APPROXIMATE PATTERN MATCHING IN A PATTERN DATABASE SYSTEM

Larry S. Davis*
Nicholas Roussopoulos*

TR-99 June, 1979

*Computer Sciences Department, The University of Texas at Austin, Austin, Texas 78712.

This research was supported in part by the National Science Foundation under Grant ENG 74-04986.

1. Introduction

Pyramidal representations for images have, over the last few years, received a considerable amount of attention as a tool for computer vision. For example, the VISIONS system developed at the University of Massachusetts (see Hanson and Riseman [1]) does almost all of its low-level vision processing in a pyramid. The pyramid, in this case, leads to great computational savings since an analysis at a high level of the pyramid (which contains only a coarse approximation to the underlying image) can serve as a plan for analyzing lower levels in the pyramid (see also Kelly [2]).

Much of the early, fundamental analysis of pyramids, or regular picture decompositions, was done by Klinger and Dyer [3] and by Tanimoto and Pavlidis [4]. Recently, Nakamura and Dyer [5] have investigated the power of pyramids as control structures, and Hunter [6] has discussed complexity issues for certain binary picture operations using pyramid representations.

This paper is also concerned with the complexity of picture operations using pyramid-like data structures. In particular, we are interested in discovering that two binary pictures (normalized for size, rotation, and position) "almost" match. Our motivation for studying this problem is the design of error-tolerant pattern database systems. Too often, in pattern analysis, matching algorithms are proposed without regard to the global organization of the representations of the models they are to match. Consequently, the algorithms are only practical for small databases.

This paper will discuss the matching problem along with the design

of a pattern database system. Section 2 contains definitions. Section 3 describes both depth-first and breadth-first approximate matching algorithms. Section 4 contains a probabilistic analysis of approximate matching both with and without pyramids. Section 5 describes the organization of the pattern database. Finally, Section 6 contains conclusions.

2. Definitions

Definition: A binary image, I, is a $2^n \times 2^n$ array of 0's and 1's.

Given a binary image, or simply a picture, I, we will refer to $I(i,j)$ as the $(i,j)^{th}$ position of I. We can then define what it means for two images to approximately match.

Definition: Let I and I' be two binary images. Then we say that I and I' match within k if

$$\sum_i \sum_j |I(i,j) - I'(i,j)| \leq k .$$

Two images match within k if they disagree in at most k positions.

The pyramid representation used in the next section differs somewhat from the standard one ordinarily used for representing binary images (called a quad tree). Ordinarily, a node in a quad tree is labeled with a 1 when at least one of its descendants is labeled with a 1, i.e., the label at a node is the maximum label of its sons. In contrast to this, we will define a sum-quad tree (or SQT) where the label of a node is the sum of the labels of its sons. If n is a node in an SQT, then $v(n)$ is the value marking node n.

The lowest level of the SQT corresponds to the individual pixels and is the nth level (i.e., node m at level n implies $v(m) = 0$ or $v(m) = 1$). Nodes at the jth level of the SQT correspond to sums of $2^{n-j} \times 2^{n-j}$ picture points, so that the 0th level contains the number of 1's in the image. An SQT can be compactly represented by removing the descendants of any

level j node, m, with either

1) $v(m) = 0$, or

2) $v(m) = 2^{n-j} \times 2^{n-j}$

since in either case, the values of all the pixels in the $2^{n-j} \times 2^{n-j}$ neighborhood underlying m are known. If this is done we say that the SQT is in compact form.

Storing images as SQT's may require more storage than storing just the original picture. A node at level j must be capable of encoding a number from 0 to $2^{2(n-j)}$, and so requires 2n-2j+1 bits of storage. There are at most $2^j \times 2^j$ nodes at the jth level, so that the total amount of image storage, PS, for an n level tree is

$$PS(n) = \sum_{j=0}^{n} 2^{2j}(2n-2j+1) \qquad (1)$$

as opposed to 2^{2n} bits of storage for the original image. So, for example, for a 16×16 image (n=4) the original image requires 256 bits while the SQT requires $1\times9 + 4\times7 + 16\times5 + 64\times3 + 256\times1 = 565$ bits. The following two remarks are relevant:

1) Equation 1 does not take into account the savings achieved when a node, m, at level j has $v(m) = 0$ or $v(m) = 2^{2(n-j)}$, and

2) In the case where the SQT is stored in compact form, pointers must be maintained to move from one level in the tree to another. Note that if the SQT is not stored in compact form, then, since the size of any node is known, a priori, the SQT can be linearized and any node at any level could be addressed directly.

To illustrate this, suppose we store the SQT using a top-down, left-right raster scan of the SQT. That is, if S(k,i,j) represents the

$(i,j)^{th}$ node at level k, then the SQT would be stored sequentially as $S(1,1,1), S(2,1,1), S(2,1,2), \ldots, S(2,2,2), S(3,1,1), \ldots, S(n,2^n,2^n)$.

In this case, $S(k,i,j)$ can be found at the offset,

$$\sum_{k'=1}^{k-1} 2^{2k'} (2n-2k'+1) + (i-1)\, 2^k\, (2n-2k+1) + j(2n-2k+1)$$

from the base address of $S(1,1,1)$. The choice of whether to store an SQT in linked, compact form or directly depends on how many nodes are pruned from the SQT by the compaction operation.

3. Approximate Matching Using SQT's

This section will consider the problem of matching SQT's within k. In order to simplify the notation, we will consider the matching problem for binary strings rather than arrays. All of the results can be easily extended to arrays. Since a node in a pyramid representation for a string will only have two sons (which we call Left and Right), we will refer to these pyramids as sum-binary trees, or SBT's.

We start with the obvious procedure for matching two strings S and T within k and we let S(j) refer to the jth position of the string in its standard representation as a one level bit string.

```
Boolean Procedure
 STRINGMATCH (S,T,K,N)
    binary array S[1:N], T[1:N];
    integer K, N, NEXTBIT;
    integer MISMATCH;
    boolean MATCH;
  begin
       MISMATCH:=0;
       MATCH:=true;
       NEXTBIT:=1;
       while (NEXTBIT ≤ N) and MATCH=true) do
          begin
             MISMATCH:=MISMATCH + exclusive-or (S(NEXTBIT), T(NEXTBIT));
             if (MISMATCH > K) then MATCH:=false;
             NEXTBIT:=NEXTBIT + 1
          end
       return (MATCH)
  end.
```

Algorithm STRINGMATCH simply examines pairs of elements of S and T, incrementing a counter whenever a mismatch occurs. In the worst case, STRINGMATCH would have to examine all n pairs of bits from S and T.

