A Fast Algorithm for the String Editing Problem and Decision Graph Complexity

by
William J. Masek

BA, University of California, Irvine
(1973)

SB, University of California, Irvine
(1974)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1976

Signature of Author .....
Department of Electrical Engineering and
Computer Science, May 14, 1976

Certified by ....
Thesis Supervisor

Accepted by .....
Chairman, Department Committee on Graduate Students

A Fast Algorithm for the String Editing Problem and Decision Graph Complexity

by
William J. Masek

Submitted to the department of Electrical Engineering and Computer Science on May 7, 1976 in partial fulfillment of the requirements for the Degree of
Master of Science

Abstract

The first part describes a fast string-editing algorithm. A string-editing algorithm measures the distance between two character strings by the minimal-cost sequence of deletions, insertions and replacements of symbols needed to transform one string into the other. The longest common subsequence problem can be solved as a special case. An algorithm assuming a finite alphabet with restrictions on the edit costs solves this problem in time $O(nm/\log n)$ where n, m are the lengths of the strings. Bounded intervals of integral linear combinations of the edit costs must not be dense. Otherwise the algorithm might not work.

The second part describes work with decision graphs. Decision graphs are like random-access finite-state machines where each state can only look at one particular input character. They model space complexity problems. Optimal real-time decision graphs are exhibited for counting functions. A time-space trade-off exists using this model. Boolean formulas and contact networks are modelled using decision graphs. Matching upper and lower bounds are shown for the complexity of arbitrary boolean functions -- $O(2^n/n)$. Last deterministic decision graphs model non-deterministic decision graphs with c states using $O(c^{1+\log c})$ states.

THESIS SUPERVISOR: Ronald L. Rivest
TITLE: Assistant Professor of Computer Science and Engineering

Table of Contents

Table of Figures

Acknowledgments

I wish to thank Ron Rivest for his tireless work. First he presented the underlying ideas for the second half of this thesis, then he was very helpful in our many discussions about both problems. He was also instrumental in making this thesis as intelligible as it is.

I would like to thank Mike Fischer for presenting the problem for the first part of the thesis and our discussions concerning it. Bob Cassals was helpful for his critical readings. Finally I must thank Ellen Lewis for the help she gave on using the XGP and supplying the fonts I needed.

Explanatory Note

This thesis describes work done on two unrelated problems. There is no correspondence between theorems or references in the two parts.

Part I -- A Fast Algorithm for the String Editing Problem

1. Introduction

Wagner and Fisher [5] presented an algorithm for determining a sequence of edit transformations that changes one string into another. The execution time of their algorithm is proportional to the product of the lengths of the two input strings.

To define a notion of "distance", use the same 3 operations: (1) inserting a character into a string; (2) deleting a character from a string; and (3) replacing one character of a string with another. Michael Paterson [4] developed a fast algorithm. It runs in time proportional to the product of the lengths of the strings, divided by the log of the length of the longer string, if both the alphabet for the strings is finite, and the number of integral linear combinations of edit costs for the edit operations within any bounded interval is finite.

This algorithm is useful for computing the edit distance between long strings. As a special case it can compute the longest common subsequence of a pair of strings.

1.1 Basic Definitions

The following symbols will be used:

A A string of characters over some alphabet Σ.

$|A|$ The length of string A.

A_n The nth character of the string A ($|A_n|=1$).

$A^{i,j}$ The string $A_i,\ldots,A_j$, ($|A^{i,j}|=j-i+1$).

A^n An abbreviation for $A^{1,n}$.

λ The null string also denoted A^0.

An edit operation is a pair $(a,b)\neq(\lambda,\lambda)$ of strings of length less than or equal to 1, also denoted as $a\to b$. String B results from string A by the edit operation $a\to b$, written "$A\to B$ via $a\to b$", if $A=\sigma a\tau$ and $B=\sigma b\tau$ for some strings σ and τ. Call $a\to b$ a <u>replacement</u> operation if $a\neq\lambda$ and $b\neq\lambda$; a <u>delete</u> operation if $b=\lambda$; and an <u>insert</u> operation if $a=\lambda$.

Let S be a sequence $s_1,\ldots s_m$ of edit operations (alternatively an edit sequence). An S derivation of A to B is a sequence of strings C_0, $C_1,\ldots,$ C_m such that $A=C_0$, $B=C_m$ and for all $1\leq i\leq m$, $C_{i-1}\to C_i$ via s_i. The sequence S takes A to B if there is some S derivation of A to B.

Let γ be an arbitrary cost function assigning a nonnegative real number to each edit operation $a\to b$. Let γ be defined for sequences S of edit operations $s_1,\ldots,s_m$ by letting $\gamma(S)=\sum_{1\leq i\leq m}\gamma(s_i)$. Define the edit distance $\delta_{A,B}$ from string A to string B to be the minimum cost of all sequences of edit operations taking A to B. Formally $\delta_{A,B}=\min\{\gamma(S)\mid S \text{ is an edit sequence taking A to B}\}$.

Assume $\gamma(a\to b)=\delta_{a,b}$ for all edit operations $a\to b$. (Equivalently assume $\gamma(a\to a)=0$ and $\gamma(a\to b)+\gamma(b\to c)\geq\gamma(a\to c)$.) This leads to no loss of generality, for if δ is the distance function associated with a cost

function γ, it is easily verified δ is the distance function associated with the cost function γ' for which $\gamma'(a \to b) = \delta_{a,b}$ and γ' has the desired properties.

Let δ always denote the distance function between the strings A and B, and denote δ_{A^i,B^j} by $\delta_{i,j}$. Write the cost of replacing a with b as $R_{a,b}$, the cost of deleting a as D_a, and the cost of inserting a as I_a.

1.2 Previous Results

The best upper bound known on the time for computing $\gamma_{A,B}$ is $O(|A| \cdot |B|)$ [5]. For infinite alphabets this result is optimal [1]; however, using the idea of the 4 Russians algorithm [2] Michael Paterson reduced the time needed to $O(|A| \cdot |B| / \log(\max(|A|, |B|)))$ if the alphabet was finite and the edit costs were restricted. The matrix-filling algorithm in [5] computes δ by computing $\delta_{i,j}$ for each pair of strings (A^i, B^j) using an $(|A|+1) \times (|B|+1)$ matrix for γ (Figure 1a). Wagner and Fisher [5] showed each element of the matrix is determined by 3 previously computed matrix elements. Theorem 2 describes how the border is computed.

Theorem 1 [5]. For all i, j such that $1 \le i \le |A|$, $1 \le j \le |B|$:

$$\delta_{i,j} = \min(\delta_{i-1,j-1} + R_{A_i,B_j}, \delta_{i-1,j} + D_{A_i}, \delta_{i,j-1} + I_{B_j}).$$

	λ	b	a	b	a	a	a
λ	0	1	2	3	4	5	6
a	1	2	1	2	3	4	5
b	2	1	2	1	2	3	4
a	3	2	1	2	1	2	3
b	4	3	2	1	2	3	4
b	5	4	3	2	3	4	5
b	6	5	4	3	4	5	6

Figure 1a

	λ	b	a	b	a	a	a
λ	0	1	2	3	4	5	6
a	1			2			5
b	2			1			4
a	3	2	1	2	1	2	3
b	4			1			4
b	5			2			5
b	6	5	4	3	4	5	6

Figure 1b

The alphabet is (a, b).
Assume $I = D = 1$, $R_{a,b} = R_{b,a} = 2$
and $R_{a,a} = R_{b,b} = 0$.

Figure 1 Computing Distances with Matrices

The initial values (i, j) of a matrix are all entries with i=0 or j=0. The initial values for the matrix can be computed in O(|A|+|B|) time. An initial value sequence is a sequence of initial values.

Theorem 2[5]. $\delta_{0,0}=0$, and for all i, j such that $1 \le i \le |A|$, $1 \le j \le |B|$, $\delta_{i,0} = \sum_{1 \le r \le i} D_{A_r}$ and $\delta_{0,j} = \sum_{1 \le r \le j} I_{B_r}$.

Each of the |A|·|B| entries in the matrix can be computed in constant time, so the matrix filling algorithm runs in time O(|A|·|B|).

2. The Faster Algorithm

A $m+1 \times m+1$ submatrix is associated with a pair of length m strings and a pair of length m initial value sequences. (The initial values for a submatrix can come from almost anywhere in the matrix.) A submatrix computes the edit distance on the given strings assuming the initial values are as specified. To compute the matrix more efficiently precompute all possible $m+1 \times m+1$ submatrices, on all pairs of length m strings and all possible length m sequences of initial values. Then compute a $|A|/m \times |B|/m$ matrix on the alphabet Σ^m by using the precomputed submatrices (Figure 1b). To prove this is better first show that all submatrices can be computed fast, then that using the submatrices does not change the answer.

2.1 Bounded Intervals

To compute all possible submatrices enumerate all m length strings and all possible length m initial value sequences. The alphabet is assumed finite, so enumerating all strings is easy. There are too many initial value sequences to enumerate economically, so enumerate all sequences of steps instead. Define a <u>step</u> to be the difference between any two adjacent matrix elements. The size of steps in a matrix is bounded. Let Ω = (the set of edit costs) and let $I(\Omega)$ be the associated ideal. The set Ω is <u>sparse</u> only if there exists some small real constant r such that the difference between any two elements of

$I(\Omega)$ is some integral multiple of r. Edit functions mapped from the integers or the rational numbers are sparse, from the real numbers may not be. If the set of edit costs is sparse and the possible steps are bounded then there is a finite number of steps to consider. Consider each initial value sequence as a starting value and a sequence of steps. Showing the steps are bounded takes two operations. First, show the initial value steps are bounded.

