An Exact Algorithm for 2-D String Matching

John Drakopoulos and Panos Constantopoulos

November 1989

FORTH/CSI/TR/1989/021
Technical Report Series
Computer Science Institute, FORTH†
P.O.Box 1385, Heraklio, Crete, 711-10 Greece
Tel.: +30.81.229302, Fax: +30.81.229342, Telex: 262389 CCI GR
E-mail – UUCP: mcvax!ariadne!drakop

† "Foundation of Research and Technology – Hellas," formerly "Research Center of Crete" (RCC).
An Exact Algorithm for 2-D String Matching

John Drakopoulos and Panos Constantopoulos

Institute of Computer Science
Foundation of Research and Technology - Hellas
Heraclion, Crete, Greece

ABSTRACT

The problem of pictorial information retrieval can be transformed into one of two-dimensional string matching by representing symbolic pictures as two-dimensional strings. The known algorithm of S.K. Chang for two-dimensional string matching allows false drops in certain cases. The algorithm and the cause of the false drops are analysed. A new algorithm which avoids false drops is presented. The trade-off of accuracy versus time and memory requirements is discussed and alternative implementations are outlined.

1. Introduction

One access structure proposed for image retrieval from image databases is an index of two-dimensional (2-D) strings [1,2]. The 2-D string data structure holds information about the objects contained in a symbolic picture and their relative positions in the form of a pair of strings representing the order relation of the projections of those objects along two axes. S.K. Chang and his co-workers have presented this data structure along with an algorithm for 2-D string matching, thus reducing the problem of partial picture matching to one of 2-D substring matching [1].

It is shown in the sequel that this algorithm allows false drops in certain cases. A false drop is an instance of the database contained in the answer set of a query without satisfying the query restrictions.

The conditions of occurrence of false drops in the case of Chang's algorithm are discussed and an example of this behaviour is given using the concept of unmatch instance of a graph, which is defined here. A new algorithm for 2-D string matching which avoids false drops is proposed. This algorithm has higher time and memory requirements than Chang's, apparently the price for accuracy. Some alternative implementations are discussed in light of the performance-accuracy trade-off. A formal proof of the correctness of the proposed algorithm is given in the Appendix. For convenience and consistency of notation, the exposition starts with the definition of the 2-D string data structure and a review of Chang's algorithm.

1.1. Definitions

Let

\[ V = \text{an alphabet} \]
\[ A = \{ \langle, \rangle, \langle, \rangle \} \quad (A \cap V = \emptyset) \]

where the symbols in A are used to specify spatial relationships between pictorial objects (which are symbols in V).

Then we define

1-D string over V: \( x_1 x_2 \cdots x_n \quad (n \geq 0) \) where \( x_i \in V \).

2-D string over V: \( (x_1 y_1 x_2 y_2 \cdots y_{n-1} x_n, x_p(z) y_{p(z)} z_2 z_3 \cdots z_{n-1} x_p(a)) \)

where

\[ x_1 x_2 \cdots x_n \] : 1-D string over V.
\[ p : \{1, \ldots, n\} \to \{1, \ldots, n\} \] : permutation function over \( \{1, \ldots, n\} \).
\[ y_1 y_2 \cdots y_{n-1} \] : 1-D string over A.
\[ z_1 z_2 \cdots z_{n-1} \] : 1-D string over A.
For example consider the symbolic picture in Figure 1.

![Symbolic Picture]

**Figure 1.** A symbolic picture. The grid resolution affects the 2-D string representation.

The 2-D string representing this picture has the form:

\[
(a=\emptyset=d=\emptyset < b=\emptyset=\emptyset=\emptyset < \emptyset=c;x=\emptyset=\emptyset < \emptyset=\emptyset=d=\emptyset, \\
\emptyset=b=\emptyset<\emptyset=\emptyset=c;x=\emptyset<\emptyset=d=\emptyset < \emptyset=\emptyset=d=\emptyset) \quad \text{(absolute 2-D string).}
\]

where 'a:b' means that the objects a, b belong to the same slot, and 'a < b' means that a precedes b. There are many forms for a 2-D string. So, eliminating the '∅' above we get

\[
(a=d< b== < c;==< d=d, \\
a=b==< c;c= < d==c=d < d=d) \quad \text{(absolute 2-D string).}
\]

Also, eliminating '=' and replacing '<< by '<' recursively in the absolute 2-D string we get

\[
(ad < b < c;c < d, \ ab < c;c < d < d) \quad \text{(normal 2-D string).}
\]

From the normal 2-D string, eliminating ':' we get

\[
(ad < b < c<c < d, \ ab < c<c < d < d) \quad \text{(reduced 2-D string).}
\]

Finally, we define the augmented 2-D string over an alphabet V of a symbolic picture f to be the pair (u, w), where (u, v) is the reduced 2-D string over V of picture f and w is produced from v by replacing symbols with the permuted indices.

So, for the above example the 2-D string is

\[
(ad < b < c < d, \ 13 < 45 < 26 < 7) \quad \text{(augmented 2-D string)}
\]

More details about the various forms of 2-D strings and the transformations between them can be found in [1]. Here we only use the augmented 2-D strings.
1.2. Types of Matching

The rank of the symbol \( \sigma \) in the string \( u \) is denoted by \( r_u(\sigma) \) and is defined as:

\[
r_u(\sigma) = 1 + \text{number of } \leq \text{ preceding } \sigma \text{ in } u
\]

A string \( u \) is contained in a string \( v \), if \( u \) is a subsequence of a permutation of \( v \). A string \( u \) is a type-1 1-D subsequence of string \( v \) (i.e., 0,1,2) iff:

1. \( u \) is contained in \( v \), and
2. if \( a_1w_1b_1 \) is a substring of \( u \), \( a_1 \) matches \( a_2 \) in \( v \) and \( b_1 \) matches \( b_2 \) in \( v \), then

\[
\begin{align*}
\text{type-0:} & \quad r_u(b_1) - r_u(a_2) \geq r_u(b_1) - r_u(a_1) \\
& \quad \text{or } r_u(b_1) - r_u(a_1) = 0 \\
\text{type-1:} & \quad r_u(b_1) - r_u(a_2) \geq r_u(b_1) - r_u(a_1) > 0 \\
& \quad \text{or } r_u(b_2) - r_u(a_2) = r_u(b_1) - r_u(a_1) = 0 \\
\text{type-2:} & \quad r_u(b_2) - r_u(a_2) = r_u(b_1) - r_u(a_1)
\end{align*}
\]

Finally, a 2-D string \( (u_1, v_1) \) is a type-i 2-D subsequence of a 2-D string \( (u_2, v_2) \) (i.e., 0,1,2) iff:

