optimal distance but fewer pumpings may be part of a path that becomes optimal when the other paths surpass the pumping limit.

6. Conclusion

An efficient algorithm has been presented for computing distances and alignments of strings with regular expressions. The algorithm can be applied to the problem of matching a string with an inferred grammar, so long as the grammar represents a regular language. A process has been described for converting a regular grammar to a prototype for which distances can be computed in O(mn) time and space, and several techniques have been introduced that allow computational improvements for small pieces of the grammar.

LIST OF REFERENCES

List of References

[1] Bunke, H. and Csirik, J. Edit Distance of Run-Length Coded Strings, *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, 1 (1992), 137-143.

[2] Csirik, J. and Bunke, H. Longest k-distance Substrings of Two Strings, *International Conf. Patt. Recognition, Rome*, (1988), 69-71.

[3] Galil Z. and Giancarlo, R. Data Structures and Algorithms for Approximate String Matching, *Journal of Complexity*, 4 (1988), 33-72.

[4] Hecht, M.S. and Ullman, J.D. A Simple Algorithm for Global Data Flow Analysis Programs, *SIAM J. Computing*, 4 (1975) 519-532.

[5] Hirschberg, D.S. A Linear Space Algorithm for Computing Maximal Common Subsequences, *Comm. ACM*, 18 (1975), 341-343.

[6] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, (1979).

[7] Hunt, J.W. and Szymanski, T.G. A Fast Algorithm for Computing Longest Common Subsequences, *Comm. ACM*, 20 (1977), 350-353.

[8] Landau, G.M. and Vishkin, U. Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, 10 (1989) 157-169.

[9] Levenshtein, B.I. Binary Codes with Correction of Deletions, Insertions and Substitutions of Symbols, *Dokl. Acak. Nauk. SSSR*, 163 (1965) 845-848.

[10] Myers, E.W. and Miller, W. Approximate Matching of Regular Expressions, *Bulletin of Mathematical Biology*, 51 (1989) 5-37.

[11] Sellers, P.H. An Algorithm for the Distance Between Two Finite Sequences, *J. Combinator. Theory*, A16 (1974), 253-258.

[12] Sellers, P.H. The Theory and Computation of Evolutionary Distance: Pattern Recognition, *J. Algorithms*, 1 (1980), 359-373.

[13] Thompson, K. Regular Expression Search Algorithm, *Comm. ACM*, 11 (1968), 419-422.

[14] Ukkonen, E. Finding Approximate Patterns in Strings, *J. of Algorithms*, 6 (1985), 132-137.

[15] Ukkonen, E. Algorithms for Approximate String Matching, *Information and Control*, 64 (1985), 100-118.

[16] Wagner, R.A. and Fischer, M.J. The String-to-String Correction Problem, *J. ACM*, 21 (Jan. 1975), 168-173.

[17] Wang, Y.P. and Pavlidis, T. Optimal Correspondence of String Subsequences, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12 (Nov. 1990), 1080-1087.

[18] Wu, S., Manber, U. and Myers, E.W. An O(NP) Sequence Comparison Algorithm, *Information Processing Letters*, 35 (1990), 317-323.

[19] Wu, S., Manber, U. and Myers, E.W. Improving the Running Times for Some String-Matching Problems, *Technical Report, Depart of Computer Science, University of Arizona* (Aug. 1991).

APPENDIX

Appendix. Equivalence of Regular Expressions and Arbitrary Prototypes

A.1. Equivalence

The algorithm presented in this paper allowing substring pumping restricts the prototype to disallow overlapping cycles. This limitation is motivated by the fact that the algorithm would require additional passes on each column, with the number of passes being dependent on the exact topology of the prototype. It turns out that this is not really a limitation, because every arbitrary prototype has an equivalent that is acceptable to the algorithm.

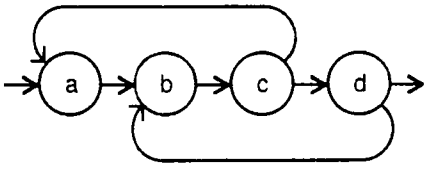
The 'arbitrary' prototypes we are referring to is the class of non-deterministic finite automata (NFAs). A well known result of automata theory is that any NFA can be represented by a regular expression. Regular expressions do not contain overlapping cycles, so given any arbitrary prototype, there is an equivalent regular expression we can use instead.

As an example, consider the prototype $\hat{P}$ shown in figure 21(a), which has two cycles that overlap. The regular expression **ab(c(a+d)b)*cd** denotes the same set of strings as $\hat{P}$. The prototype corresponding to this regular expression is shown in figure 21(b).

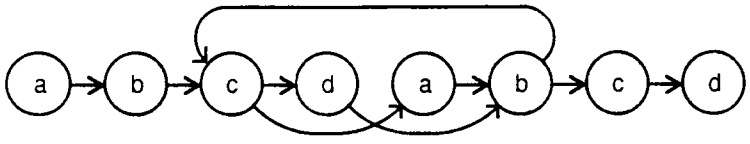
A.2. Construction

A means of constructing a regular expression from a finite automaton is described and proved by Hopcroft and Ullman [6], pages 33-35. This appendix presents a brief overview of the process and a corresponding algorithm.

The construction builds a regular expression, for every pair of states, for the language of strings that will take the automaton from the first state in the pair to the second.