Lemma 1. For all i, j; if $1 \le i \le |A|$, $1 \le j \le |B|$,

(i) $(\min D_a) \le \delta_{i,0} - \delta_{i-1,0} \le D_{A_i} \le (\max D_a)$

(ii) $(\min I_a) \le \delta_{0,j} - \delta_{0,j-1} \le I_{B_j} \le (\max I_a)$.

PROOF. (i) By Theorem 2, $\delta_{i,0} = \sum_{1 \le r \le i} D_{A_r}$ and $\delta_{i-1,0} = \sum_{1 \le r \le i-1} D_{A_r}$. Therefore $\delta_{i,0} - \delta_{i-1,0} = D_{A_i}$, so

$(\min D_a) \le \delta_{i,0} - \delta_{i-1,0} \le D_{A_i} \le (\max D_a)$.

(ii) follows by a similar argument. □

And second, show all steps in the matrix are bounded.

Theorem 3. For all i, j; if $1 \le i \le |A|$, $1 \le j \le |B|$, then

(i) $\min((\min R_{a,b}) - (\max I_a), (\min D_a)) \le \delta_{i,j} - \delta_{i-1,j} \le D_{A_i} \le (\max D_a)$

(ii) $\min((\min R_{a,b}) - (\max D_a), (\min I_a)) \le \delta_{i,j} - \delta_{i,j-1} \le I_{B_j} \le (\max I_a)$.

PROOF. (i) Proceed by induction on i+j.

If $i+j \le 0$ the theorem vacuously holds.

Now suppose the theorem holds for all i,j such that $i+j < r$. If $j=0$ Lemma 1 holds and the theorem holds. If $i=0$ it is meaningless.

By Theorem 1, $\delta_{i,j} = \min(\delta_{i-1,j-1} + R_{A_i,B_j}, \delta_{i-1,j} + D_{A_i}, \delta_{i,j-1} + I_{B_j})$. In particular:

$$\delta_{i,j} \leq \delta_{i-1,j} + D_{A_i}$$

or $\delta_{i,j} - \delta_{i-1,j} \leq D_{A_i} \leq (\max D_a)$.

To show the lower bound there are 3 cases corresponding to the 3 ways of computing $\delta_{i,j}$.

Case 1: Suppose $\delta_{i,j} = \delta_{i-1,j-1} + R_{A_i,B_j}$. The inductive hypothesis yields:

$$\delta_{i,j} \geq \delta_{i-1,j} - (\max I_a) + R_{A_i,B_j},$$

$$\delta_{i,j} - \delta_{i-1,j} \geq R_{A_i,B_j} - (\max I_a)$$

$$\geq (\min R_{a,b}) - (\max I_a).$$

Case 2: Suppose $\delta_{i,j} = \delta_{i-1,j} + D_{A_i}$, then

$$\delta_{i,j} - \delta_{i-1,j} = D_{A_i}$$

$$\geq (\min D_a).$$

Case 3: Suppose $\delta_{i,j} = \delta_{i,j-1} + I_{B_j}$. By the inductive hypothesis: $\delta_{i,j-1} - \delta_{i-1,j-1} \geq \min((\min R_a) - (\max I_a), (\min D_a))$, $I_{B_j} \geq \delta_{i-1,j} - \delta_{i-1,j-1}$. Substituting yields

$$\delta_{i,j} - \delta_{i-1,j} \geq I_{B_j} - I_{B_j} + \min((\min R_{a,b}) - (\max I_a), (\min D_a))$$

$$\geq \min((\min R_{a,b}) - (\max I_a), (\min D_a)).$$

(ii) follows by a similar argument. □

Now given the sparse domain show there is a finite number of steps in the matrix.

Corollary 1. Given a sparse set of edit distances Ω the set of possible steps is finite.

PROOF. For all i, j such that $0 \le i \le |A|$ and $0 \le j \le |B|$ $\delta_{i,j}$ is the sum of the costs of a series of edit operations. Therefore the steps are merely linear combinations of Ω. By Theorem 3 the steps are bounded. Since the bounds on the step sizes are known, the number of possible steps is finite. □

2.2 Submatrix Insertion Is Valid

Now show that using a submatrix produces the correct values. First show any constant added to the initial values is uniformly propagated through the entire submatrix. Let A and B be strings. The pair (A, B) is said to have a common suffix of length m if the last m characters of A and B are identical.

Theorem 4. Given the pairs of strings (A, C), and (B, D) with common suffixes of lengths n and m respectively.

Let $p=|A|-n$, $r=|C|-n$, $s=|B|-m$ and $t=|D|-m$.

Assume δ acts on (A, B) and δ' acts on (C, D). If for all i, j such that $0 \le i \le n$, $0 \le j \le m$, $\alpha \in R$,

$$\delta_{p+i,s} = \delta'_{r+i,t} + \alpha,$$

and $\delta_{p,s+j} = \delta'_{r,t+j} + \alpha$

then $\delta_{p+i,s+j} = \delta'_{r+i,t+j} + \alpha.$

PROOF. Proceed by induction on i+j.

If i=0 or j=0 the theorem is true.

Now assume it is true for $i+j<r$, $i\geq 0$ and $j\geq 0$. By Theorem 1:

$$\delta_{p+i,s+j} = \min(\delta_{p+i-1,s+j-1} + R_{A_{p+i},B_{s+j}}, \delta_{p+i-1,s+j} + D_{A_{p+i}}, \delta_{p+i,s+j-1} + I_{B_{s+j}}).$$

By the inductive hypothesis:

$$\delta_{p+i,s+j} = \min(\delta'_{r+i-1,t+j-1} + R_{C_{r+i},D_{t+j}} + \alpha, \delta'_{r+i-1,t+j} + D_{C_{r+i}} + \alpha, \delta'_{r+i,t+j-1} + I_{D_{t+j}} + \alpha)$$

$$= \delta'_{r+i,t+j} + \alpha. \square$$

2.3 Generating All Submatrices

A <u>step sequence</u> is a sequence of possible steps, a step sequence and a starting value determines an initial value sequence. First enumerate all length m strings and length m step sequences. Define $f:\Sigma \rightarrow \{0, \ldots, |\Sigma|-1\}$. Let J= (all possible steps in $I(\Omega)$), then define $g:J \rightarrow \{0, \ldots, |J|-1\}$. Enumerate all m length strings and all sequences of m steps in lexicographic order using f and g. Using an increment mod p algorithm, this can be done in time $O(m2^{km})$ for some appropriate k.

Next calculate the submatrices. This must be done for all pairs

of length m strings and sequences of m steps-- $O(|\Sigma|^{2m}|J|^{2m})$ times. Assume G(i,S) returns the ith step in the step sequence S. Calculate each submatrix with step sequences S1 and S2 on strings C and D using Algorithm Y [5]. The function store saves the matrix T so that it can be easily recovered given the step sequences and the strings. The algorithm takes time $O(m^2)$ for each submatrix, so calculating all of the submatrices takes time $O(m^2 2^{4\ell m})$ for some appropriate ℓ. The correctness of Y is shown in [5] using Theorems 1 and 2.

Algorithm Y

```
for each pair C,D of strings in Σ and step sequences S1 and S2 do
begin
T(0,0):= 0;
for i=1 to m do begin
    T(i,0):= T(i-1, 0) + G(i, S1);
    T(0,i):= T(0, i-1) + G(i, S2);
  end;
for i= 1 to m do
  for j= 1 to m do begin
       k1:= T(i-1, j-1) + R_{C_i,D_j};
       k2:= T(i-1, j) + D_{C_i};
       k3:= T(i, j-1) + I_{D_i};
       T(i, j):= min(k1, k2, k3);
     end;
store (T,S1,S2,C,D);
end;
```

2.4 Computing the Edit Distance

Finally compute δ using the information stored in T. Assume $E(x_0, \ldots, x_m)$ returns the sequence S of steps for the matrix values $x_0, \ldots, x_m$. Assume F(S1, S2, C, D) returns a pointer to the submatrix

with the step sequences S1 and S2 using the strings C and D, and T(P, i, j) returns the (i, j) element of the submatrix pointed to by P. (Note F and store are each other's inverses.) The functions E and F can both be computed in time O(m). The additive constant result (Theorem 4) ensures the algorithm works.

Algorithm Z

```
δ(0,0):=0;
for i=1 to |A| do δ_{i,0}:= δ_{i-1,0} + D_{A_i};
for j=1 to |B| do δ_{0,j}:= δ_{0,j-1} + I_{B_j};
for i=0 to |A|-m by m do
  for j=0 to |B|-m by m do begin
    S1:= E(δ_{i,j}, ..., δ_{i+m,j});
    S2:= E(δ_{i,j}, ..., δ_{i,j+m});
    P:= F(S1, S2, A^{i+1,i+m}, B^{j+1,j+m});
    for d=1 to m-1 do begin
      δ_{i+m,j+d}:= δ_{i,j} + T(P, m, d);
      δ_{i+d,j+m}:= δ_{i,j} + T(P, d, m);
     end
  end
```

The timing analysis of Z is straightforward, Algorithm Z runs in time $O(|A|\cdot|B|/m)$. Choosing $m = \min((\log n)/k, (\log n)/4\ell)$ the entire Algorithm (Y and Z) runs in time $O(|A|\cdot|B|/\log(\max(|A|, |B|)))$. If $|A| >> |B|$ the algorithm uses time $O(|A|\cdot|B|/\log|A|)$ using $\log|A| \times \log|A|$ submatrices.