1. \( u_1 \) is a type-i 1-D subsequence of \( u_2 \), and
2. \( v_1 \) is a type-i 1-D subsequence of \( v_2 \)

2. The original matching algorithm

Here we present a matching algorithm for augmented 2-D strings, following [1], with some modifications to indices and hence the meaning of symbols. Also, the permutation function has been eliminated in this version and so, the objects are accessed through a simple index. We consider the 2-D string to be a linked list of objects where each object is represented by its name and a number (named pos) which is an encoding of the \( x \)- and \( y \)-rank of the object. More specifically, this list of objects is sorted in an increasing order, according to the \( x \)-coordinate of the center point of the minimum rectangle enclosing each object. So, if \( b \) is an object next to another object \( a \) in such a list then \( \text{rank}_x(b) = \text{rank}_x(a) \) or \( \text{rank}_y(b) = \text{rank}_y(a) + 1 \) depending on whether the center point of the minimum enclosing rectangle of \( a \) and \( b \) have the same or different \( x \)-coordinates. This information is kept on the sign bit of field \( \text{pos} \), and so

\[
\text{rank}_x(b) = \begin{cases} 
\text{rank}_x(a) & \text{if } a \rightarrow \text{pos} < 0 \\
\text{rank}_x(a) + 1 & \text{otherwise}
\end{cases}
\]

Also, the absolute value of field \( \text{pos} \) is the \( y \)-rank. So, we can compute the \( \text{rank}_x(b) \) directly from the absolute value of \( \text{pos}(b) \), while knowing the \( \text{rank}_x(a) \) and the \( \text{pos}(a) \) we can compute the \( \text{rank}_x(b) \) (using the above formula). Therefore, running once over a 2-D string we can compute the \( x \)- and \( y \)-ranks of objects, (the pos-values are available and the first object in the list has x-rank equal to one).

It is not in the scope of this paper to examine how to get the list of objects from a real picture, and it is assumed that for each picture there is such a list available. Then the construction of the 2-D string from this list is as follows: The list is sorted according to the \( y \)-coordinate of the center point of the minimum enclosing rectangle of each object and the \( y \)-ranks are set into field \( \text{pos} \). The same is done with the \( x \)-coordinate and the sign bit of \( \text{pos} \) is set according to the above rule. Finally, all information in this list, except the names and the \( \text{pos} \) fields is discarded. The resulting list is the required 2-D string.
Now, consider two 2-D strings $s, t$ containing $N$ and $M$ objects respectively. The algorithm which matches $s$ with $t$ initially decomposes $s$ and $t$ into $x_1, r_1, s_1$ and $x_2, r_2, s_2$ respectively, where $x_1, x_2$ keep the names, $r_1, r_2$ keep the $x$-ranks and $s_1, s_2$ keep the $y$-ranks of objects. This is done by routine 'ConvertToArray'. After this has been accomplished, construction of the following $MI$ sets takes place:

$$MI[n] = \left\{ j \in \{0, \ldots, M-1\} \mid x_2[j] = x_1[n] \right\}, \quad (n \in \{0, \ldots, N-1\})$$

i.e., $MI[n]$ is the set of objects of $t$, which have the same name with the object indexed by $n$ in string $s$. Note that indices run over $[0, \ldots, N-1]$ and $[0, \ldots, M-1]$ rather than $[1, \ldots, N]$ and $[1, \ldots, M]$. This has been done for implementation reasons without loss of generality, a conversion to indices as in the definitions is straightforward. Also, there is a triple indexed structure 'a' which is initially empty, and which the algorithm tries to fill up so that

$$a \in a(j, n, m) \iff (x, \ldots, j) - (n-m-1, \ldots, n)$$

where by writing $(x, \ldots, j) - (m, \ldots, n)$ we mean that the sequence $(m, \ldots, n)$ is a type-i 2-D subsequence of $(x, \ldots, j)$ (for some $i \in \{0, 1, 2\}$), and the two sequences have the same length. In this case we say that the two sequences are matched sequences.

Then by setting $m = N-2$, $n = N-1$ we get

$$a \in a(j, N-1, N-2) \iff (x, \ldots, j) - (0, \ldots, N-1)$$

but the later 'subsequence' of the 2-D string $s$ is the whole string $s$. Hence if the algorithm succeeds in putting elements into the set $a(j, N-1, N-2)$ we have a successful match; otherwise an unmatch occurs.

The matching algorithm for augmented 2-D strings is given below in a C-like language:

```c
match1(s, t) /* check if the 2-D string s is a type-i 2-D subsequence of 2-D string t */
{
    /* Convert s to x1[N], r1[N], s1[N] and
        t to x2[M], r2[M], s2[M]
    where N = number of symbols in s
        M = number of symbols in t
    ConvertToArray(s, x1, r1, s1);
    ConvertToArray(t, x2, r2, s2);
    if (N > M) return(0);
    for (j = 0; j <= M-1) insert j in set mic(x2[j]));
    for (n = 0; n <= N-1) MI[n] = mic(x1[n]);
    /* j \in MI[n] \iff x2[j] = x1[n] */
    Mk = MI[0];
    for (n = 0; n <= N-2) /*
        (MC = \phi;
        for (j \in MI[n+1])
            for (k \in Mk)
                if (not agree(n+1, n, j, k)) continue;
                insert k in set a(j, n+1, 0);
            if (n == 0) continue;
            AP = a(k, n, 0);
            for (m = 1; m <= n )
                temp = \phi;
                for (i \in AP)
                    if (agree(n+1, n-m, j, i)) insert i in set temp;
```
a(j, n+1, m) = a(j, n+1, m) \cup \text{temp};
AP = \bigcup_{i \in \text{temp}} a(i, n-m, 0) \cap a(k, n, m);
}
/* for m */
if ( all sets a(j, n+1, m) (m=0,1, \ldots, n) are not empty )
  insert j in set MC;
}
/* for k */
}
/* for j */
if ( MC is empty ) return(0);
Mk = MC;
}
return(1);
}