The next procedure, DFSBT, performs a depth-first match using the

SBT's of the two strings, S and T. In the following, we let S[k,i] denote the ith node at the kth level of the SBT. Given any node m, we refer to L(m) and R(m), the left and right subtrees of m. The original string occupies positions S[n,1],S[n,2],...,S[2^n,n] in the SBT.

```
Boolean Procedure
 DFSBT (S,T,K,MISMATCH)
    SBT S,T;
    integer K, MISMATCH, LEFTMISMATCH, RIGHTMISMATCH;
  begin
       if K < 0 then return (false);
       MISMATCH:=|v(S)-v(T)|;
       if MISMATCH > K then return (false);
       if TIP(S) or TIP(T) then return (true);
       /*compute mismatch to left*/
       if DFSBT (L(S), L(T), K, LEFTMISMATCH)=false then return (false);
       /*compute mismatch to right*/
       if DFSBT (R(S), R(T), K-LEFTMISMATCH, RIGHTMISMATCH)=false
                 then return (false);
       MISMATCH:=LEFTMISMATCH + RIGHTMISMATCH;
       return (true);
  end.
```

Algorithm DFSBT works by first computing the mismatch of S and T in the left subtrees. If this mismatch is greater than k, then the procedure halts. If the mismatch is less than or equal to k, or either S or T is a tipnode, then DFSBT returns true. Otherwise, if m is the mismatch between L(S) and L(T), then R(S) and R(T) are compared to determine if their mismatch is within k-m. Figure 1 displays two applications of Algorithm DFSBT, one for a pair of strings which do match within k, and one for a pair which do not. In the diagrams representing the sequence of calls to DFSBT, downward arrows represent procedure calls and are labeled with a number representing the chronological order of the call; upward dashed arrows are labeled with results which are ordered pairs

(true/false, mismatch value). The nodes are marked with the names of the subtrees being matched and the allowable number of mismatches, e.g., $(S_{122}, T_{122}, 2)$ is a request to determine if the trees rooted at S_{122} and T_{122} match to within 2.

A shortcoming of algorithm DFSBT is that it can only take advantage of lower bound estimates on mismatch, i.e., at a node (S,T,k), if $|v(S)-v(T)| > k$, then we know that the trees rooted at S and T do not match within k, because

$$|v(S)-v(T)| \leq |v(L(S))-v(L(T))| + |v(R(S))-v(R(T))|.$$

However, we can also compute upper bounds on the mismatch of two SBT's, and these upper bounds can be used by a breadth-first matching procedure.

Let S and T be the roots of SBT's for strings of length n. Then, if M(S,T) is the true mismatch of S and T we have, from above,

$$M(S,T) \geq |v(S)-v(T)|.$$

However, it is also the case that

$$M(S,T) \leq \min\{v(S)+v(T),\ 2n-(v(S)+v(T))\}.$$

The following argument shows why this is true. First, it is clear that if $v(S)+v(T) < 2n-(v(S)+v(T))$, then $v(S)+v(T) < n$, i.e., there are fewer 1's than 0's in S and T combined. Let us suppose that this is the case. Then what ordering of v(S) 1's and n-v(S) 0's in S, and v(T) 1's and n-v(T) 0's in T gives the maximal mismatch of S with T? Obviously, the one that pairs the most 1's of S with 0's of T, and 1's of T with 0's of S. Now, assume without loss of generality that $v(S) \leq v(T)$. Then $v(S) \leq n/2$, since if $v(S) > n/2$, then $v(S) + v(T) > n$. Then the configuration

$$
\begin{array}{ll}
S & \overbrace{1\;1\;1\;\ldots\;1}^{v(S)}\;\ldots\;\overbrace{0\;0\;0\ldots\;0}^{n-v(S)} \\
T & \underbrace{0\;0\;0\;\ldots\;0\;\ldots\;0}_{n-v(T)\,>\,v(S)}\underbrace{1\;1\;1\;\ldots\;1}_{v(T)}
\end{array}
$$

yields the maximal number of mismatches. This is the configuration where all of the 1's of S are placed at the head and all of the 1's of T are placed at the tail. Clearly, the total number of mismatches is $v(S) + v(T)$. If $v(S) + v(T) \geq 2n-(v(S) + v(T))$, then we simply complement S and T, and the above argument shows that the mismatch is then $(n-v(S)) + (n-(v(T))$, the sum of the numbers of 1's in the complemented versions of S and T. (Note: complementing S and T obviously doesn't affect their match.)

The following algorithm, BFSBT, makes use of both the lower and upper bounds to solve the approximate matching problem. The main argument to procedure BFSBT is QUADSET, a set of records (QRECS) of the form (T_1, T_2, u, ℓ), where T_1 and T_2 are the roots of two SBT's and u and ℓ are upper and lower bounds on the match of T_1 to T_2. Clearly, if $u = \ell$, then we know exactly to what extent T_1 and T_2 match. Also note that if either T_1 or T_2 is a leaf (or tip), then it must be the case that $u = \ell$, so that BFSBT does not have to check explicitly for tip nodes.

Figure 2 contains two traces of the application of BFSBT to the patterns in Figure 1.

```
Boolean Procedure:  BFSBT (QUADSET, K)
   set of QRECS:  QUADSET, NEWSET;
   record QRECS
      begin
           T1:SBT;
           T2:SBT;
           UP: integer;
           LOW: integer;
      end;
   integer K, LOWERSUM, UPPERSUM;
   type QRECS Q1;
   type SBT LT1, LT2, RT1, RT2;
 begin
      LOWERSUM:=UPPERSUM:=0;
      for each Q1 in QUADSET do
         begin
              LOWERSUM:=LOWERSUM + LOW(Q1);
              UPPERSUM:=UPPERSUM + UP(Q1);
         end;
      if UPPERSUM ≤ K then return (true);
      if LOWERSUM > K then return (false);
      NEWSET:=∅;
      for each Q1 in QUADSET do
         if UP(Q1) = LOW(Q1) then NEWSET:=NEWSET U {Q1}
         else begin
              LT1:=L(T1(Q1));
              LT2:=L(T2(Q1));
              RT1:=R(T1(Q1));
              RT2:=R(T2(Q1));
              NEWSET:=NEWSET U {(LT1,LT2,UPBOUND(LT1,LT2),LOWBOUND(LT1,LT2))}
                             U {(RT1,RT2,UPBOUND(RT1,RT2),LOWBOUND(RT1,RT2))}
         end
      return (BFSBT (NEWSET,K))
 end.
```

4. A Probabilistic Analysis of STRINGMATCH and BFSBT

In this section we will present a probabilistic model for the number of comparisons that will be made by procedures STRINGMATCH and BFSBT in performing an approximate match of two strings. We will assume that strings are formed by randomly assinging a 1 to a string position with probability p, and randomly assigning a 0 to a string position with probability q = 1-p. Note that this model ignores the spatial dependence one ordinarily finds in real patterns.