3. Sparseness Is Necessary

The last section used an nxn matrix to compute the edit distance between two strings assuming a finite alphabet and a sparse domain.

This section presents an example demonstrating sparseness of the set of edit costs is a necessary condition for using Algorithms Y, Z.

3.1 Computation with Paths

All possible ways of changing one string into another using matrices must be described. A path in the matrix (Figure 2) $P((i,j), (k,\ell))$ is defined as a starting point (i,j), an ending point (k,ℓ), and a sequence of edit operations, such that $k-i$= the number of replacements plus deletions and $\ell-j$= the number of replacements plus insertions. A replacement moves the path one space diagonally $(i,j) \rightarrow (i+1,j+1)$; an insertion moves the path one space to the right $(i,j) \rightarrow (i,j+1)$; and a deletion moves the path one space down $(i,j) \rightarrow (i+1,j)$. The number of operations in P is $|P|$. Write $P(k,\ell)$ for $P((0,0), (k,\ell))$. The concatenation of two paths $P((i,j), (k,\ell))$ and $Q((k,\ell), (s,t))$ is written $P \circ Q$. The edit operations for $P \circ Q$ are the edit operations for P followed by those for Q. Let $H_p(i)$ be the ith edit operation applied by P for $1 \leq i \leq |P|$.

Each operation is associated with the letters it affects. Let C(P) be the sum of the costs for each operation, or more precisely $C(P) = \sum_{1 \leq i \leq |P|} \gamma(H(i))$. An alternate definition for δ is $\min\{ C(P(|A|, |B|))\}$.

Describe the eccentricity of $P((i,j), (k,\ell))$. Define d as:

	λ	a	b	a	b	a	a
λ	0	1	2	3	4	5	6
b	1	2	1	2	3	4	5
a	2	1	2	1	2	3	4
a	3	2	2	2	2	2	3
b	4	3	2	3	2	3	4
a	5	4	3	2	3	2	3
b	6	5	4	3	2	3	4

Assume $R_{a,a} = R_{b,b} = 0$, $R_{a,b} = 1$,
$R_{b,a} = 2$, and $I = D = 1$.

The underlined entries of the matrix form the path
$P((0,0), (6,4)) = P(6,4)$.
$H_P = (D, D, I, R, R, R, D)$, and $D(P) = \{0, 1, 2\}$.
$C(P) = 8$, so P is not optimal.

Figure 2 Paths in Matrices

$$d(m) = \begin{cases} i-j & \text{if } m=0 \\ d(m-1)+1 & \text{if } H(m) \text{ is a delete operation} \\ d(m-1)-1 & \text{if } H(m) \text{ is an insert operation} \\ d(m-1) & \text{if } H(m) \text{ is a replace operation} \end{cases}$$

The eccentricity of P is $D(P) = \{d(m) \mid 0 \leq m \leq |P|\}$. The function d describes how far the path gets from the center. The set D gives all distances P can get from the center. If D is a singleton set, then P consists of just replacements. Now describe some simple results giving the effects of concatenation and eccentricity on costs.

Lemma 2. For all i, j, k, ℓ, m, n such that $1 \leq i \leq k \leq m \leq |A|$ and $1 \leq j \leq \ell \leq n \leq |B|$ and there are paths $P((i,j), (k,\ell))$, $Q((k,\ell), (m,n))$ such that

$|P| = s$ and $|Q| = t$:

(i) $C(P \circ Q) = C(P) + C(Q)$;

(ii) if $D(P) = (d)$ then $C(P) = \sum_{1 \le r \le |P|} R_{A_{i+r}, B_{j+r}}$.

PROOF. (i) By the definition of $C(P)$

$$C(P) + C(Q) = \sum_{|P|} \gamma(H_P(i)) + \sum_{|Q|} \gamma(H_Q(j))$$

Since $H_{P \circ Q}(i) = H_P(i)$ and $H_{P \circ Q}(|P|+i) = H_Q(i)$,

$$C(P) + C(Q) = \sum_{|P|+|Q|} \gamma(H_{P \circ Q}(i))$$
$$= C(P \circ Q).$$

(ii) Since $D(P) = (d)$, $H_P(i) =$ replace. By the definition of $C(P)$, $C(P) = \sum_{1 \le r \le |P|} R_{A_{i+r}, B_{j+r}}$. □

3.2 The Cost Function Example

Define the cost function for the example. Let $\Sigma = (a, b, c)$, then for all $\alpha, \beta \in \Sigma$ define the cost function γ:

$R_{\alpha,\beta} = 0$ for $\alpha = \beta$

$R_{a,b} = R_{b,a} = \pi$

$R_{c,b} = R_{c,a} = R_{a,c} = R_{b,c} = 6$

$I_\alpha = D_\alpha = 20$.

Use γ to generate A and B (Figure 3) as follows.

Set $A_1 = b$, $B_1 = a$, $S(1) = 20$, $M(1) = \pi$;

If k is odd let $A_k = b$, $B_k = a$, $S(k) = S(k-1) + R_{A_k, B_{k-1}}$, $M(k) = M(k-1) + R_{A_k, B_k}$;

If k is even and $S(k-1) + 20 < M(k-1) + 2\pi$

then $A_k=B_k=c$, $S(k) = S(k-1)+R_{A_k,B_{k-1}}$, $M(k) = M(k-1)+R_{A_k,B_k}$;
else $A_k=a$ and $B_k=b$; $S(k) = S(k-1)+R_{A_k,B_{k-1}}$, $M(k) = M(k-1)+R_{A_k,B_k}$;

A^{50} = bababababcbababcbabcbababcbabcbabcbababcbabcbaba

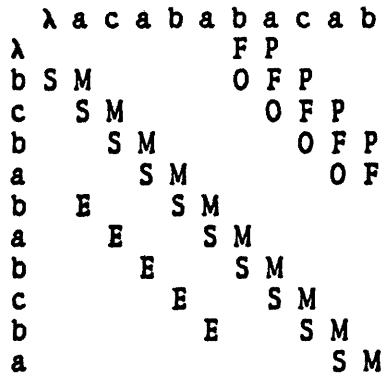
B^{50} = abababababacababacabacababacabacabacababacabacabab

Figure 3 The first 50 characters of A and B

The matrix used to compute γ has O(n) steps. In the example M(k) is the cost of P with H_P= (replace^k) and S(k) is the cost of Q with H_Q= (delete, replace^{k-1}). If P and Q are optimal the Algorithm Y,Z will not work because the number of possible step sequences will not be finite. Inserting c's in the construction of A and B is vital. If no c's were inserted Q would become optimal for computing P (add an insertion at the end of the path) after 13 characters ($13\gamma > 40$). When a 'c' is inserted P becomes more efficient than Q at that point. The alphabet and cost function are designed so the matrix is symmetric across the main diagonal.

3.3 Symmetries in the Matrix

There are many useful symmetries in the strings A and B. The even diagonals behave like M, and the odd diagonals behave like S. Figure 4 describes even and odd diagonals.



M is the path M(i) uses.
S is the path S(i) uses.
E is an even diagonal, D(E) = 4.
O is an odd diagonal, D(O) = -5.
O and P are adjacent odd diagonals.
F is an even diagonal adjacent to O and P.

Figure 4 Diagonals in Matrices

Lemma 3. For all i, j such that $1 \le i \le |A|$, $1 \le j \le |B|$, $k \ge 0$--

(i) if $|i-j|$ is even $A_i = B_j = c$ or $A_i \ne B_j$;

(ii) if $|i-j|$ is odd $A_i \ne B_j$ iff $A_i = c$ or $B_j = c$.

PROOF. (i) If $|i-j| = 2k$, i and j are both even or both odd. If they are odd $A_i = b$ and $B_j = a$. If they are both even one of $A_i = B_j = c$; $A_i = a$ and $B_j = b$ or c; or $B_j = b$ and $A_i = a$ or c must hold.

(ii) If $|i-j| = 2k+1$ i and j are alternatively even and odd.

If i is even and j odd $A_i = a = B_j$ or $A_i = c \ne a = B_j$.

If j is even and i odd $A_i=b=B_j$ or $A_i=b\neq c=B_j$. □

The cost function is symmetric.

Lemma 4. For all i, j such that $0\leq i\leq|A|$, $0\leq j\leq|B|$, $R_{A_i,B_j} = R_{A_j,B_i}$.
PROOF. An examination of R shows it is invariant whenever a's and b's are transposed. In addition switching c's from insert to delete and vice versa do not change R. □

Now show symmetry across the main diagonal in the matrix used to solve δ.

Lemma 5. $\forall$ i, j such that $0\leq i\leq|A|$, $0\leq j\leq|B|$, $\delta_{i,j}= \delta_{j,i}$.
PROOF. The insert and delete costs are all equal. The replacement costs are equal by Lemma 4. Since $A^i=B^i$ with the a's and b's switched, any minimum value for $\delta_{i,j}$ will be a minimum value for $\delta_{j,i}$. □

To show S and M model δ look at all ways of changing A into B. Compare the costs of different sections of paths with S and M. The necessary relationships do not hold for paths with either coordinate less than 10. The next lemma allows these paths to be ignored.

Lemma 6. For all i, j such that $0 \le i \le |A|$, $0 \le j \le |B|$ where $i \le 10$ or $j \le 10$:

(i) if $i > j$ then H_p= (delete, replace^j, delete^{i-j-1}) is optimal.

(ii) if $i < j$ then H_p= (insert, replace^i, insert^{j-i-1}) is optimal.

(iii) if $i = j$ then H_p= (replace^i) is optimal.