ConvertToArray(t, x, r, s) /* convert the 2-D string t to arrays x, r, s */
{
  rank = 1;
  for ( symbol j \in t )
    {*x++ = j->name; *
     *r++ = rank;
     *s++ = j->position ;
     if ( j->position \geq 0 ) rank++;
    }
}

agree(m, n, j, k)
{
  if ( j == k ) return(0);
  ds2 = s2[j] - s2[k];
  ds1 = s1[m] - s1[n];
  /* Check the same order of (jk) and (mn) */
  if ( diff_sign(ds2, ds1) ) return(0);
  dr2 = r2[j] - r2[k];
  dr1 = r1[m] - r1[n];
  /* (supposed) m > n \iff dr1 \gg 0 */
  switch (matchtype)
    { case 0 : if ( ( diff_sign(dr2, dr1) == 0 ) &&
                   ( abs(ds2) >= abs(ds1) ) &&
                   ( abs(dr2) >= abs(dr1) ) ) return(1);
              break;
    case 1 : if ( ( dr2 == dr1 && dr1 > 0 ) ||
                  ( dr2 <= dr1 && dr1 == 0 ) )
              if ( ( abs(ds2) >= abs(ds1) &&
                    abs(ds1) > 0 ) ||
                   ( ds2 == ds1 && ds1 == 0 ) ) return(1);
              break;
    case 2 : if ( ds2 == ds1 && dr2 == dr1 ) return(1);
              break;
    default : break;
    }
  return(0);
}
The routine agree(m, n, j, k) simply examines (applying the proper relations) whether (k, j) - (n, m). Also, the variable 'matchtype' is a global variable which determines the type of matching and it is set before match(a, t) is called.

We now show that the above algorithm allows false drops. The algorithm works by solving the following recursive equation:

\[ a(j, n+1, m+1) = \bigcup_{k \in MC} \{ x \in a(k, n, m) : \text{agree}(n+1, n-m-1, j, x) = 1 \text{ and } \text{agree}(n+1, n, j, k) = 1 \text{ and } \exists i \in a(j, n+1, m) : \text{agree}(n-m, n-m-1, i, x) = 1 \} \]

This recursive equation (and so the algorithm), cannot ensure that the elements of \( a(j, n+1, m+1) \) are chosen correctly.

According to the algorithm:

\[
x \in a(j, n+1, m+1) \iff \exists k \in MC : \begin{cases} 
\text{agree}(n+1, n, j, k) = 1 \\
\text{agree}(n+1, n-m-1, j, x) = 1 \\
x \in a(k, n, m) \\
\exists i \in a(j, n+1, m) : x \in a(i, n-m, i) 
\end{cases}
\]

The scheme in Figure 2 represents these relations-requirements graphically.

![Figure 2. Graphical representation of conditions for matching subsequences.](image)

where a straight line between two symbols means that these two are the first and last symbols of a subsequence of string \( t \) and this subsequence and the corresponding one in string \( s \) are matched sequences. For example the straight line between \( x \) and \( k \) means that

\[(x, \ldots, k) - (n-m-1, \ldots, n).\]

A curved line between two symbols signifies agreement only in the order between these two symbols and the corresponding ones in string \( s \); i.e. the curved line between \( x \) and \( j \) means that

\[\text{agree}(n+1, n-m-1, j, x) = 1\]
Now to prove that \( x \in a(j, n+1, m+1) \), (i.e. to be able to make the line from \( x \) to \( j \) straight), we have to prove that \( x \) agrees in the order with each element, on at least one path from \( i \) to \( j \). In other words we have to show that there is a path \( p \) from \( i \) to \( j \) (\( p = (i, i_1, \ldots, i_m, j) \)) such that

\[
\text{for every } \mu \in \{1, \ldots, m\} \quad \text{agree}(n-\mu, n-m-1, i_{\mu-1}, x) = 1.
\]

Graphically we have to verify the curved line in Figure 3 for every point on the line (path) from \( i \) to \( j \).

\[\text{Figure 3. Proving a matching subsequence.}\]

Consider such a point \( y \). Then

\[
y \in a(j, n+1, \mu)
\]

since the points \( y, j \) are connected with a straight line. Also, from the construction of \( a(j, n+1, \mu) \), we observe that

\[
\text{for every } y \in a(j, n+1, \mu) \Rightarrow y \in a(k, n, \mu-1) \quad \text{ (for some } k \in a(j, n+1, 0))
\]

(since each element which is a candidate for insertion into \( a(j, n+1, \mu) \), is an element of \( a(k, n, \mu-1) \) for some \( k \)).

Then we have

\[
\begin{align*}
y & \in a(k, n, \mu-1) \\
x & \in a(k, n, m) \\
\mu & < m
\end{align*}
\]

but we cannot conclude from this that

\[
\text{agree}(n-\mu, n-m-1, y, x) = 1 \quad \text{(or that } (x, \ldots, y) - (n-m-1, \ldots, n-\mu))
\]

because we know that \( y \in a(k, n, m-1) \) but do not know whether \( x \in a(y, n-\mu, m-\mu) \) (which is equivalent to \( (x, \ldots, y) - (n-m-1, \ldots, n-\mu) \)). That is, we know that there are some paths from \( x \) to \( k \) (\( x \in a(k, n, m) \)), but do not know whether these paths contain the \( y \). Of course, these paths contain at least one of the symbols of \( a(j, n+1, \mu) \), but not necessarily the specific symbol \( y \).

However, if a path from \( x \) to \( j \) exists, then there is a symbol \( k \) belonging to \( a(j, n+1, 0) \), such that, the path from \( x \) to \( j \) flies over \( k \). Also, there is a symbol \( i \) belonging to \( a(j, n+1, m) \), such that the path from \( x \) to \( j \) flies over \( i \). Then the path from \( i \) to \( j \) is the path \( p \) i.e.
\[ p = (i, \ldots, k, j) \text{ and } (x, i, \ldots, k, j) \rightarrow (n-m-1, n-m, \ldots, n, n+1), \]

and for every element \( y \) of \( p \) we have that

for some \( \mu : \)

\[ y \in a(j, n+1, \mu) \]
\[ (x, \ldots, y) \rightarrow (n-m-1, \ldots, n-\mu) \]
\[ \Rightarrow \text{agree}(n-\mu, n-m-1, y, x) = 1. \]

which is what we wanted. But the algorithm does not check for \( p \), when deciding which symbols to put into \( AP \), so making them candidates for entrance into \( a(j, n+1, m+1) \). Also, searching for such a path is time consuming and raises the time-complexity of the algorithm too high. This is probably a good reason for avoiding such searching. However, if such a path exists then the algorithm will continue correctly : it simply selects elements from \( a(k, n, m) \) and does not care about \( p \). This is correct if we have secured the existence of \( p \), but we did not.

3. A False Drop Example

We now present a false drop case for the algorithm of the previous section.

Let us first define the notion of unmatch instance of a graph, which turns out to be a convenient device.

Let \( G = (V, E) \) to be a graph
where
\[ V = \text{set of vertices}, \]
\[ E = \text{set of edges connecting vertices}. \]

Then every partitioning of \( V \) into \( V_0, \ldots, V_{N-1} \) (for some \( N \)) is called an unmatch instance of \( G \) and is denoted as \( A_G \). The sets \( V_0, \ldots, V_{N-1} \) are called groups of the instance \( A_G \), and the number \( N \) is called length of \( A_G \). It is denoted as \( L(A_G) \).