Suppose we are given two such strings of length n, $b = b_1 b_2 \dots b_n$ and $d = d_1 d_2 \dots d_n$. Then

$$\begin{aligned} \text{prob}\,[b_i = d_i] &= \text{prob}\,[b_i = d_i = 1] + \text{prob}\,[b_i = d_i = 0] \\ &= p^2 + (1-p)^2 \\ &= 2p^2 - 2p + 1 \\ &= p' \end{aligned}$$

$$\begin{aligned} \text{prob}\,[b_i \neq d_i] &= \text{prob}\,[b_i{=}1, d_i{=}0] + \text{prob}\,[b_i{=}0, d_i{=}1] \\ &= 2p(1-p) \\ &= 2p - 2p^2 \\ &= (1-p') = q' \end{aligned}$$

Now, the probability that exactly k out of j bits from b and d don't match, prob [k mismatches|j tries] =

$$\binom{j}{k} q'^{k} p'^{(j-k)}$$

Suppose that b and d do *not* match within k. Then procedure STRINGMATCH will discover this when comparing b_j to d_j if

1) $b_1 \ldots b_{j-1}$ and $d_1 \ldots d_{j-1}$ mismatch in *exactly* k positions, and

2) $b_j \neq d_j$.

This happens with probability

$$\binom{j-1}{k} q'^{k} \; p'^{(j-k-1)} * \; q'$$

$$= \binom{j-1}{k} q'^{k+1} \; p'^{(j-k-1)}$$

If a mismatch is detected at the jth position, then j comparisons have been made, so that the expected computation (measured in bit comparisons) to discover that two strings which don't match within k fail to match is

$$W_s(k) = \sum_{j=k+1}^{n} j \binom{j-1}{k} q'^{k+1} \; p'^{(j-k-1)}$$

Now, the probability that two random strings *do* match within k is

$$p_m = \sum_{j=0}^{k} \binom{n}{j} q'^{j} \; p'^{(n-j)}$$

so that the total expected work of Procedure STRINGMATCH for matching two random strings within k is

$$W_s(k) = n \; p_m + W(k)(1-p_m).$$

We will now consider the amount of work that Procedure BFSBT performs in matching two pyramid representations. Let s_r be the random variable corresponding to the sum of r random positions of a string. Then

$$\text{prob}\ [s_r = m] = \binom{r}{m} p^m\ q^{(r-m)}$$

For large r, we can approximate the distribution of s_r by a normally distributed random variable with mean rp and variance rpq. Now, let Δ_r be the r.v. s_r-s_r. Then Δ_r is normally distributed with mean 0 and variance 2rpq. So, the probability of two strings of length r mismatching by at least k is

$$p_r(k) = \text{prob}\ [\ |\Delta_r| \geq k] = 2 * \text{prob}\ [\ \Delta_r \geq k]$$

Next, we can express the probability of two arbitrary strings matching within k at the ℓth level of the SBT representation, denoted $p_s(\ell,k)$, as

$$p_s(\ell,k) = \text{prob}\ [\ 2^{\ell-1} * \left|\Delta_{\frac{n}{2^{\ell-1}}}\right| < k] = p_{\frac{n}{2^{\ell-1}}}\ (k/2^{\ell-1})$$

since we have to add the mismatches of $2^{\ell-1}$ independent nodes in the SBT, each being the root of an SBT for a string of length $n/2^{\ell-1}$. If we assume that mismatches at adjacent levels of the tree are independent events, then the probability of first detecting a mismatch at level ℓ between two strings that don't match within k, denoted $p_f(\ell,k)$, is

$$p_f(\ell,k) = \left[\sum_{\ell'=1}^{\ell-1} (p_s(\ell',k))\right] (1-p_s(\ell,k))$$

The independence assumption will hold most closely at high levels of the pyramid where upper bound estimates on mismatch tend to be grossly overexaggerated.

Next, the number of comparisons required to compare ℓ levels of two SBT's (ignoring compaction) is

$$W_\ell = \sum_{j=1}^{\ell} 2^{j-1} = 2^{\ell-1},$$

so that the total amount of work required to recognize that two strings mismatch by k using the SBT representation is

$$W_{SBT}(k) = \sum_{\ell=1}^{\log_2 n+1} W_\ell \; p_f(\ell,k),$$

and the overall work required to match two strings is

$$W_{SBT}(k) = p_m 2^n + (1-p_m)\; W_{SBT}(k).$$

The natural question to ask is: under what conditions does one algorithm perform less work than the other? Notice that the probability of detecting failure using procedure STRINGMATCH at the jth bit position is independent of the length, n, of the input string (i.e., n does not appear in this formula). But the probability of failure at any level of the SBT is a function of n. At the top level of the pyramid, the mismatch of two random strings, Δ_n, is, we have seen, a random variable that is normally distributed with mean 0 and variance 2npq. As n increases, so does the variance of that random variable, so that the probability that $|\Delta_n| > k$ is an increasing function of n. The importance of this

observation is that for large n, mismatches will almost always be detected at or near the root of the SBT, so that matching using the SBT representation will be faster than using a simple string representation.

To illustrate this, Figure 3 lists the probability of failure as a function of string position j for procedure STRINGMATCH with $k=4$, 5, and 6 and $p=.5$. Figure 4 contains $p_f(\ell,k)$ for SBT's of strings of length 2^7, $\ell=1,2,3$ and $k=4$, 5, and 6. These figures indicate that the pyramid representation should lead to much faster approximate matching than the conventional string matching approach.

5. The Design of the Pattern Database

In this section we describe the organization of a pattern database, which consists of binary images, and of a matching algorithm. The pattern matching takes one of the following forms: (a) finds all images in the database which match a given input pattern within k; or (b) finds all images in the database which best match the input pattern.

Clearly, when the database is large, i.e., 300-500k bytes, a simple straightforward linear matching algorithm which matches the input pattern against the whole database is formidable in terms of execution time and I/O operations, assuming that most of the database will have to reside on secondary storage. For this reason efficient organization and matching algorithms are necessary. Section 5.1 describes the organization of the database and Section 5.2 describes the matching algorithms which are based on the pyramid representation presented in Section 2.

5.1 Storage Organization

The image database is organized around a hierarchical indexing scheme based on the SQT representation. There are n levels of indices and each level is stored as a table. Each entry in the index table of level i is a 3-tuple (t, p, q) where:

1) $t = \langle t_1, t_2, \ldots, t_{2^i} \rangle$ is the ith level of a subset of the database, and

2) p and q are pointers to the $i+1^{st}$ table. All of the entries from p to q in the $i+1^{st}$ level table are patterns having t as their i^{th} level.