PROOF. (i) If $i > j$ then $|A^i| > |B^j|$, so at least i-j deletions from A are necessary to match their lengths Each deletion is of cost 20, so $\delta_{i,j} \ge 20(i-j)$.

For $k \le 10$ $S(k-1) + 20 \ge M(k-1) + 2\pi$ holds. Therefore $A_\ell = B_\ell = c$ implies $\ell > 10$. Consider the path P, all of the replacements have cost 0, so $C(P) = 20(i-j)$. This is the lower bound for $\delta_{i,j}$, so P is optimal.

(ii) follows by symmetry.

(iii) Calculate δ in the beginning of the matrix to confirm it is M. □

3.4 Costs Along the Odd Diagonals

The next lemma gives bounds for the costs associated with subpaths. It is important to remember that in an odd diagonal section ($D(P) = (2k+1)$) $A_i = B_j = c$ cannot occur. The cost of the diagonal is 6 times the number of c's in the relevant portions of A and B. Let E_p be the positions of c's in the relevant parts of A and F_p be the positions of c's in the relevant parts of B. More formally, if $P((i,j), (k,\ell))$ is a path then $E_p = \{e \mid i < e \le k \text{ and } A_e = c\}$ and $F_p = \{f \mid j < f \le \ell \text{ and } B_f = c\}$. If there is no possibility of ambiguity the subscripts will be omitted.

Lemma 7. Given $P((i,j), (k,\ell))$, if $D(P)=(m)$ odd then $C(P)= 6(|E|+|F|)$.

PROOF. Let $|P| = s$, since $D(P)= (m)$ Lemma 2(ii) holds.

$$C(P) = \sum_{1\le r\le s} R_{A_{i+r},B_{j+r}}.$$

But all of those terms but the ones with c's being inserted or deleted are 0 (Lemma 3 and equal replacements are 0), so C(P) is just 6 times the number of c's in the relevant portions of A and B. There are precisely $|E|+|F|$ of these terms, so $C(P)= 6(|E|+|F|)$. □

Equal length adjacent odd diagonals affect almost identical portions of A and B; however, one diagonal can only affect 2 string positions not affected by the other diagonal. And even if both string positions are 'c' one string cannot cost more than 12 units more than the other.

Lemma 8. $\forall$ i,j,k,ℓ such that $1\le i\le k\le |A|$, $1\le j\le \ell\le |B|$, there exists paths $P((i,j), (k,\ell))$ and $Q((i-1,j+1), (k-1,\ell+1))$ so that if $D(P)= (r)$ and $D(Q)= (r+2)$, odd then $C(P)+12 \ge C(Q)$.

PROOF. Lemma 7 says $C(P)= 6(|E_P|+|F_P|)$ Since $E_P + F_P + 2 \ge E_Q + F_Q$, $C(P) \ge 6(|E_Q|+|F_Q|-2)$, or $C(P)+12\ge C(Q)$. □

Now compare any odd diagonal path with a center odd diagonal path. For every 'c' on the outside diagonal there may be two in the center, except for the edges. Indeed since a center path looks at slightly

different sections there may be one more 'c' on the outside. The outside odd diagonal may cost 6 units more than the center.

Lemma 9. $\forall$ i,j,k,ℓ such that $1\le i\le k\le |A|$, $1\le j\le \ell\le |B|$, assume $P((i,j),(k,\ell))$ exists such that $D(P)=(2s+1)$.

(i) if $|E|\ge|F|$ then $C(P)\le S(k)-S(i)+6$,

(ii) if $|E|\le|F|$ then $C(P)\le S(\ell)-S(j)+6$.

PROOF. (i) Lemma 6 says $C(P)=6(|E_p|+|F_p|)$. Assume $i>j$, and define $Q((i,i-1),(k,k-1))$ such that $D(Q)=(1)$.

Since every $A_i=c$ implies $B_i=c$ then if $e\in E_Q$ then $e\in F_Q$ unless $e=k$, so $|E_Q|\le|F_Q|+1$.

Then $C(P)=6(|E_p|+|F_p|)\le$

$\le 6(|E_p|+|E_p|)=$

$=6(|E_Q|+|E_Q|)\le$

$\le 6(|E_Q|+|F_Q|+1)$

$C(P)\le C(Q)+6$.

$C(Q)=S(k)-S(i)$, therefore $C(P)\le S(k)-S(i)+6$.

The case $i<j$ follows by similar argument.

For (ii) prove the lemma for $Q((j-1,j),(\ell-1,\ell))$ then use symmetry to show the lemma for $Q((j,j-1),(\ell,\ell-1))$.

3.5 Costs along the Even Diagonals

The adjacent even diagonals are not comparable, but the even diagonals (D(P)=2m) are comparable with adjacent odd (D(P)=2m-1) diagonals. If the relationships between an even diagonal and the center even diagonal (D(P)= 0) can be computed, then given the relationship between S and M, the even-odd diagonal relationship can be evaluated. First show the center even diagonal is an upper bound for the outer even diagonals.

Lemma 10. $\forall$ i, j, k, ℓ such that $1 \le i \le k \le |A|$, $1 \le j \le \ell \le |B|$, if the paths P((i, j), (k, ℓ)) and D(P)= {2m$\neq$0}; then

(i) $C(P) \ge M(k) - M(i)$,

(ii) $C(P) \ge M(\ell) - M(j)$.

PROOF. (i) Define Q=((i, i), (k, k)) such that D(Q)=0. Assume C(Q) > C(P). Then there exists some pair of terms for C(P) and C(Q) such that $R_{A_{i+r},B_{j+r}} < R_{A_{i+r},B_{i+r}}$.

Case 1. $A_{i+r} = c$. By the construction $B_{i+r} = c$, this gives $\delta_{i+r,i+r} = 0$ the smallest possible cost for an edit transformation.

Case 2. $A_{i+r} \neq c$. The construction gives $R_{A_{i+r},B_{i+r}} = \pi$. By Lemma 3(i) since D(P) is even $A_{i+r} \neq B_{j+r}$, so $R_{A_{i+r},B_{j+r}} \ge \pi$.

So $C(P) \ge M(k) - M(i)$.

(ii) follows by a similar argument. □

3.6 Interval Bounds

If S and M are used to compare the even and odd diagonals then bounds are necessary for $M(k)-S(k)$. The definitions for S and M yield the relationship $20 \geq M(k)-S(k) > 8-\pi$, but only after S and M have had a chance to get started (about k=10).

Lemma 11. $\forall k\ k \geq 10,\ 20 \geq M(k)-S(k) > 8-\pi$.

PROOF. If k=10 $S(10) = 20$ and $M(10) = 10\pi$ and $20 \geq 10\pi-20 > 8-\pi$.

So $M(k)-S(k)$ starts in the interval $(8-\pi, 20]$. Show it can never get out of the interval. While $A_k \neq c$, $M(k)-S(k)$ is increasing, so the maximum value for $M(k)-S(k)$ occurs at the largest k when k is odd and $S(k-2)+20 \geq M(k-2)+2\pi$, or $M(k-2)-S(k-2) \leq 20-2\pi$.

This gives $M(k) = M(k-2)+2\pi$ and $S(k) = S(k-2)$.

Substituting yields

$$\begin{aligned} M(k)-S(k) &= M(k-2)+2\pi-S(k-2) \\ &\leq 20-2\pi+2\pi \\ &= 20. \end{aligned}$$

Whenever a 'c' is inserted $M(k)-S(k)$ gets smaller. The smallest $M(k)-S(k)$ occurs just after the smallest $M(k)-S(k)$ difference where a 'c' has been inserted. If $A_k = c$ then $S(k) = S(k-1)+6 = S(k-2)+12$ and $M(k) = M(k-1)+\pi = M(k-2)+\pi$. The minimum value occurs when k is odd and $20+S(k-2) < M(k-2)+2\pi$, or $M(k-2)-S(k-2) > 20-2\pi$. In that case

$$M(k)-S(k) = M(k-2)+\pi-S(k-2)-12$$

$> 20-2\pi+\pi-12$

$> 8-\pi.$ □

3.7 Sparseness Is Necessary

Now show S(k) represents $\delta_{k,\ k-1}$ and M(k) represents $\delta_{k,k}$. Use an induction on the eccentricity of the optimal paths.

Lemma 12. For all $k \geq 1$

(i) $S(k) = \delta_{k,k-1}$

(ii) $M(k) = \delta_{k,k}$

(iii) $S(k) = \delta_{k-1,k}$.

PROOF. (i), (ii). Construct P(k,k) and Q(k,k-1) such that H_P= (replace^k) and H_Q= (delete, replace^k) then show $\delta_{k,k}$ = C(P) (= M(k)) and $\delta_{k,k-1}$ = C(Q) (= S(k)). Since δ is the minimum over all paths, $\delta_{k,k} \leq$ M(k), and $\delta_{k,k-1} \leq$ S(k).

Show equality by a double induction. First on k, the length of path, then on the eccentricity of the "better" path.

For $k \leq 10$ (i) and (ii) hold by Lemma 6 (M and S have not stabilized yet).

For $k > 10$, assume (i) or (ii) does not hold for some k. Let k be the least value where (i) or (ii) fails. If both fail for the same k the proof for (i) fails still holds. There are three cases.

Case 1 $M(k) > \delta_{k,k}$ and $S(k) = \delta_{k,k-1}$. By the inductive hypothesis and

symmetry $S(k) = \delta_{k,k-1} = \delta_{k-1,k}$. By Theorem 1 $\delta_{k,k} = \min(M(k), S(k)+20, S(k)+20)$. Since $\delta_{k,k} \neq M(k)$, $M(k) > S(k)+20$ or $M(k)-S(k) > 20$ which contradicts Lemma 11.