There is a strong relationship between unmatch instances and pairs of symbolic pictures one of which is used as a query onto the other. More specifically, we determine that when an edge connects two vertices \( v_1 \) and \( v_2 \), and these two vertices belong to \( i_1 \) and \( i_2 \) groups respectively, then

\[ \text{agree}(i_2, i_1, v_1, v_2) = 0, \]

where \( i_2, i_1 \) are symbols of the query picture \( v_2, v_1 \) are symbols of the symbolic picture and the matching type is initially specified. For example, consider the symbolic picture and its query, as shown in Figure 4, where we have noted indices on symbols from left to right and from top to bottom. The unmatch instance, for this pair of symbolic pictures, for type-0 matching, is as shown in Figure 5. Observe that, there is an 1-1 correspondence between the elements of the query-picture and the groups of the instance. More specifically, the i-th symbol of the query corresponds to the i-th group of the unmatch instance. So, the query picture contains \( L(A_G) \) symbols.

Now we define as matching path of an unmatch instance \( A_G \), a set of \( L(A_G) \) vertices of \( G \), one from each group of \( A_G \), where no two vertices are connected together with an edge.

The existence of a matching path in an unmatch instance \( A_G \), means that the query-picture \( Q \) matched the symbolic picture \( P \), where the pair \( (P, Q) \) corresponds to \( A_G \). Observe here that, in general, there are many such pairs corresponding to an unmatch instance. But this has no effect at all, because all these pairs, are pairs of matched or unmatched pictures, depending on the existence of a matching path.

Consider now the unmatch instance shown in Figure 6 and the pair of pictures in Figure 7, which corresponds to it, as can be easily verified. In checking for a type-0 match, the algorithm of the previous section gives the following results:

\[ a(3, 1, 0) = [0, 2] \]
\[ a(8, 1, 0) = [0, 2] \]
\[ a(5, 2, 0) = [8] \]
\[ a(5, 2, 1) = [0, 2] \]
Figure 4. A pair of stored and query symbolic pictures.

Figure 5. Unmatch instance corresponding to the pair of Figure 4.

\[
\begin{align*}
\text{a}(5, 2, 0) &= \{3, 8\} \quad \text{a}(6, 2, 1) = \{2\} \\
\text{a}(4, 3, 0) &= \{5, 6\} \quad \text{a}(4, 3, 1) = \{3, 8\} \quad \text{a}(4, 3, 2) = \{0, 2\} \\
\text{a}(7, 3, 0) &= \{5, 6\} \quad \text{a}(7, 3, 1) = \{3, 8\} \quad \text{a}(7, 3, 2) = \{0, 2\} \\
\text{a}(9, 4, 0) &= \{4, 7\} \quad \text{a}(9, 4, 1) = \{5, 6\} \quad \text{a}(9, 4, 2) = \{3\} \quad \text{a}(9, 4, 3) = \{0\}
\end{align*}
\]

The last result, however, contradicts the fact that there is no matching path in the unmatch instance of Figure 6. Observe that, although all the other sets contain the proper information, the information stored in \(a(9, 4, 3)\) and derived from the contents of the other sets is incorrect. Hence a false drop has occurred.
Figure 6. An unmatch instance for which algorithm match1 incorrectly produces a matching path.

Figure 7. A pair of stored and query symbolic pictures corresponding to the unmatch instance of Figure 6.

4. An Exact Algorithm

If a matching path exists then the algorithm will find it, since the conditions it checks will be true. However, there are cases, as the example given above, where there is no matching path but the algorithm produces one. Here it is worth to note that this cannot happen in type-2 matching since this type of
matching is transitive (if agree(n, m, j, i) = 1 and agree(n, μ, k, i) = 1 then agree(n, μ, j, i) = 1) as can be derived from its conditions. However, the other types of matching suffer from the above error because they are not transitive. Here, for purposes of simplicity and illustration we manipulate all types of matching in the same way losing some amount of efficiency.

Now consider the following algorithm:

```c
match2(s, t) /* check if the 2-D string s is a type-1 2-D subsequence of 2-D string t */
{
    /* Convert s to x1[N], r1[N], s1[N] and t to x2[M], r2[M], s2[M] */
    N = number of symbols in s
    M = number of symbols in t
    ConvertToArray(s, x1, r1, s1);
    ConvertToArray(t, x2, r2, s2);
    if (N > M) return(0);
    for (j = 0, ..., M-1) insert j in set mtc(x2[j]);
    for (n = 0, ..., N-1) M[n] = mtc(x1[n]);
    /* j ∈ M[n] ⇐⇒ x2[j] = x1[n] */
    Mk = M[0];
    for (n = 0, ..., N-2)
    {
        MC = ∅;
        for (j ∈ M[n+1])
        {
            AP = Mk;
            for (m = 0, ..., n)
            {
                from_k = from_j = ∅;
                for (i ∈ AP)
                {
                    if (agree(n+1, m-i, j, i))
                    {
                        insert i in set a(j, n+1, m);
                        from_j = from_j ∪ a(i, n-m, 0);
                    }
                }
                from_k = ⋃_{i∈A, j=n+1, i, k ∈ Mk} a(k, n, m);
                AP = from_j ∩ from_k;
            }
            project(j, n);
            discard(j, n);
            if (all sets a(j, n+1, m) (m = 0, 1, ..., n) are not empty)
            insert j in set MC;
        }
    }
    /* for j */
    if (MC is empty) return(0);
    Mk = MC;
}
return(1);
}
```

project(j, n)
{
    for (m = n, n-1, ..., 1)
    {
        projPlane = a(j, n+1, m);
        for (d = 1, ..., m)
        {
            for (i ∈ a(j, n+1, m-d))
            {
                b(i, n+m+d, d-1) = a(i, n-m+d, d-1) ∩ projPlane;
            }
        }
    }
}
discard(j, n)
{
for (m = n-2, n-3, ..., 0)
  for (k ∈ a(j, n+1, m))
    for (d = 0, ..., n-m-2)
      b(k,n,m,d+1) = b(k,n-m,d+1) \cap (\bigcup_{i \in k, x=m, d} b(i,n-m-1,d)) \cap (\bigcup_{i \in k, x=m-d-1,0} b(i,n-m,d-1,0));
  for (d = 0, ..., n-1)
    a(j,n+1,d+1) = a(j,n+1,d+1) \cap (\bigcup_{k \in k, x=n, d} b(k,n,d)) \cap (\bigcup_{i \in k, x=n+1,d} b(i,n,d));
}

where the routines ConvertToArray(...) and agree(...) are as described in section 2.