The p and q pointers of the $(n-1)^{st}$ level point to the images in the

database, which can be thought of as the nth level in this organization. The images in the database are thus organized using a prefix coding scheme; i.e., if (t, p, q) is an ith level entry, then all patterns between p and q at level i+1 are identical up to and including level i. Figure 5 shows the indexing scheme for a set of 24 16-bit string images.

In the rest of this section we compute an upper-bound on the storage requirements of the above organization. The analysis is based on a string image database but a similar analysis for two-dimensional images is straightforward.

Let 2^n be the number of bits in each image and let N be the total number of images stored in the database. The maximum storage required for the index table of level i, $0 \leq i \leq n-1$ is

$$\min\left\{N,(2^n/2^i)^{2^i}\right\}(2^i(n-i+1)+2p)$$

where $2^n/2^i$ is the largest possible integer appearing in an entry of the index table of level i, $(2^n/2^i)^{2^i}$ is the maximum possible number of combinations that can be entered in the ith level index table which clearly cannot exceed N, $n-i+1$ is the number of bits required to store each value of the 2^i-tuple entry, and p is the number of bits per pointer. For simplicity here, we assume that all pointers have the same size which is determined by the necessary number of bits p to represent the pointers of the nth index table. Since the size of the index tables is non-decreasing, $p = \lceil \log_2 N \rceil$ would be sufficient to represent any pointer at any level.

The total storage requirements for the index tables are

$$TSR = \sum_{i=0}^{n-1} \min\left\{N, (2^n/2^i)^{2^i}\right\} (2^i(n-i+1) + 2p)$$

$$= \sum_{i=0}^{n-1} \min\left\{N, (2^{n-i})^{2^i}\right\} (2^i(n-i+1) + 2p)$$

TSR is the maximum storage for any database of 2^n bit images. For example, if the database contains all the 2^{16} 16-bit strings, TSM is a little less than 30,000 bits assuming a 16-bit pointer. This is only a small fraction of the total of over one million bits required to store all images.

5.2 The Matching Algorithms

The first algorithm, FINDALL, described below, matches a given pattern against the database and finds all images that match the pattern with k or less mismatches at the pixel level. It uses the lower bound to reject images that exceed the maximum allowed mismatch k. It returns a list of pointers to the matching images.

The algorithm consists of three loops. The outmost loop is over all index tables. When the algorithm starts examining the ith index table, CLIST contains a list of pointers to those entries of this i^{th} table whose lower bound, LOWSUM, did not exceed k at level i-1. The second loop goes over all elements of CLIST and for each of them, LOWSUM is computed (through the innermost loop). If LOWSUM does not exceed k, the p^{i+1} and q^{i+1} pointers of the ith table are used to construct the LIST of entries of the $i+1^{st}$ level table. At the beginning, CLIST is initialized to every entry of the first level index table. The algorithm terminates after constructing the final LIST which consists of pointers to the images that

match within k. If CLIST ever becomes empty at some level, then no image in the database matches the PATTERN within k.

Note that TABLE [I, J, L] in the algorithm refers to the Lth column of the Jth row of the Ith index table, and PATTERN [I, J] refers to the Jth component of the Ith level in the pyramid representation of PATTERN.

```
Procedure FINDALL (PATTERN, K, LIST)
LIST, CLIST = list of integers
integer K, LOWSUM, P, Q, LEVEL, I, J, L, N, NO.OF.ENTRIES.IN.TABLE.1
begin
  /*initialize LIST to every entry of index table 1*/
  for I:=1 to NO.OF.ENTRIES.IN.TABLE.1 do
      LIST:=APPEND (LIST, I)
  /*outmost loop, over all index tables*/
  for LEVEL:=0 to N do
      begin
        /*CLIST becomes the LIST constructed at LEVEL-1
          and LIST is initialized to NIL*/
        CLIST:=LIST
        LIST:=NIL
        /*loop over CLIST and compute the LOWSUM for each element*/
        for each element L in CLIST do
            begin
              LOWSUM:=0
              for I:=1 to 2**LEVEL do
                  LOWSUM:=LOWSUM + ABS(PATTERN [LEVEL, I ] - TABLE [LEVEL, L, I])
                  if LOWSUM ≤ K then do
                  /*this constructs the LIST for the next level using
                    the P and Q pointers of the index tables*/
                    begin
                      P:=TABLE [LEVEL, L, 2**LEVEL + 1]
                      Q:=TABLE [LEVEL, L, 2**LEVEL + 2]
                      for J:=P to Q do
                          LIST:=APPEND (LIST, J)
                    end
            end
      end
end.
```

It is worth noting here that the algorithm does not make use of the upper bound. Whenever the upper bound becomes less than or equal to k, then direct pointers to the images can be used and put immediately into

the final LIST rather than into the LIST of pointers to the next level. This, in some cases, will eliminate unnecessary computation at subsequent levels. To achieve this saving, we have to incorporate the direct pointers to our index tables at some additional space cost. However, since the additional space may be very large, this is only done for the entries whose p^{i+1} and q^{i+1} pointers are equal at some level i. In this case one of the two pointers may be used as a direct pointer to the unique image in the database.

The second algorithm, FINDBEST, finds the image (or images) which has the smallest number of mismatches when matched against a given pattern. It makes use of both the lower bound LOWSUM and upper bound UPSUM to reject any candidate image whose LOWSUM becomes greater than the UPSUM of any other candidate.