Case 2 $S(k) > \delta_{k,k-1}$ and $S(k) > M(k)+20$. By Theorem 1 $\delta_{k,k-1} = \min(M(k-1)+20, S(k), \delta_{k,k-2}+20)$. Then $M(k) \leq M(k-1)+\pi$, $M(k)-S(k) > 8-\pi$ (Lemma 11). Substituting yields $M(k-1)+\pi-S(k) > 8-\pi$ or $M(k-1)-8+2\pi > S(k) > M(k-1)+20$.

Case 3 $S(k) > \delta_{k,k-1}$ and $S(k) > \delta_{k,k-2}+20$. This means there exists a path $R(k,k-1)$ such that $C(R) < S(k)$ and $\exists c \in D$ such that $c > 1$. Note symmetry guarantees that for all c, $c \geq 0$. Show no such R exists by induction on the eccentricity of R.

It holds for $\max(D(R)) \leq 1$, the induction hypothesis.

The induction variable is m, the induction hypothesis is $\max(D(R)) \leq 2m+1$ and $C(R)$ is optimal then $\max(D(R)) \leq 1$. The induction step shows $\max(D(R)) \leq 2m+3$ and $C(R)$ is optimal implies $\max(D(R)) \leq 1$. This implies $C(Q)$ is optimal. Assume there exists a path R such that $C(R) < S(k)$ and $\max(D(R)) \leq 2m+3 = p+2$.

Divide R into sections where $d(R) > p$ and $d(R) \leq p$. Let $U((i,i-p),(j,j-p))$ be a section of R where $\max(D(U)) \geq p$, and $d(U) = p$ exactly twice (at the end points). Define $T((i,i-p),(j,j-p))$ such that $D(T) = \{p\}$. If $C(T) \leq C(U)$ then the subpath U of R can be replaced with T to get a path which is no worse than R. Now show $C(T) \leq C(U)$.

Split U into 3 types of sections.

(1) U_I= (i| H_U(i)= insert or delete), all insertions and deletions of Q.

(2) U_E= (i| H_U(i)= replace and d(i)= p+1), all replacements along the p+1 (even) diagonal.

(3) U_O= (i| H_U(i)= replace and d(i)= p+2), all replacements along the p+2 (odd) diagonal.

U_E and U_O can be organized in sections of one or more consecutive replacements. Compute the cost of all U_E and U_O sections by summing the costs of each U_E and U_O section.

Compare the costs of the paths T and U. The replacements of T will be mapped onto the operations of U. Figure 5 describes all possible paths for U. The sections of U can be split into two types -- E= (DVI, R^*, DVI) which pass through state E, and O= (R^*) which stay in state O. For each subpath of E with r+2 map subpaths of T with r+1 replacements onto it as follows (Figure 6). Apply the mappings to each subpath of U in order. If more c's are inserted than deleted in that subpath then map the first r replacements of T onto the r replacements of E, and map the r+1st replacement of T onto the second insert or delete of E. Otherwise map the 2nd thru r+1st replacements of T onto the r replacements of E and the first replacement of T onto the second insert or delete of E. Each r replacements of O are mapped onto by the next r replacements of T in order.

Redistribute the costs for the sections of U as follows.

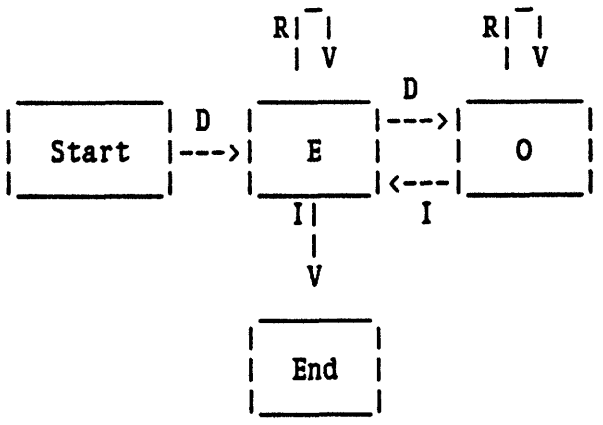
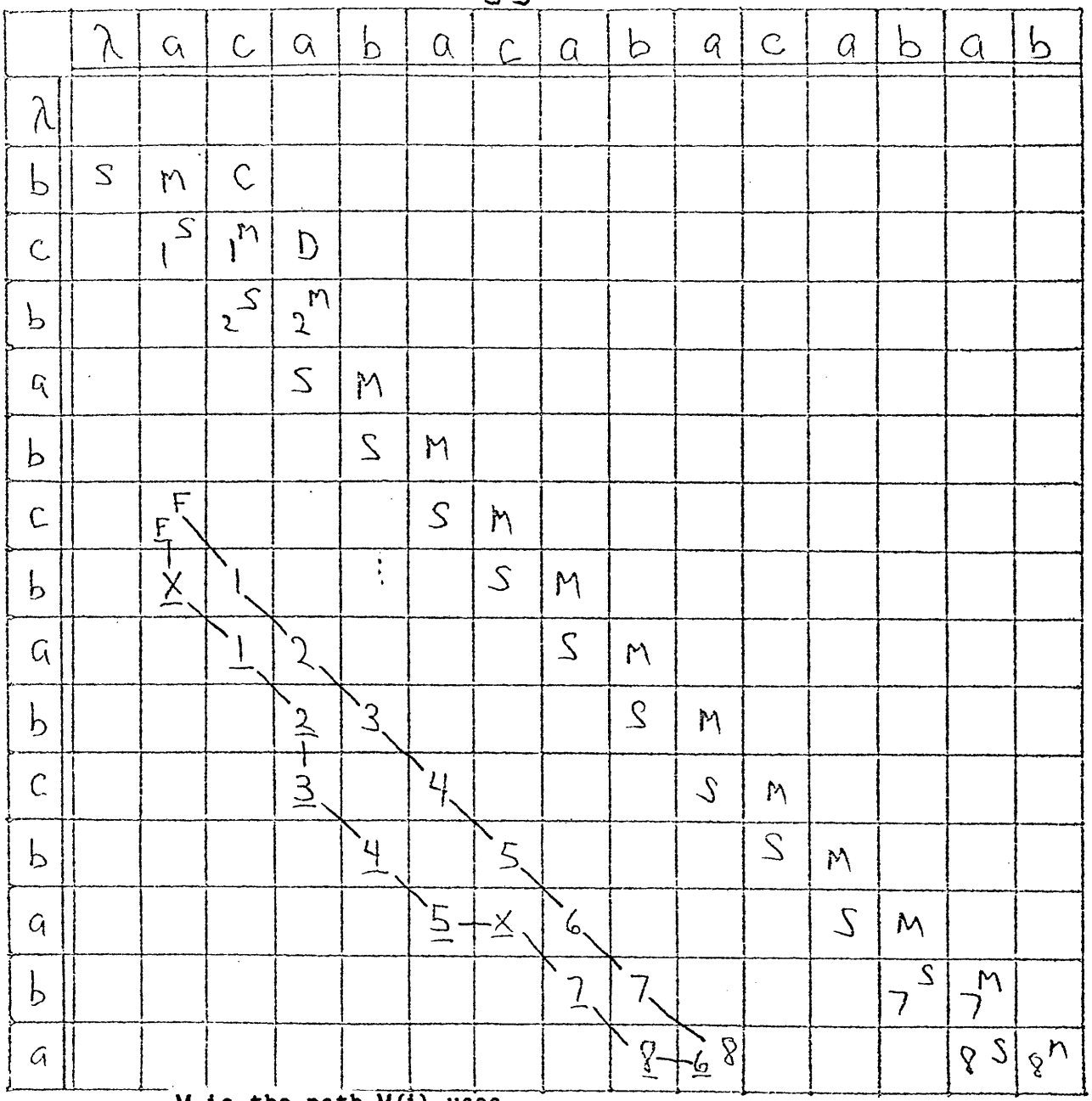


Figure 5 The State Transition Diagram for all Possible U Paths

(a) Add the total cost for each insert or delete just before an U_E section to the cost of the corresponding U_E section.

(b) Add 12 units of the cost for each insert or delete just before an U_O section to the cost of the corresponding U_O section. Add 2 more units of the cost to the U_E section directly preceding it.

Given these transformations but no change in the cost of U, the cost of each U_E section has been increased by 22 and the cost of each U_O section has been increased by 12. The insert or delete directly preceding a U_E section is not mapped onto by T, so its cost can be reduced to 0. The costs for the rest of the insertions and deletion are now 6. If there is no replacement in a section to add the costs to



M is the path M(i) uses.
S is the path S(i) uses.
F is the starting point for paths U and T.
Operation n of T is mapped onto operation n of U.
The operations of U are underlined.
Numbers in the center show the parts of S and M used to compare the adjacent even and odd diagonals.
Note 1 and 2 map to C and D which translates to 1 and 2 on S.
The X's are the operations of U not mapped onto. Sections 1, 2, 3 and 6, 7, 8 are the two different types of E sections.
Section 4, 5 is an O section.

Figure 6 A Sample U-T Mapping

assume the costs in U have been reduced as if there was a section there.

Now compare the edit operation costs of T with their corresponding costs in U. There are three types of mappings.

(1) A mapping of T onto an insert or delete (U_I). This only occurs directly following U_E sections. They all have cost 6 (the maximum cost for replacements), so T is just as good as U.