We will say that there is an ill path between two symbols k and j of the symbolic picture, corresponding to u and u+n symbols of the query picture, iff there is no path from k to j and the algorithm yields k ∈ a(j, n+n, n-1).

Conversely, we will say that there is a healthy path between two symbols k and j of the symbolic picture, corresponding to u and u+n symbols of the query picture, iff there is at least one path from k to j.

The algorithm presented above stores every healthy and no ill paths into structure 'a', and so responds correctly in all cases. A detailed proof is given in the Appendix.

5. Discussion

Algorithm match2 presented in section 4 does not allow false drops, whereas algorithm match1 of section 2 does in type-0 or type-1 matching. When either algorithm responds that the query picture does not match the picture, this is really true, since error can be made only by including ill paths into structure 'a', and so only when the algorithm responds that type-0 or type-1 matching occurs. The false drop probability in queries using the algorithm of section 2 has yet to be studied, judging though from the rather contrived example of section 3 we expect it to be very small in practice.

Let us now compare these two algorithms with respect to time and memory requirements. Letting lp be the maximal length of matching tables Ml[n] (n = 0, ..., N-1) the algorithm of section 2 has time complexity of O(M) + O(N^2 * lp^3), where

\[ N = \text{number of symbols in query picture} \]
\[ M = \text{number of symbols in stored picture} \]

since we need O(M) time to construct the 'ntc' table, O(N) time to construct the 'Ml' table, and O(N^2 * lp^3) for the rest of the algorithm, because the lengths of sets Ml, Ml[n+1] and AP are not greater than lp and the highest value of n is N-1. So, the running time is

\[ O(M) + O(N) + O(N^2 * lp^3) = O(M) + O(N^2 * lp^3) \]

On the other hand, the algorithm of section 4 has time complexity of O(M') + O(N^2 * lp^3), since, in addition, the complexity of 'discard(j, n)' is O(n^2 * lp^3), the complexity of 'project(j, n)' is O(n^2 * lp), and the highest value of n is N-1. The time complexity of 'discard(j, n)' can be verified by looking at its body; so its first and third loop cause the factor n^2, its second loop causes the factor lp, and the operation inside the loops takes time O(lp). The time complexity of 'project(j, n)' can be verified similarly. Hence the running time of this algorithm is
\[ O(M) + O(N) + O(N \ast \ell p \ast (N + N^2 \ast \ell p + N^2 \ast \ell p^2)) = O(M) + O(N^3 \ast \ell p^3) \]

To implement the structure 'a', which is the main memory-consuming data element, we need \( O(N \ast M) \) sets for the algorithm of section 2, whereas for the algorithm of section 4 we need \( O(N^2 \ast M) \) sets. The reason for this requirement for the first algorithm is the fact that at step-\( n \), where the algorithm constructs the sets \( a(x, n+1, y) \), it uses only 'neighbouring' symbols, i.e. symbols belonging to sets \( a(x, n, y) \) or \( a(x, y, 0) \). This can be verified by a simple examination of the algorithm in section 2.

So we can have three \( M \ast N \) arrays \( a_0, a_1 \), and \( a_2 \), corresponding to sets of structure 'a' having the form \( a(x, y, 0) \), \( a(x, n, y) \), and \( a(x, n+1, y) \) respectively, and implement 'a' as a routine accessing these arrays, instead of an \( M \ast N \ast N \) array. Also, we swap the roles of \( a_2 \) and \( a_3 \) arrays at each step to reuse the same space. Hence, we have the following variant of algorithm match2:

```c
match3(s, t) /* check if the 2-D string s is a type-i 2-D subsequence of 2-D string t */
{
    ...
    p[1] = a0;
    p[2] = a1;
    Mk = M[0];
    for (n = 0,1,...,N-2)
    { MC = \emptyset;
      nvalue = n;
      for (j \in M[n+1])
      { for (m = 1,2,...,n) a(j, n+1, m) = \emptyset;
        for (k \in Mk)
        { ...
          // for k */
          } /* for j */
        if (MC is empty) return(0);
        Mk = MC;
        swap(p2, p3);
      }
    return(1);
  }

  a(x, y, z)
  { if (z == 0) return(a0[x][y]);
    else if (y == nvalue) return(p1[x][z]);
    else return(p2[x][z]);
  }
where we have used the pointers \( p_1 \) and \( p_2 \) to access the arrays \( a_1 \) and \( a_2 \) for swapping their roles easily at each step. Also, we have used the global variable 'nvalue' in order to distinguish accesses to \( a(\cdot, n+1, \cdot) \) from accesses to \( a(\cdot, n+1, \cdot) \), during the \( n \)-th iteration of the outermost loop.

We can reduce the time complexity of the above algorithm by eliminating the third loop (for \( (k \in Mk) \) and selecting elements from \( a(j, n+1, 0) \), i.e. by using the body of algorithm match2 without any call to procedures 'project' and 'discard'. Thus we get the following algorithm:
match4(s, t)
{
    ... 
    p1 = a2;
    p2 = a1;
    Mk = M[0];
    for (n = 0,1,...,N-2)
    {
        MC = ∅;
        nvalue = n;
        for (j ∈ M[n+1])
        {
            AP = Mk;
            for (m = 0,1,...,n)
            {
                ... 
            }
            if (all sets a(j, n+1, m) (m=0,1,...,n) are not empty)
                insert j in set MC;
        } /* for j */
        if (MC is empty) return(0);
        Mk = MC;
        swap(p1, p2);
    }
    return(1);
}

where 'a' is a routine as above.

This algorithm has time complexity of O(N² * lp²), but it is more open to false drops than match1.

It is possible to discard ill paths from structure 'a', just after the matching procedure, keeping the
time complexity low. An algorithm to this end is shown below:

discard_ill_paths()
{
    for (j ∈ MC)
    {
        for (d = 1, ..., N-2)
        {
            for (m = N-2, N-3, ..., d)
            {
                top = a(j, N-1, m-d);
                bot = a(j, N-1, m);
                un = temp = ∅;
                for (i ∈ top)
                {
                    verif = a(i, N-2-m+d, d-1) ∩ bot;
                    if (verif ≠ ∅)
                    {
                        un = un ∪ verif;
                        insert i in set temp;
                    }
                }
                a(j, N-1, m-d) = temp;
                a(j, N-1, m) = un;
            }
        }
    }
}
Now the last 'return' of procedure 'match4' must be replaced by the following instruction:

```
return(discard_ill_paths());
```

This algorithm discards ill paths that have components which cannot be paths, since there is no way to access them by 'traveling' across a path, from a previous element belonging to this ill path. More specifically, suppose that \(j \in \text{MC}\) at last step of the algorithm \((n = N-2)\). Then

for every \(n \in \{0, \ldots, N-3\}\)
for every \(n' \in \{n+1, \ldots, N-2\}\)
for every \(w' \in a(j, N-1, N-2-n')\) must be

\[a(j, N-1, N-2-n) \cap a(w', n', n'-n-1) \neq \emptyset\]

i.e. for every symbol \(w'\) of the \(n'\)-group of the unmatch instance there must exist at least one symbol \(w\) of the \(n\)-group, such that the path from \(w\) to \(j\) lies over \(w'\). So, \(w\) must belong to both sets \(a(j, N-1, N-2-n)\) and \(a(w', n', n'-n-1)\). If the result of this intersection is the empty set, then such a \(w\) does not exist and so \(w'\) must be discarded from \(a(j, N-1, N-2-n')\). Also, the set \(a(j, N-1, N-2-n)\) must be updated by 'projecting' onto it (across paths) the remaining elements of the set \(a(j, N-1, N-2-n')\).

Now setting \(d = n' - n\), \(m = N-n-2\) we have

\[
\left\{ \begin{array}{l}
0 \leq n \leq N-3 \\
n+1 \leq n' \leq N-2-n-2
\end{array} \right\} \Rightarrow \left\{ \begin{array}{l}
1 \leq N-n-2 \leq N-2 \\
1 \leq n' \leq N-2
\end{array} \right\} \Rightarrow \left\{ \begin{array}{l}
d \leq m \leq N-2 \\
1 \leq d \leq N-2
\end{array} \right\}
\]

and

\[N-2-n' = m - d, \quad n' = N-2-m+d\]

So, by making these substitutions, we get the conditions the procedure 'discard_ill_paths' examines.

However, this last variant destroys the 'locality' which the algorithm originally had with respect to structure 'a', since it uses symbols far in 'a'. So, the memory space now required is \(O(N^2 \cdot M)\) measured in sets. Of course, these sets have size \(O(ip)\) and so the space actually required by these algorithms is \(O(N \cdot M \cdot ip)\) or \(O(N^2 \cdot M \cdot ip)\).

All the algorithms presented in this paper have been implemented in the C programming language under the UNIX operating system and are running on Sun workstations. Many examples have been used for testing them and none caused false responses, except some constructed especially for this purpose.

**Conclusion**

2-D string matching can be performed exactly in \(O(N^3 \cdot ip^2)\) time and with \(O(N^2 \cdot M)\) memory space using algorithm match2. However, alternative algorithms faster and more economic in memory than the exact one can be constructed. Each of these algorithms allows false drops which, however, are expected to be rare in practice.
Appendix

A formal proof of the correctness of the algorithm match2 introduced in section 4 is given here. In the following we denote the structures a and b after discarding (at each step) by \( \bar{a} \) and \( \bar{b} \), respectively. Before discarding they keep their original names. From the algorithm, we have that

1. \[ AP = AP(j, n, m) = (\bigcup_{k \in S(j, n+1, 0)} \bar{a}(k, n, m)) \cap (\bigcup_{i \in S(j, n, m)} \bar{a}(i, n-m, 0)) \]

2. \[ a(j, n+1, m+1) = \{ x \in AP(j, n, m) / \text{agree}(n+1, n-m-1, j, x) = 1 \} \]

3. \[ b(j, n+m+d, d-1) = \bar{a}(i, n-m+d, d-1) \cap a(j, n+1, m) \]

4. \[ \bar{b}(k, n-m, d+1) = b(k, n-m, d+1) \cap (\bigcup_{i \in S(k, n, 0)} \bar{b}(i, n-m-1, d)) \cap (\bigcup_{i \in S(k, n, 0)} \bar{b}(i, n-m-d-1, 0)) \]

5. \[ \bar{a}(j, n+1, d+1) = a(j, n+1, d+1) \cap (\bigcup_{k \in S(j, n+1, 0)} \bar{b}(k, n, d)) \cap (\bigcup_{i \in S(j, n+1, 0)} \bar{b}(i, n-d, 0)) \]

where we have put indices to the AP set for discrimination reasons. Of course in step-\( n \) where we execute 'discard(\( j, n \))' is

\[ a(l, n-m, d) = \bar{a}(l, n-m, d) \quad \text{for every } l, m \in \{ 0, \ldots, n-1 \}, \; d \in \{ 0, \ldots, n-m-1 \}. \]

Also at each step is

\[ a(j, n, 0) = \bar{a}(j, n, 0). \]

The proof uses five lemmas.

**Lemma 1**: The algorithm of section 4 stores into structure 'a' each healthy path, from an element \( j_0 \) belonging to group 0, to an element \( j_{n-1} \) belonging to group \( N-1 \) of the unmatch instance, where \( N = L(\text{unmatch instance}) \) = number of symbols in query symbolic picture.

**Proof**

Let us consider such a path

\((j_0, j_1, \ldots, j_{n-1}) = (0, 1, \ldots, N-1)\)

To continue with the proof of this proposition we need the following three lemmas.

**Lemma 2**: If \( j_{n-d-1} \in \bar{b}(k, n, d) \) and \( j_{n-d} \in \bar{b}(j_{n+1}, n+1, d) \) then

\[ j_{n-d-1} \in a(j_{n+1}, n+1, d+1) \]

**Proof**

It is

\[ (j_n, j_{n+1}) = (n, n+1) \Rightarrow \text{agree}(n+1, n, j_{n+1}, j_n) = 1 \]

\[ \Rightarrow j_n \in a(j_{n+1}, n+1, 0) \]

and because \( j_{n-d-1} \in \bar{a}(j_n, n, d) \) (hypothesis), we get

\[ j_{n-d-1} \in \bigcup_{k \in S(j_n, n+1, 0)} \bar{a}(k, n, d) \] (6).