FINDBEST is very similar to the FINDALL algorithm. The main difference is that k is assigned to the lowest UPSUM of each level. Note that since CLIST is examined only once at each level, if a lower UPSUM is discovered for a later element of CLIST which would otherwise reject an earlier element of CLIST, the constructed LIST may contain entries which have LOWSUM greater than the late UPSUM. Those entries, however, would be rejected at the next level because LOWSUM is a non-decreasing quantity.

```
Procedure FINDBEST (PATTERN, LIST)
LIST, CLIST = list of integers
integer K, LOWSUM, UPSUM, P, Q, LEVEL, I, J, L, N, NO.OF.ENTRIES.IN.TABLE.1
begin
  /*initialize LIST to every entry of index table 1*/
  for I:=1 to NO.OF.ENTRIES.IN.TABLE.1 do
      LIST:=APPEND (LIST, I)
  /*initialize K to a large number*/
      K:=2**N
  /*outmost loop over all index tables*/
  for LEVEL:=0 to N do
      begin
        /*CLIST becomes the LIST constructed at LEVEL-1 and
          LIST is initialized to NIL*/
        CLIST:=LIST
        LIST:=NIL
        /*loop over CLIST and compute the LOWSUM and UPSUM for
          each element of CLIST*/
        for each element L in CLIST do
            begin
              LOWSUM:=UPSUM:=0
              for I:=1 to 2**LEVEL do
                  begin
                    A:=PATTERN [LEVEL, I]
                    B:=TABLE [LEVEL, L, I]
                    LOWSUM:=LOWSUM + ABS(A-B)
                    UPSUM:=UPSUM + MIN(A+B, 2**(N-LEVEL)-(A+B))
                  end
              K:=MIN (K, UPSUM)
              if LOWSUM ≤ K then do
              /*this constructs the LIST for the next level using
                the P and Q pointers to the index table*/
                  begin
                    P:=TABLE [LEVEL, L, 2**LEVEL + 1]
                    Q:=TABLE [LEVEL, L, 2**LEVEL + 2]
                    for J:=P to Q do
                        LIST:=APPEND (LIST, J)
                  end
            end
      end
end.
```

6. Conclusions

We are currently applying the ideas described in this paper to the design and development of a pattern database system of Chinese characters. Eventually, we hope to be able to digitize a page of fixed-font Chinese characters, and "translate" the characters into a form useful for real language translation. Several major problems still have to be solved before the goal can be reached, including:

1) generalizing the matching algorithms to allow for pattern translation,

2) adding a model for distortion to the pattern matches (matching within k really only accounts for noise), and

3) developing more general regular decompositions than the pyramid which more closely reflect the spatial organization of real patterns.

References

1. A. Hanson and E. Riseman, "VISIONS: A computer system for interpreting scenes," in *Computer Vision Systems* (A. Hanson and E. Riseman, eds.) Academic Press, NY, 1978.

2. M. Kelly, "Edge detection in pictures by computer using planning," *Machine Intelligence, 6,* 379-409, 1971.

3. A. Klinger and C. Dyer, "Experiments in picture representation using regular decomposition," *Computer Graphics and Image Processing, 5,* 68-105, 1976.

4. S. Tanimoto and T. Pavlidis, "A hierarchical data structure for picture processing," ibid, 4, 104-119, 1975.

5. A. Nakamura and C. Dyer, "Bottom-up cellular pyramids for image analysis," in *Proc. 4th Int. Joint Conf. on Pattern Recognition,* Kyoto, Japan, 494-496, 1978.

6. G. Hunter, "Efficient computation and data structures for graphics," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