(2) A mapping of T onto a section of U_E. Assume the first r replacements of T were mapped in the r replacements of U_E. Compare the subpaths $T_E((i,j), (i+r,j+r))$ and $U_E((i+1,j), (i+r+1,j+r))$. The mapping guarantees Lemma 9(ii) holds, so $C(T_E) \leq S(j+r)-S(j)+6$. By Lemma 10(ii) $C(U_E) \geq M(j+r)-M(j)$. Each section of U_E has 22 units added to its normal cost, so T_E is worse only if

$$C(T_E) > C(U_E)+22$$
$$S(j+r)-S(j)+6 > M(j+r)-M(j)+22$$
$$M(j)-S(j) > M(j+r)-S(j+r)+16$$

Use what is known about M(k)-S(k) (Lemma 11)--

$$20 > 8-\pi+16$$
$$0 > 4-\pi.$$

(3) A mapping of T onto a section of U_0. Here Lemma 8 applies and $C(T) \leq 12+ C(U_0)$. Each section of U_0 has been increased by 12, so T is not worse.

Replace each section U with a section T, that does not increase

the cost of R. Repeat the replacement process for all sections of R with $d(i) > p$ until $\max(D(R)) \leq p$. This new path is no worse than the old one, but by the inductive hypothesis if $\max(D(R)) \leq p-2$ then R is no better the path with $D(R) \leq 1$, or S(k). Therefore $S(k) = \delta_{k,k-1}$.

(iii) follows by symmetry. □

Now show there is no finite set containing all of the intervals of a matrix computing δ.

Theorem 5. Given the infinite strings A and B as generated, there does not exist a finite set W such that $\langle \delta_{i,i} - \delta_{i-1,i} \rangle \in W$.

PROOF. By Lemma 12 $\delta_{i,i} - \delta_{i-1,i} = M(i) - S(i)$. Assume W exists and $|W| = n$. Consider the prefixes of A and B such that $|A| = |B| \geq 3n$. Since equal multiples of π may exist only twice in M, there must be at least 1 pair (k, ℓ) such that $M(k) = p\pi$, $M(\ell) = m\pi$, $m \neq p$, and $M(k) - S(k) = M(\ell) - S(\ell)$. All values of S are integers, let $S(k) = x$ and $S(\ell) = y$ so

$$m\pi - x = p\pi - y$$

or $\pi = (x-y)/(m-p)$. Since m, p, x, y are all integers and $m \neq p$ no such W exists. □

In this example all conditions but sparseness hold and the algorithm does not work, so sparseness is a necessary condition.

Corollary 2. Sparseness is a necessary condition for using the Algorithms Y Z.

PROOF. If not give it the example. The number of intervals |J| in the program is too large (O(|A|)), so there are too many m length sequences of J to enumerate. □

4. Longest Common Subsequence

Let U and V be strings. U is a subsequence of length n of V if there exists $1 \leq r_1 < \ldots < r_n \leq |U|$ such that $U_i = V_{r_i}$. U is the longest common subsequence of A and B if U is a subsequence of both A and B and there is no longer subsequence of both A and B.

To compute it define γ over the integers as

$R_{a,b} = 0$ if $a=b \in \Sigma$,

$= 2$ otherwise

$D = I = 1$.

Now $\delta = |A|+|B|-2|U|$, or $|U| = (|A|+|B|-\delta))/2$. The domain is sparse, so if $|\Sigma|$ is finite, |U| can be computed in time $O(|A| * |B| / \log(\max(|A|, |B|)))$, using Algorithm Y Z. The cost function in this section is due to [5].

5. Conclusion

An algorithm for computing the shortest edit distance between two strings of length n in time $O(n^2/\log n)$ was presented. The alphabet

must be finite and the domain for the cost function must be sparse. If the domain is not sparse the algorithm might not work. The problem remains to find a better algorithm for the finite alphabet case without the sparseness condition.

For the infinite alphabet case Wong and Chandra [6] obtained $O(n^2)$ upper and lower bounds using a slightly restricted model of computation. Hirschberg, Aho and Ullman [1] obtained similar results for the longest common subsequence problem. Lowrance and Wagner [3] extended the results of [5] to include the operation of interchanging adjacent characters. They developed an $O(n^2)$ algorithm for the extended problem under some cost constraints.

The question of the complexity of the shortest edit distance problem for finite alphabets remains. The best lower bound is $O(n)$ [6], the upper bound is $O(n^2)$ or $O(n^2/\log n)$. This gap seems too large. The work done so far seems to indicate the $O(n)$ lower bound can be raised.

6. References

[1] Aho, A.V., D.S. Hirschberg and J.D. Ullman. "Bounds on the Complexity of the Longest Common Subsequence Problem". JACM 23:1 (January 1976), 1-12.

[2] Arlazarov, V.L., E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. "On Economic Construction of the Transitive Closure of a Directed Graph", Dokl. Akad. Nauk SSSR 194 (1970), 487-488 (in Russian). English translation in Soviet Math Dokl. 11:5, 1209-1210.

[3] Lowrance R. and R.A. Wagner. "An Extension of the String to String Correction Problem". JACM 22:2 (April 1975), 177-183.

[4] Paterson, M.S., Private discussions with M. Fischer (1975).

[5] Wagner, R.A. and M.J. Fischer. "The String to String Correction Problem." JACM, 21:1 (January 1974), pp168-173.

[6] Wong, C.K. and A.K. Chandra. "Bounds for the String Editing Problem". JACM 23:1 (January 1976), 13-16.

Part II -- Decision Graph Complexity

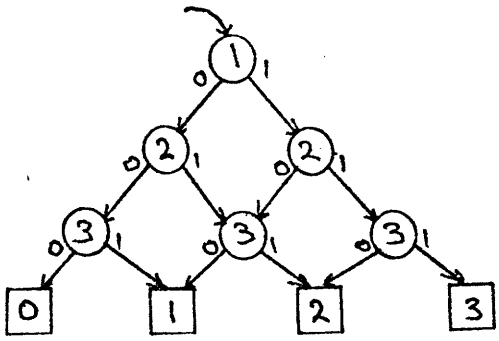
1. Introduction

Decision graphs were developed by C.Y. Lee to compute switching functions [6]. This paper shows how they are useful for studying space complexity problems. In the first section of the paper the decision-graph model is presented and related to the space complexity of Turing machine (TM) computations. The second section looks at the space complexity of real-time programs. The third section compares decision-graph complexity with boolean formula and contact network complexity. The last section looks at the complexity of decision graphs for arbitrary boolean functions.

1.1 Basic Definitions

Decision graphs are like "random-access" finite-state machines where only one input character can be looked at in any state. Let Σ be an alphabet and R be some arbitrary set. Define the <u>decision graph</u> A computing $f:\Sigma^n \to R$ as a 9-tuple $\langle S, s, \lambda, \tau, F, \Omega, \Sigma, R, n \rangle$, where S is the set of states in A, s is the initial state, and F is the set of final states. For each intermediary state the function $\lambda: (S-F) \to (1,\ldots,n)$ determines which input character is tested and $\tau: (S-F) \times \Sigma \to S$ describes the state-transition function. The computation starts in state s with input $X = x_1, \ldots, x_n$. A graph

computation is denoted by a sequence of states (a path) $s_0, \ldots, s_m$ where $s_0 = s$, $s_m \in F$, and $\forall i$, $1 \leq i \leq m$, $s_i = \tau(s_{i-1}, x_{\lambda(s_{i-1})})$. When the computation reaches state s_m the graph prints the output $\Omega(s_m)$. (Here Ω maps F into R.) The decision graph complexity of any finite function f, denoted by C(f), is the minimum number of states required by any decision graph to compute f. A graph which solves a problem using this minimum number of states is an optimal graph. In all of the following assume Σ is (0,1). Figure 1 gives an example of a decision graph. The following theorem shows how



The numbers in the circles denote the bit tested at that state.
The numbers in the boxes are the outputs.

Figure 1. A real time bit counting decision graph for n=3.

decision graphs model TM space complexity.

Theorem 1. If f has complexity C(f), then a TM with a read-only input tape and a read-write work tape using s states must use at least k bits of storage, where $C(f) \leq 2^k ksn$.

Proof. Assume there exists a TM M such that the theorem does not hold for inputs of length n. An instantaneous description (id) for M consists of $\langle q, W, w, h \rangle$ where q is the state M is in, W is the work-tape contents, w is the work-tape head position and h is the input head position. Write down all achievable id's for inputs of length n, and construct the decision graph A from the id's. Let each id represent a state. A particular id gets transformed into at most two other id's determined (since the tape contents is already known) entirely by an input bit. Let A use the same transitions to go to the corresponding states. Whenever a TM halts the work tape gives the answer. Make the id's for halt states decision graph output states and let the graph output the work-tape contents in that id. There are no more than $2^k ksn$ id's, so A computes f with less than C(f) states, a contradiction. □

2. Real Time Complexity

A decision graph computes f in real time if no input bit is ever looked at twice during a particular computation -- like the graph in Figure 1. Most programs with input $(0,1)^n$ try to look at each bit only once, and remember it. This is real-time computation.

It is a very restrictive model, so it is easy to show algorithms are optimal. A label, $\ell \in (0,1,*)^n$, will be used to describe the progress of a real-time computation. A '*' in the label will represent a bit not yet asked about, a '0' or '1' will represent a bit already asked about.