Similarly \( j_{n-d-1} \in a(j_{n-d}, n-d, 0) \), and because \( j_{n-d} \in \overline{a}(j_{n+1}, n+1, d) \) (hypothesis), we get

\[
j_{n-d-1} \in \bigcup_{i \in \overline{a}(j_{n+1}, n, d)} a(i, n-d, 0)
\] (7).

But (1), (6), (7) give that

\[
j_{n-d-1} \in AP(j_{n+1}, n, d)
\]

which combined with

\[
(j_{n-d-1}, \ldots, j_{n+1}) - (n-d-1, \ldots, n+1) \Rightarrow \text{agree}(n+1, n-d-1, j_{n+1}, j_{n-d-1}) = 1
\]

and (2) completes the proof of this lemma.

\[\text{Lemma 3: For every } n \in \{0, \ldots, N-3\}, \quad j_n \in b(j_{n+1}, n+1, 0), \quad j_n \in \overline{a}(j_{n+2}, n+2, 1).\]

\[\text{Proof}\]

It is

\[
(j_n, j_{n+1}) - (n, n+1) \Rightarrow \text{agree}(n+1, n, j_{n+1}, j_n) = 1
\]
\[
\Rightarrow j_n \in a(j_{n+1}, n+1, 0) = \overline{a}(j_{n+1}, n+1, 0)
\]
\[
(j_{n+1}, j_{n+2}) - (n+1, n+2) \Rightarrow \text{agree}(n+2, n+1, j_{n+2}, j_{n+1}) = 1
\]
\[
\Rightarrow j_{n+1} \in a(j_{n+2}, n+2, 0) = \overline{a}(j_{n+2}, n+2, 0)
\]

and (3) after 'project \( j_{n+2}, n+1 \)' for \( n \) substituted by \( n+1, m = d = 1 \) and \( j = j_{n+2} \) we have that

\[
b(j_{n+1}, n+1, 0) = \overline{a}(j_{n+1}, n+1, 0) \cap a(j_{n+2}, n+2, 1)
\]

by (8) we get that

\[
j_n \in b(j_{n+1}, n+1, 0) = \overline{a}(j_{n+1}, n+1, 0).
\]

This relation by (9), (10) and (5) gives that

\[
j_n \in \overline{a}(j_{n+2}, n+2, 1),
\]

while these two last relations complete the proof of this lemma.

\[\text{Lemma 4: For every } n \in \{0, \ldots, N-3\}, \quad \text{for every } i \in \{0, \ldots, n\} \quad j_i \in b(j_{i+1}, n+1, n-i), j_i \in \overline{a}(j_{i+2}, n+2, n-i+1).\]

\[\text{Proof}\]

We will prove the above statement by induction on \( n \).

\[n = 0:\]

then \( i = 0 \) and so, we have to prove that \( j_0 \in b(j_1, 1, 0) \) and \( j_0 \in \overline{a}(j_2, 2, 1) \).
These relations are the results of lemma 3 for $n = 0$.

**Induction hypothesis:**

$$j_{i} \in \bar{b}(j_{i}, n, n-i-1), \quad j_{i} \in \bar{a}(j_{i+1}, n+1, n-i)$$

**Induction step:**

We have to prove this statement for $n$; i.e. we have to prove that for every $d \in \{0, \ldots, n\}$ it is

$$j_{n-d} \in \bar{b}(j_{n+1}, n+1, d), \quad j_{n-d} \in \bar{a}(j_{n+2}, n+2, d+1)$$

where $d = n - i$.

This statement will be proved by induction on $d$.

$d = 0$

the relations to be proved are the results of lemma 3.

**Induction hypothesis:**

$$j_{n-d} \in \bar{b}(j_{n+1}, n+1, d'), \quad j_{n-d} \in \bar{a}(j_{n+2}, n+2, d'+1)$$

**Induction step:**

We have to prove this statement for $d' = d + 1$. But from induction hypothesis <2>, for $d' = d$, and from induction hypothesis <1>, for $i = n - d - 1$, we get respectively:

$$j_{n-d} \in \bar{a}(j_{n+2}, n+2, d+1)$$

and by applying lemma 2 to these relations we get that

$$j_{n-d} \in a(j_{n+3}, n+2, d+2)$$

Also by (9) and (3) (for $n$ substituted by $n+1$, $m$, $d$ substituted by $d+2$, $i = j_{n+1}$ and $j = j_{n+2}$) we have that

$$b(j_{n+1}, n+1, d+1) = \bar{a}(j_{n+1}, n+1, d+1) \cap a(j_{n+2}, n+2, d+2).$$

But the last three relations give us that

$$j_{n-d} \in b(j_{n+1}, n+1, d+1)$$

Lemma 3 and induction hypothesis <1> (for $i = n - d - 1$), give respectively:

$$j_{i} \in \bar{b}(j_{i+1}, n+1, 0)$$

which give us that

$$j_{n-d} \in \bigcup_{k \in \bar{b}(j_{n+1}, 0)} \bar{b}(k, n, d)$$

Also lemma 3 and induction hypothesis <2> (for $d' = d$), give respectively:

$$j_{n-d} \in \bar{b}(j_{n-d}, n-d, 0)$$

which give us that

$$j_{n-d} \in \bigcup_{i \in \bar{b}(j_{n+1}, d)} \bar{b}(i, n-d, 0)$$
Now by (4), (13), (14), and (16) we get that
\[ j_{n-d-1} \in \overline{b}(j_{n-1}, n+1, d+1) \]  
which is the first relation we want to prove.

Now, from lemma 3 we have that
\[ j_{n+1} \in \overline{a}(j_{n+3}, n+2, 0). \]

This last relation together with (17) give us that
\[ j_{n-d-1} \in \bigcup_{k \in \mathbb{N}_{n+2, d+1}} \overline{b}(k, n+1, d+1). \]

Also (11) and (15) give us that
\[ j_{n-d-1} \in \bigcup_{i \in \mathbb{N}_{n+2, d+1}} \overline{b}(i, n-d, 0). \]

So, by (5), (12), and the last two equations above we get that
\[ j_{n-d-1} \in \overline{a}(j_{n+2}, n+2, d+2) \]
which together with (17) completes the induction and so the proof of lemma 4.

Now, we can return to the proof of lemma 1. More specifically, from lemma 4, for \( n = N - 3 \), we get that
\[ \text{for every } i \in \{0, \ldots, N-3\} \quad j_i \in \overline{a}(j_{i+1}, N-1, N-2-i). \]

and so, we have to prove that \( j_{N-2} \in \overline{a}(j_{N-1}, N-1, 0) \).

But
\[ (j_{N-2}, j_{N-1}) - (N-2, N-1) \Rightarrow \text{agree}(N-1, N-2, j_{N-1}, j_{N-2}) \]
\[ \Rightarrow j_{N-2} \in \overline{a}(j_{N-1}, N-1, 0) = \overline{a}(j_{N-1}, N-1, 0). \]

Therefore the path \( (j_0, \ldots, j_{N-1}) \) is in fact stored into structure 'a'.

\textbf{Lemma 5 :} The algorithm match2 stores no ill paths into structure 'a'.

\textbf{Proof}