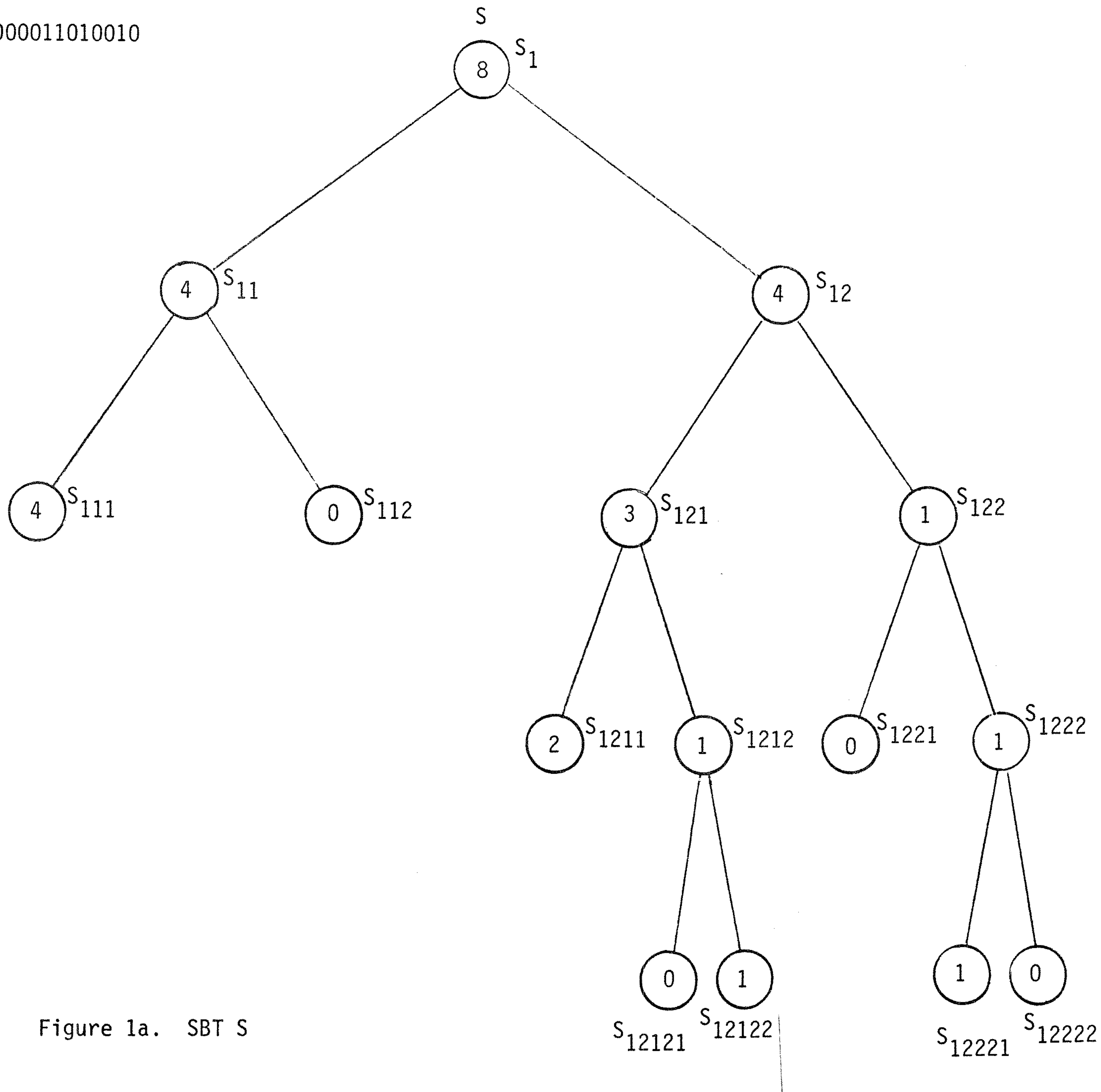

Figure 1a. SBT S

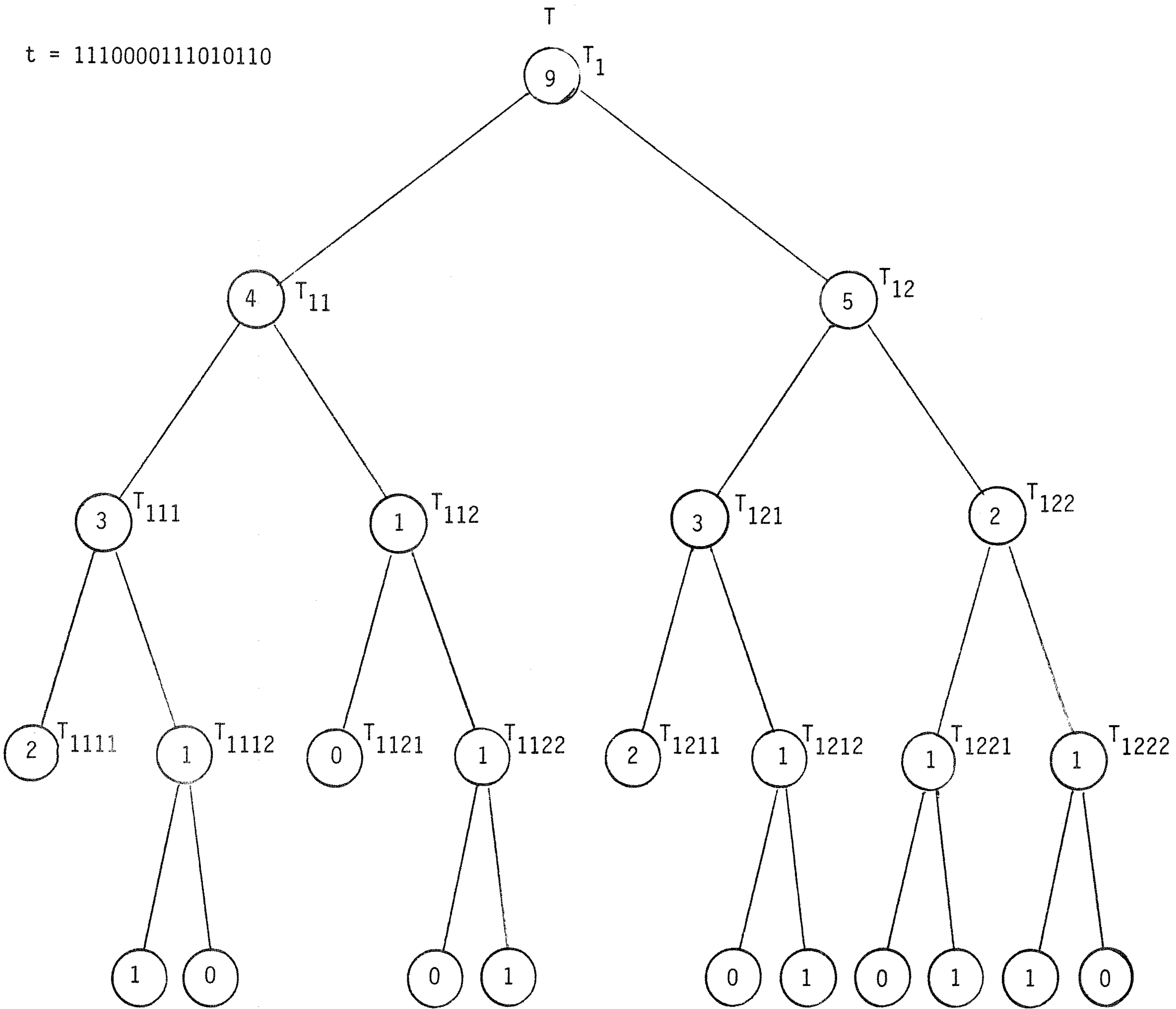

Figure 1b. SBT T

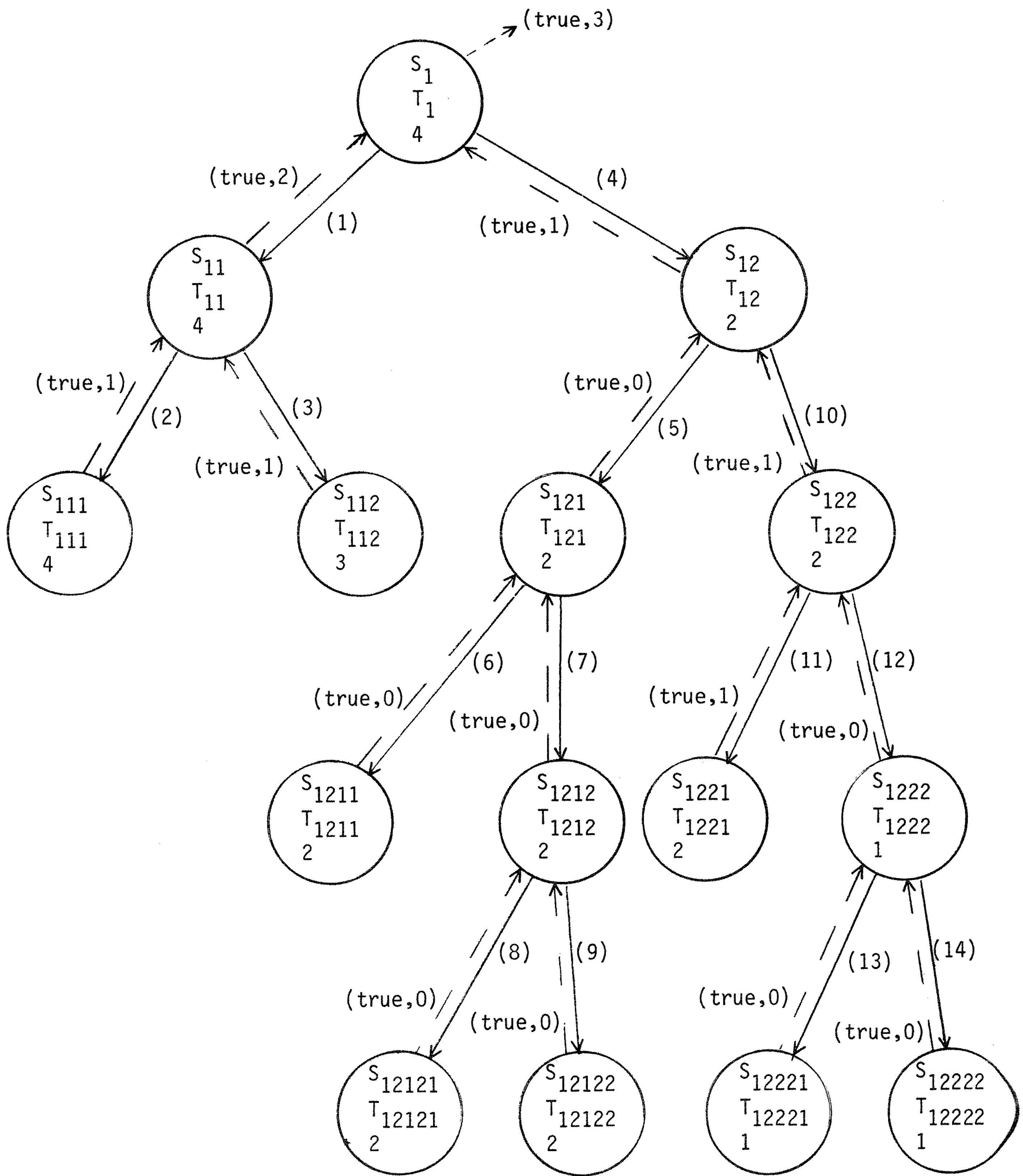

Figure 1c. DFSBT (S,T,4,mismatch)

s' = 1111000011110000

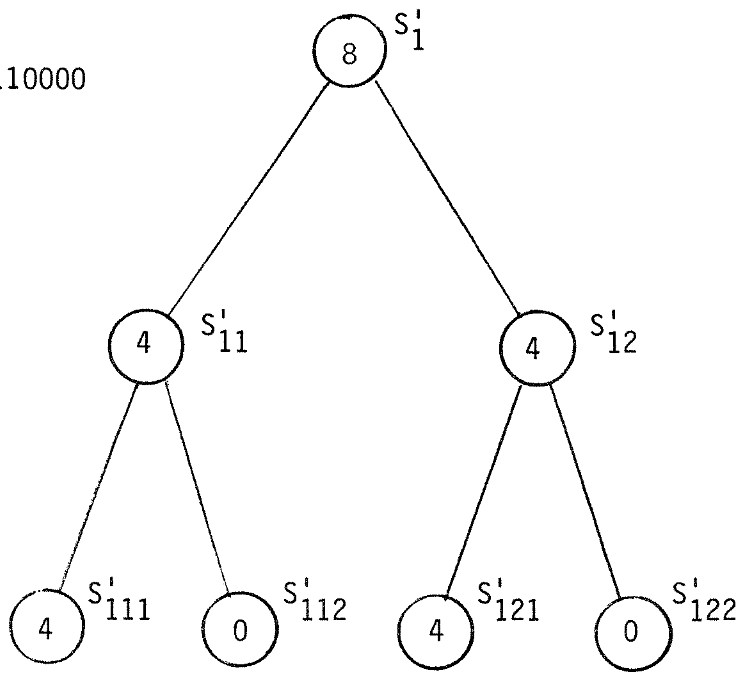

Figure 1d. SBT S'

t' = 1101011100001111

10 T'_1

6 T'_{11} 4 T'_{12}

3 T'_{111} 3 T'_{112} 0 T'_{121} 4 T'_{122}

2 1 1 2

0 1 0 1

Figure 1e. SBT T'

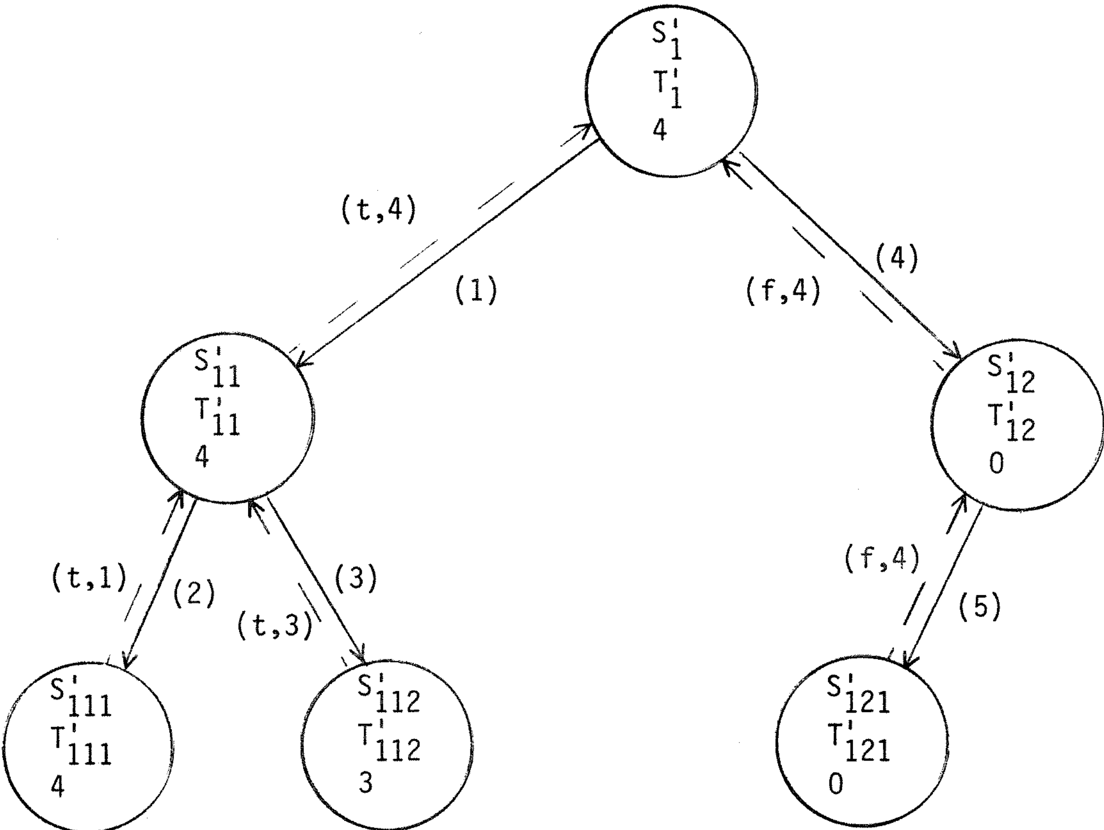

Figure 1f. DFSBT (S',T',4,mismatch)

1) MATCH $(\{(T_1,S_1,15,1)\},3)$

-Uppersum = $15 > 3$

-Lowersum = $1 < 3$

2) MATCH $(\{(T_{11},S_{11},8,0),(T_{12},S_{12},7,1)\},3)$

-Uppersum = 15

-Lowersum = 1

3) MATCH $(\{(T_{111},S_{111},1,1),(T_{112},S_{112},1,1),(T_{121},S_{121},2,0),(T_{122},S_{122},3,1)\},3)$

-Uppersum = 7

-Lowersum = 3

4) MATCH $(\{(T_{111},S_{111},1,1),(T_{112},S_{112},1,1),(T_{1211},S_{1211},0,0),$
$(T_{1212},S_{1212},1,0),(T_{1221},S_{1221},1,1),(T_{1222},S_{1222},1,0)\},3)$

-Uppersum = 5

-Lowersum = 3

5) MATCH $(\{(T_{111},S_{111},1,1),(T_{112},S_{112},1,1),(T_{1211},S_{1211},0,0),$
$(T_{12121},S_{12121},0,0),(T_{12122},S_{12122},0,0),(T_{1221},S_{1221},1,1),$
$(T_{12221},S_{12221},0,0),(T_{12222},S_{12222},0,0)\},3)$

-Uppersum = 3

-Lowersum = 3

Match = true

Figure 2a. Application of BFSBT to Figures 1a-b.