(a) With Overlapping Cycles



(b) Without Overlapping Cycles

Figure 21. Equivalent Prototypes

These regular expressions are built by first considering strings using no intermediate states, then by allowing more and more intermediate states. Upon completion, the regular expressions that take the automaton from the initial state to any final state are combined to give the regular expression for the language recognized by the automaton.

The construction can be expressed by the recurrence below, with the states numbered q_1 through q_s. r_{ij}^k is a regular expression for the strings that can take the NFA from state q_i to q_j, without visiting any state with index greater than k.

$$r_{ii}^0 = \varepsilon + a_1 + \ldots + a_m \qquad 1 \le i \le s \qquad \text{(A.1)}$$

$$r_{ij}^0 = a_1 + \ldots + a_m \qquad 1 \le i \le s, 1 \le j \le s, i \ne j, \{a_1, \ldots, a_m\} = \{a | \delta(q_i, a) = q_j\}$$

$$r_{ij}^k = r_{ij}^{k-1} + r_{ik}^{k-1} (r_{kk}^{k-1})^* r_{kj}^{k-1} \qquad 1 \le i \le s, 1 \le j \le s, 1 \le k \le s$$

Figure 22 shows algorithm F, following the construction in [6], for converting a non-deterministic finite automaton (NFA) to a regular expression. The matrix R holds the regular expressions r_{ij}^k and $\hat{R}$ holds r_{ij}^{k-1}. Lines 7 through 9 initialize each regular

```
1)  given a finite automaton with...
2)      states q_1..q_s
3)      function d where d(i,a) is { j such that δ(q_i,a) = q_j }
4)  R and R̂ are matrices of strings over Σ∪{ε,+,(,),*} over [1..s,1..s]
5)  r (the answer) is a string (like R), representing the regular expression
6)  ∘ is the concatenation operator on strings

7)  for 1≤i≤s
8)      for 1≤j≤s
9)          R[i,j] = ''
10) for 1≤i≤s
11)     R[i,i] = 'ε'
12)     for each a in Σ
13)         for each j in d(i,a)
14)             if R[i,j] isn't empty
15)                 R[i,j] = R[i,j] ∘ '+'
16)             R[i,j] = R[i,j] ∘ j
17) for 1≤k≤s
18)     R̂ = R
19)     for 1≤i≤s, where R̂[i,k] isn't empty
20)         for 1≤j≤s, where R̂[k,j] isn't empty
21)             if R[i,j] isn't empty
22)                 R[i,j] = R[i,j] ∘ '+'
23)             R[i,j] = R[i,j] ∘ '(' ∘ R̂[i,k] ∘ ')'
24)             R[i,j] = R[i,j] ∘ '(' ∘ R̂[k,k] ∘ ')*'
25)             R[i,j] = R[i,j] ∘ '(' ∘ R̂[k,j] ∘ ')'
26) r = ''
27) for 1≤i≤s, where q_i is a final state
28)     if r isn't empty
29)         r = r ∘ '+'
30)     r = r ∘ R[1,i]
```

Figure 22. Algorithm F: Constructing Regular Expression from an NFA

expression as the null language[1]. Lines 10 through 16 complete the initialization of r_{ij}^{0}. Lines 17 through 25 build r_{ij}^{k} from r^{k-1}. Lines 26 through 30 combine the regular expressions for each final state into one regular expression.

Algorithm F is presented only to demonstrate the existence of an algorithm to convert a finite automaton to a regular expression. It is not necessarily the most efficient one, and it generally does not produce a short regular expression. In fact, most small automata are translated to very long regular expressions. Several obvious practical improvements can be made.

Lines 23 through 25 create a regular expression that is always correct, but usually unwieldy. The parentheses in lines 23 and 25 are necessary only if $\hat{R}$[i,k] or $\hat{R}$[k,j] contains any unparenthesized union. Similarly, the parentheses in line 24 are unnecessary if $\hat{R}$[k,k] is a single symbol. If any of $\hat{R}$[i,k], $\hat{R}$[k,k], or $\hat{R}$[k,j] are ε, there is no need to build with them as lines 23 through 25 do (but if all three are ε we must keep one). Early in the algorithm, expressions of the type **a+a** are very common and easy to simplify. A more complicated situation arises when $\hat{R}$[k,k] is of the form ...+ε+...— the ε contributes nothing inside a closure.

Other improvements can undoubtedly be made. The author has implemented the algorithm in the C language. The improvements above reduced the regular expression produced for the 4-state NFA in figure 99(a) to **bc(abc)*d+bc(abc)*d(abc(abc)*d)*(ε+abc(abc)*d)**.[2] However, this is still a good bit larger than the equivalent **ab(c(a+d)b)*cd** determined by inspection.

---

1 The reader should be aware of the difference between the language containing the empty string, which is denoted by the regular expression ε, and the null language, which contains no strings and is represented in algorithm F by an empty string.

2 From an expression about 3 times as long that starts with `((b+εε*b)ε*c)(ε+aε*bε*c)*d + ...`.

Vita

Robert Scott Harris[1] was born in Boalsburg, Pennsylvania on August 28, 1957. He received a Bachelor of Science in Applied Mathematics from Carnegie-Mellon University (Pittsburgh, Pennsylvania) in May, 1979. After employment with Milton Bradley (East Longmeadow, Massachusetts), North American Philips (Knoxville, Tennessee), and Perceptics (also Knoxville), he entered the University of Tennessee, Knoxville in January, 1992, and in August, 1993, received a Master of Science degree in Computer Science.

---

1 Bob.