2.1 Optimal Real-Time Graphs

The real-time constraint can be circumvented. A function f on n bits that needs to look at each bit three times to be computed efficiently could be computed as a function g on 3n bits. Just define g so that f(X)=g(XXX) for all inputs X. To prevent this, call a problem real time (different from a real-time algorithm) only if for all possible labels the function on the remaining bits is trivial (they all give the same answer) or there exists some assignment of 0's and 1's to the *'s so that all input bits must be looked at. Most interesting real-time algorithms compute real-time problems, and it is usually easy to show algorithms for real-time problems are optimal. Associate each state of a decision graph with the labels of computations occuring there. Two labels are incompatible if either (1) the *'s are not in the same positions, or (2) they do not have the same output for identical assignments to the *'s. Incompatible labels cannot be associated with the same state in real-time graphs computing real-time functions.

Theorem 2. Let A be a decision graph computing a real-time function f. If any incompatible labels are associated with the same state, then A is not a real-time decision graph.

Proof. Assume A computes f in real time and two incompatible labels ℓ_1 and ℓ_2 are associated with the same state in the graph. There are two cases.

Case 1. Condition (1) is violated. Consider the assignment to the *'s in ℓ_1 when the whole input must be looked at. Apply the same assignment to ℓ_2, it must follow the same path. Therefore if a position in ℓ_1 has a '*' then the corresponding position of ℓ_2 must have a '*'. By a similar argument, every position of ℓ_2 with a '*' guarantees a '*' in ℓ_1; so the *'s are in the same positions for both labels.

Case 2. Condition (2) is violated. Follow the path taken by ℓ_1 when the answer is different. A real-time decision graph would follow the same path for ℓ_2 and give the same (wrong) answer. □

Theorem 2 can be used to give a lower bound on the number of states needed to compute the sum of the input bits.

Corollary 1. Any real time graph counting the number of inputs set to one uses at least $(n+2)(n+1)/2$ states.

Proof. This is trivially a real-time problem. After k bits have been asked about there must be k+1 states one for each possible

answer 0, 1, ..., k. By Theorem 2 the decision graph must have $\sum_{0\le k\le n} k+1 = (n+2)(n+1)/2$ states. □

Other optimal real-time decision graphs Theorem 2 describes include computing the sum of the inputs mod p, and $f(X) = "\Sigma x_i = m"$.

Corollary 2. Any real-time decision graph computing the sum of the bits on mod p ($n \ge p$) uses $p(p+1)/2 + (n+1-p)(p+1)$ states.

Corollary 3. Any real time decision graph computing $f(X) = "\Sigma x_i = m"$ uses at least $m(n-m+1)+2$ states.

Proof. The proofs for both corollaries are left to the reader. □

2.2 Real-Time Programs Are Not Always Optimal

The definition for real-time computation is very restrictive, so it is not surprising that real-time decision graphs are not always optimal. Michael Fredman [1] exhibited an algorithm computing $f(X) = "\Sigma x_i = n"$ using $O(n\log^2 n/\log\log n)$ states.

Theorem 3. (Chinese Remainder Theorem) [5]. Let $m_1, m_2, \ldots, m_r$ be positive integers which are relatively prime in pairs, i.e.,

$$\gcd(m_j, m_k) = 1 \text{ when } j \ne k.$$

Let $m = m_1 m_2 \ldots m_r$, and let $a, u_1, u_2, \ldots, u_r$ be integers. Then there is exactly one integer u which satisfies the conditions

$$a \le u < a+m, \text{ and } u \equiv u_j \text{ (modulo } m_j) \text{ for } 1 \le j \le r.$$

Theorem 4. (Prime Number Theorem) [2]. Let r be some real number.

(a). There are $O(r/\log r)$ prime numbers between 2 and r.

(b). The product of all primes less than r is $O(e^r)$.

The Chinese Remainder Theorem and the Prime Number Theorems can be used to find a quick algorithm to compute f(X).

Theorem 5. $f(X) = "\Sigma x_i = m"$ can be computed with a decision graph using $O(n\log^2 n/\log\log n)$ states.

Proof. Take the first k primes whose product is greater than n and let ℓ be the largest of the k primes. Precompute the residues $m_1, \ldots, m_k$ such that $m \pmod{p_i} = m_i$. Make the decision graph compute the residues of the n input bits mod $p_1, \ldots, p_k$ respectively. If the ith residue is not m_i for any i there are not exactly m bits on. If all k residues are correct there are exactly m bits on (Chinese Remainder Theorem). Counting mod p with the real-time counting mod p graph for each of the k primes uses $O(nk\ell)$ states.

The Prime Number Theorem gives minimum values for k and ℓ. The largest prime needed is $O(\log n)$. There are $O(\log n/\log\log n)$ primes between 2 and log n. The decision graph uses $O(n\log^2 n/\log\log n)$ states. □

Theorem 5 shows looking at bits more than once can be useful. However for some problems real-time graphs seem to be optimal, in

particular the bit counting problem.

3. Using Decision Graphs for Other Models

This section compares decision graphs with boolean formulas and contact networks. Turing-machine space complexity is an upper bound on decision graph complexity (Theorem 1).

3.1 Modelling Boolean Formulas

Decision graphs model almost all boolean formulas easily.

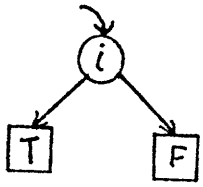
Theorem 6. Any boolean formula E without $\oplus$ or $\equiv$ with n literals can be computed with a decision graph using n+2 states.

Proof. By induction of the connectives in the formula. First rewrite the formula in terms of $\wedge$ and $\neg$. Since $\oplus$ and $\equiv$ are not allowed, this does not increase the number of literals. Let T(E) be the number of connectives in the formula E, and let N(E) be the number of literals in the formula E.

If T(E)=0 then E is just a literal, say x_i. Use a graph like Figure 2. There is one literal, and 1+2=3 states.

Assume the theorem holds for all formulas F such that k>T(F), now show it holds for k=T(F). There are 2 cases.

Case 1. $F = \neg G$, $k=T(G)+1$ and $N(F)=N(G)=m+2$. Compute G using m+2 states. Relabel the true output of G false, and vice versa. The graph now computes F using m+2 states.



The i in the circle is the bit tested.
T and F are the true and false labels.

Figure 2. A decision graph building block.

Case 2. $F = G \wedge H$, $k = T(G) + T(H) + 1$, and $N(F) = N(G) + N(H) - 2$. Compute G, then send the true output of G to the start state of H. Leave the true output of H alone, and send the false output of G to the false output of H. The output states of G are no longer used, and no new states are introduced, so F takes $N(G) + N(H) - 2$ states to compute. □

3.2 Modelling Contact Networks

The number of edges in a directed-contact network and the number of states in a non-deterministic decision graph are asymptotically equal within a constant factor. There is only one difference between non-deterministic decision graphs (NDDG's) and deterministic decision graphs -- the state-transition function τ maps into sets of possible states rather than just one state. A NDDG graph A outputs $r \in R$ for input X if and only if there exists a

computation on X with output 'r'. Notice a NDDG may have several outputs for the same input.

A contact network [3] is a 6-tuple $\langle G, g, R, \Sigma, \Omega, n \rangle$ where G is an undirected graph, g is the designated start node, R is the set of output markings for the nodes, Σ is the input alphabet, Ω is the set of marked nodes in G, and n is the number of inputs. Each arc of G is labeled with a statement "$x_i = s$", $s \in \Sigma$. A contact network outputs r on input X if there exists a path from g to a node marked 'r' and the statement on each arc of the path is true. Figure 3 is a contact network. A contact network is directed if the

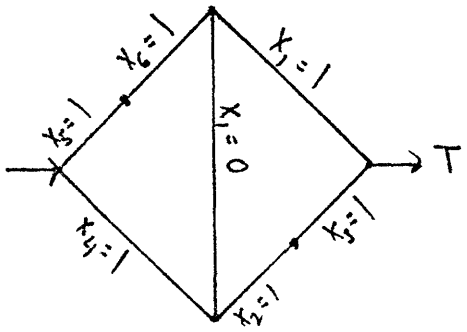


Figure 3. A sample contact network.

graph G is a directed graph (then any computation must follow a directed path). Non-deterministic decision graphs can model contact

networks using twice the number of states as edges in G.

Theorem 7. Let C be a contact network with E edges computing f, there exists a non-deterministic decision graph with 2E states computing f.

Proof. Let G=(E,V) be the graph describing C. First discard any portions of G not connected with g, the start vertex. Use one state for each vertex of G as a vertex state. Each vertex state of A will initiate the decision graph's simulation of paths in G through that vertex. For each non-output vertex repeat the following process. List all edges going out of it with their respective conditions. For each edge (except the first one) add a state. For each state s (including the vertex state) test the bit the corresponding edge tests, then define a transition for s in τ corresponding to the condition on the edge being true to the vertex state corresponding to the vertex the edge goes to, and define two more transitions for s in τ -- one with the same condition, and one with the condition does not hold -- to the state testing the next edge (direct the last state to the vertex state of the present vertex). Every path possible in C is now possible in A. Each vertex has at least one edge coming into it and each edge is represented by 2 states so $C(A) \leq 2E$. □

Undirected contact networks cannot model decision graphs easily because spurious paths may be introduced, however decision graphs may be simulated with directed contact networks using as many edges as lines coming out of the decision graph states.

Theorem 8. Let A be a decision graph with m lines leading out of its states, there exists a directed contact network C computing A with m edges.

Proof. Let the graph G=(E,V) for C be identical to the graph for A. Label each edge of G with the condition tested in the state the edge comes out of. Mark each vertex of C corresponding to an output state of A with the same output of A. Define the output of C in the same manner as the output of A. □

Notice Theorem 8 is not restricted to deterministic decision graphs.