1) MATCH $(\{(S'_1, T'_1, 14, 2)\}, 3)$

-Uppersum = 14

-Lowersum = 2

2) MATCH $(\{(S'_{11}, T'_{11}, 6, 2), (S'_{12}, T'_{12}, 8, 0)\}, 3)$

-Uppersum = 14

-Lowersum = 2

3) MATCH $(\{(S'_{111}, T'_{111}, 1, 1), (S'_{112}, T'_{112}, 3, 3), (S'_{121}, T'_{121}, 4, 4), (S'_{122}, T'_{122}, 4, 4)\}, 3)$

-Uppersum = 12

-Lowersum = $12 > 3$

HALT - failure.

Figure 2b. Result of applying BFSBT to Figures 1c-d.

j \ k	4	5	6
5	.03	X	X
6	.08	.02	X
7	.12	.05	.01
8	.13	.08	.03
9	.13	.11	.05
10	.12	.12	.08

Figure 3. $\binom{j-1}{k} q'^{j-1}(1-q')^{(k-j-1)} * q'$ for $q' = .5$

ℓ \ k	4	5	6
1	.62	.5	.46
2	.27	.33	.32
3	.09	.12	.15
	.98	.95	.93

Figure 4. prob [fail first at level ℓ within k]

Entries	p^1	q^1
3	1	1
5	2	2
6	3	4
8	5	8
9	9	10
10	11	13
11	14	15
12	16	16

(a) Level 0

	Entries		p^2	q^2
1	2	1	1	1
2	4	1	2	2
3	3	3	3	4
4	4	2	5	6
5	2	6	7	7
6	3	5	8	8
7	4	4	9	10
8	5	3	11	12
9	3	6	13	14
10	5	4	15	17
11	5	5	18	19
12	6	4	20	20
13	7	3	21	21
14	5	6	22	22
15	7	4	23	23
16	6	6	24	24

(b) Level 1

	Entries				p^3	q^3
1	1	1	1	0	1	1
2	2	2	0	1	2	2
3	1	2	2	1	3	3
4	2	1	1	2	4	4
5	1	3	1	1	5	5
6	2	2	2	0	6	6
7	0	2	2	4	7	7
8	1	2	2	3	8	8
9	2	2	2	2	9	9
10	2	2	3	1	10	10
11	2	3	1	2	11	11
12	3	2	2	1	12	12
13	1	2	2	4	13	13
14	3	0	3	3	14	14
15	2	3	1	3	15	15
16	2	3	2	2	16	16
17	3	2	3	1	17	17
18	1	4	3	2	18	18
19	1	4	4	1	19	19
20	4	2	3	1	20	20
21	4	3	2	1	21	21
22	3	2	3	3	22	22
23	4	3	3	1	23	23
24	2	4	3	3	24	24

(c) Level 2

	Entries								p^4	q^4
1	1	0	1	0	1	0	0	0	1	1
2	1	1	2	0	0	0	0	1	2	2
3	1	0	1	1	2	0	0	1	3	3
4	1	1	0	1	0	1	1	1	4	4
5	1	0	1	2	0	1	0	1	5	5
6	1	1	2	0	1	1	0	0	6	6
7	0	0	2	0	1	1	2	2	7	7
8	1	0	1	1	0	2	2	1	8	8
9	1	1	1	1	1	1	1	1	9	9
10	1	1	2	0	2	1	1	0	10	10
11	0	2	2	1	1	0	1	1	11	11
12	1	2	1	1	2	0	0	1	12	12
13	0	1	1	1	1	1	2	2	13	13
14	2	1	0	0	1	2	2	1	14	14
15	1	1	2	1	1	0	2	1	15	15
16	1	1	1	2	0	2	1	1	16	16
17	1	2	1	1	2	1	1	0	17	17
18	0	1	2	2	1	2	1	1	18	18
19	0	1	2	2	2	2	0	1	19	19
20	2	2	1	1	1	2	1	0	20	20
21	2	2	2	1	2	0	1	0	21	21
22	2	1	1	1	1	2	1	2	22	22
23	2	2	1	2	1	2	0	1	23	23
24	1	1	2	2	2	1	1	2	24	24

(d) Level 3

Figure 5

	Entries																p^5	q^5
1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0		
2	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1		
3	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0		
4	1	0	0	1	0	0	1	0	0	0	0	1	0	1	0	1		
5	0	1	0	0	0	1	1	1	0	0	1	0	0	0	0	1		
6	1	0	1	0	1	1	0	0	0	1	0	1	0	0	0	0		
7	0	0	0	0	1	1	0	0	0	1	0	1	1	1	1	1		
8	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0		
9	0	1	0	1	1	0	1	0	1	0	1	0	0	1	1	0		
10	0	1	0	1	1	1	0	0	1	1	0	1	0	1	0	0		
11	0	0	1	1	1	1	1	0	1	0	0	0	1	0	0	1		
12	0	1	1	1	1	0	1	0	1	1	0	0	0	0	0	1		
13	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1		
14	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0		
15	0	1	0	1	1	1	0	1	1	0	0	0	1	1	0	1		
16	0	1	1	0	0	1	1	1	0	0	1	1	0	1	0	1		
17	1	0	1	1	0	1	0	1	1	1	1	0	1	0	0	0		
18	0	0	1	0	1	1	1	1	0	1	1	1	1	0	1	0		
19	0	0	1	0	1	1	1	1	1	1	1	1	0	0	1	0		
20	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	0		
21	1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	0		
22	1	1	0	1	0	1	0	1	0	1	1	1	0	1	1	1		
23	1	1	1	1	1	0	1	1	1	0	1	1	0	0	1	0		
24	1	0	1	0	1	1	1	1	1	1	1	0	1	0	1	1		

(e) Level 4

Figure 5 cont'd.