4. Decision-Graph Complexity for Arbitrary Functions

This section describes three problems dealing with the decision graph complexity of arbitrary functions. The first result describes the complexity of arbitrary predicates on n bits. The second provides an upper bound for predicates on the sum of the bits. The last result gives a better than exponential, but not polynomial method for simulating non-deterministic decision graphs

with deterministic decision graphs.

4.1 Simulating Arbitrary Boolean Functions

Lupanov[7] described upper and lower bounds on the complexity of circuits for simulating boolean functions. The same ideas can be used to determine the decision graph complexity for arbitrary boolean functions. (The results in this section were independently discovered in [6].)

First determine the lower bound. There are $n^k k^{2k}$ different decision graphs with k nodes. Even if each one of them computes a different one of the 2^{2^n} possible boolean functions on n variables, there must be $2^{n-1}/n$ states to compute all possible functions.

Theorem 9. There are boolean functions on n variables requiring $O(2^{n-1}/n)$ states to compute.

Proof. Consider a decision graph with k nodes. Each state can test any one of n bits then branch to any 2 of the k states. This results in nk^2 possible configurations for each state, or $n^k k^{2k}$ different graphs with k states.

There are 2^{2^n} different boolean functions on n bits, so even if every different graph computed a different function, $2^{2^n} \geq n^k k^{2k}$ would have to hold for k. Let $k = 2^{n-1}/n$.

$2^n \geq 2k\log k + k\log n$, substituting for k

$$2^n \geq \frac{2 \cdot 2^{n-1} \log 2^{n-1}}{n} - \frac{2 \cdot 2^{n-1} \log n}{n} + \frac{2^{n-1} \log n}{n}.$$

$$2 \geq \frac{2 \cdot (n-1)}{n} - \frac{\log n}{n}. \quad \square$$

Decision graphs can simulate arbitrary boolean functions with n variables using $O(2^n/n)$ states using a two stage construction. The first stage separates the inputs into the 2^k possible arrangements of the first k bits, then each state is fed into a graph computing all $2^{2^{n-k}}$ functions on the remaining n-k bits.

Lemma 1. All functions of ℓ bits can be computed with the same network of 2^{2^ℓ} states.

Proof. Proof by induction on the number of bits ℓ. If $\ell=0$ there are 2 possible functions, constant true or false.

Assume the lemma holds for ℓ-1 bits, show it holds for ℓ bits.

There are 2^{2^ℓ} functions on ℓ bits. However, $2^{2^{\ell-1}}$ of them do not depend on the first bit. Simulate them using the $2^{2^{\ell-1}}$ states of the ℓ-1st stage of the induction. Use one state for each of the $2^{2^\ell} - 2^{2^{\ell-1}}$ remaining functions. The outputs for these states are functions on ℓ-1 bits, so attach them to the appropriate state in the $2^{2^{\ell-1}}$ section. The decision graph uses 2^{2^ℓ} states. $\square$

If the first n - log(n-log n) bits are split, and all functions on the remaining log(n-log n) bits are simulated, then

all boolean functions can be computed using $O(2^n/(n-\log n))$ states.

Theorem 10. All boolean functions on n variables can be computed using a decision graph with $O(2^n/n)$ states.

Proof. The computation is split into 2 stages.

Stage 1. Decode each n-k length input prefix into the 2^{n-k} possible configurations using a binary tree with $2^{n-k}-1$ states.

Stage 2. Construct the complete graph for k bits. Direct each output of Stage 1 into the state which describes its function on the last k bits. This uses 2^{2^k} states (Lemma 1).

The decision graph uses $2^{n-k} + 2^{2^k}$ states. Let $k = \log (n-\log n)$, the decision graph uses $2^n/(n-\log n) + 2^n/n = O(2^n/n)$ states. □

Notice the decision graph computes the functions in real time. It uses about 4 times the states required by the lower bound. The lower bound counts many meaningless programs, like the one which never leaves the start state.

The number of states needed to compute some fraction of the possible functions can be calculated using the method in Theorem 9.

Theorem 11. [6] Given any ϵ, $0 < \epsilon < 1$, a fraction $1-2^{-\epsilon 2^n}$ of boolean functions on n variables will need at least $2^n(2n)^{-1}(1-\epsilon)$ states to be computed.

4.2 Computing Arbitrary Predicates on Bit Sums

The same idea, computing in stages, works for computing arbitrary predicates on sums of bits (such as "are there at least k bits on?"). Most graphs use $O(n^2)$ states for these problems. Michael Fredman noticed the Chinese Remainder Theorem makes these problems solvable using $O(n^2/\log n)$ states. The first stage counts the bits mod $2n/\log n$ and the second counts them mod $(\log n)/2$. There are relatively few ways of assigning outputs to the $(\log n)/2$ answers, so all possible functions can be computed using $O(n^2/\log n)$ states.

Theorem 12. Let f be a boolean predicate on the sum of the input bits, then f can be computed with a decision graph using $O(n^2/\log n)$ states.

Proof. The computation is split into two stages. Pick two relatively prime numbers roughly $2n/\log n$ and $(\log n)/2$. Count the bits mod $2n/\log n$ using $O(n^2/\log n)$ states. Then count the bits mod $(\log n)/2$ for each answer. Each sum mod $(\log n)/2$ has $(\log n)/2$ outputs and there are $2^{\log n/2} = n^{1/2}$ ways of assigning values to the outputs. Compute all possible second stages using $O(n^{3/2}\log n)$ states. Attach each of the first stage answers to the correct second stage. This requires just $O(n^2/\log n) + O(n^{3/2}/\log n) = O(n^2/\log n)$ states. □

4.3 Simulating Non-Deterministic Decision Graphs

Jones and Laaser [4] introduce log space-complete for P problems. They showed finding a path between 2 points in a graph is a log space-complete problem for P. Savitch [8] related non-deterministic space s with deterministic space s^2. The same result applies to decision graphs.

Theorem 13. A non-deterministic decision graph A with c states can be simulated with a deterministic decision graph B using $O(c^{1+\log c})$ states.

Proof. Assume there is a unique state for each output of A. All outputs of A can be reduced to the existence of length c paths to specific output states. B computes what A computes if B can decide what A can compute in c steps. Number the states of A in increasing order. Let s be the start state of A, and t be an output state for A -- B computes A goes from s to t in no more than c steps, abbreviated (s, t, c). Let f(a, b) ask the question needed to go from state a to state b in A, then Find (B) computes A.

```
Find (s, t, c):
  begin
  if c=1 then return f(s, t);
  for u = 1 to c
    do if Find (s, u, ⌊c/2⌋) ∧ Find (u, t, ⌈c/2⌉) then return
true;
  return false;
  end Find;
```

Let F(i) be the number of steps Find uses to find length i paths. F(i) = 2cF(i/2), and F(1) = 1. Therefore B uses $O(c^{1+\log c})$ states. □

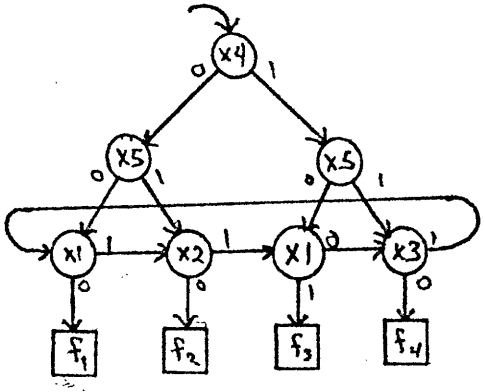
4.4 Circuits in Optimal Graphs

A decision graph has a circuit if there is a series of arcs leading from one state back to itself. R.A. Short [9] showed the graph in Figure 4 was optimal. He used an exhaustive case analysis to prove it.

Theorem 14. [9] There exists an optimal decision graph with a circuit.

5. Conclusion

Decision graphs provide a precise method for calculating the storage necessary to compute functions. Optimal decision graphs give lower bounds on the space complexity. Real time constraints are tight, and showing non-trivial optimal real time graphs optimal without using the real time constraint is difficult. However, real time bit counting seems to be optimal. The most important problem related to this model is whether deterministic decision graph complexity is polynomially related to non-deterministic decision graph complexity.



The numbers in circles are the bits to test.
The letters in boxes are the outputs.

Figure 4. An Optimal Decision Graph with a Circuit.

6. References

[1] Fredman, M., Private Discussions with Ron Rivest (1975).

[2] Hardy, G.H. and E.M. Wright, An Introduction to the Theory of Numbers, Oxford at the Clarendon Press, (1960), Oxford.

[3] Harrison, M.A., Introduction to Switching and Automata Theory, McGraw-Hill Book Company, (1965), New York.

[4] Jones, N.D., and W.T. Laaser, "Complete Problems for Deterministic Polynomial Time", 6th SIGACT Proceedings, (1974), 40-46.

[5] Knuth, D.E., The Art of Computer Programming, Vol. 2 Seminumerical Algorithms, Addison-Wesley Publishing Company, (1969), Reading, Massachusetts.

[6] Lee, C.Y., "Representation of Switching Functions by Binary Decision Programs", Bell Sys. Tech. J., 38(1959), 985-999.

[7] Lupanov, O.B., "On the Complexity of the Realization of Formulas of the Functions of an Algebra of Logic", Probl. Kibernet. 3(1960), 61-80.

[8] Savitch, W.J., "Relationships Between Nondeterministic and Deterministic Tape Complexities", Journal of Computer and System Sciences, 4:2 (April 1970), 177-192.

[9] Short, R.A., "A Theory of Relations between Sequential and Combinatorial Realizations of Switching Functions", Stanford Electronics Laboratories, Menlo Park, Calif., Tech. Report. 098-1, 1960.