To prove this statement we will prove (by induction on \( n \)) that after the end of step-\( n \) (of outermost loop), there are no ill paths in sets
\[ \overline{a}(j, v, m) \quad (v \in [1, \ldots, n+1], m \in \{0, \ldots, v-1\}, j \in MI[v]) \]
(after the end of step-\( n \))

and in sets
\[ \overline{b}(j, v, m) \quad (v \in [1, \ldots, n+1], m \in \{0, \ldots, v-1\}, j \in MI[v]) \]
(after the end of step-\( (n+1) \)).

\( n = 0 : \)

Then \( v \in \{1\}, m \in \{0\} \) and so
\[ x \in \bar{a}(j, v, m) \iff x \in \bar{a}(j, 1, 0) \]
\[ \iff x \in a(j, 1, 0) \]
\[ \iff \text{agree}(1, 0, j, x) = 1 \]
\[ \iff (x, j) = (0, 1). \]  \( (18) \).

So, there is a healthy path between \( x \) and \( j \), for every \( x \in \bar{a}(j, 1, 0) \) i.e. there are no ill paths in set \( \bar{a}(j, 1, 0) \) at step-0.

Also
\[ x \in \bar{b}(j, 1, 0) = b(j, 1, 0) = \bar{a}(j, 1, 0) \cap \text{projPlane} \subseteq \bar{a}(j, 1, 0) \]
and so by \( (18) \)
\[ (x, j) - (0, 1) \]  \( (\text{for every } x \in \bar{b}(j, 1, 0)) \)
i.e. there are no ill paths in set \( \bar{b}(j, 1, 0) \) at step-1.

**Induction hypothesis:**
Consider the above statement true for \( n \) i.e.
\[ \text{for every } v \in \{1, \ldots, n\} \text{ for every } m \in \{0, \ldots, v-1\} \text{ for every } j \in Ml[v] \]
\[ \text{at step-}(v-1) \text{ there are no ill paths in sets } \bar{a}(j, v, m) \]
\[ \text{at step-}v \text{ there are no ill paths in sets } \bar{b}(j, v, m) \]

**Induction step:**
Now at step-\( n \) the elements of structure 'a', with the middle index smaller than \( n+1 \), are not changed and so, according to the induction hypothesis, they continue to contain no ill paths. Also, at step-\( n+1 \), by (3) and (5) we have that
\[ \text{for every } v \in \{1, \ldots, n\} \text{ for every } m \in \{0, \ldots, v-1\} \text{ for every } j \in Ml[v] \]
\[ \bar{b}(j, v, m) \subseteq b(j, v, m) \subseteq a(j, v, m) = \bar{a}(j, v, m) \]
and since no ill paths are contained in sets \( \bar{a}(j, v, m) \) the same is true for the sets \( \bar{b}(j, v, m) \). So, to complete the induction we have to prove that the same is true for \( v = n+1 \) i.e. we have to prove
\[ \text{at step-}n \text{ there are no ill paths in sets } \bar{a}(j, n+1, m) \]
\[ \text{at step-}(n+1) \text{ there are no ill paths in sets } \bar{b}(j, n+1, m) \]
\( (m=0, \ldots, n). \)

To prove this, let's consider an ill path between \( j_u \) and \( j \), that is \( j_u \in a(j, n+1, n-u) \), where \( j_u \) and \( j \) belong to the \( u \)- and \( (n+1) \)-group of the unmatch instance respectively \( (u < n+1) \).

We will prove that
\[ j_u \text{ does not belong to } \bar{a}(j, n+1, n-u) \]  \( (\text{at step-}n) \)
and
\[ j_u \text{ does not belong to } \bar{b}(j, n+1, n-u) \]  \( (\text{at step-}(n+1)) \).

Now if
\[ \text{there is } j_u \in \bar{a}(j, n+1, 0) : j_u \in \bar{b}(j_u, n, n-u-1) \]
then
\[ (j_u, \ldots, j_n) - (u, \ldots, n) \]
since we know (from induction hypothesis) that there are no ill paths stored in sets $\bar{b}(i, v, m)$ (for every $i$, $v = 1, \ldots, n$, and $m = 0, \ldots, v-1$) and so in set $b(j_a, n, n-u-1)$ ($i = j_a$, $v = n$, $m = n-u-1 \leq n-1 = v-1$).

So for every $i \in \{u+1, \ldots, n-1\}$ there is $j_i \in \bar{b}(j_a, n, n-i-1)$ (19)

such that

$$(j_a, j_{u+1}, \ldots, j_{n-1}, j_i) \in (u, u+1, \ldots, n-1, n).$$

Now we have that

$$(j_a, j_{u+1}, \ldots, j_{n-1}, j_i, j) \in (u, u+1, \ldots, n-1, n, n+1)$$

iff

(a). agree$(n+1, u, j, j_b) = 1$
(b). agree$(n+1, n, j, j_a) = 1$
(c). agree$(n+1, i, j, j_i) = 1$ ($i \in \{u+1, \ldots, n-1\}$).

But the part of the algorithm storing elements into sets $a(x, y, z)$ and so into sets $b(x, y, z)$ requires that

$$w \in a(x, y, z) \Rightarrow \text{agree}(y, y-z-1, x, w) = 1$$

and we have supposed that

$$j_a \in a(j_a, n+1, n-u) \quad \text{and} \quad j_a \in \bar{a}(j_a, n+1, 0) = a(j_a, n+1, 0).$$

So (a), (b) above are true. Also by (4) and (3) we get that

$$\bar{b}(j_a, n, n-i-1) \subset b(j_a, n, n-i-1) \subset a(j_a, n+1, n-i)$$

which by (19) gives that

$$j_i \in a(j_a, n, n-i-1)$$

and so

$$\text{agree}(n+1, i, j, j_i) = 1 \quad \text{for every } i \in \{u+1, \ldots, n-1\}.$$\n
Therefore (a), (b), (c) are true and so

$$(j_a, j_{u+1}, \ldots, j_{n-1}, j_i, j) \in (u, u+1, \ldots, n-1, n, n+1) : \text{contradiction}$$

since there is ill path (i.e. no path) between $j_a$ and $j$.

Therefore the hypothesis that

there is $j_a \in \bar{a}(j_a, n+1, 0)$ : $j_a \in \bar{b}(j_a, n, n-u-1)$

is false; i.e.

$j_a$ does not belong to $\bigcup_{k \in \mathbb{N}, k+1, 0} \bar{b}(k, n, n-u-1)$

and so
\( j_u \) does not belong to \( \overline{a}(j, n+1, n-u) \) since (by (5))

\[
\overline{a}(j, n+1, n-u) \subseteq \bigcup_{k \in \mathbb{N}, x+1,0} \overline{b}(k, n, n-u-1).
\]

Now to finish the proof we have to prove that

\( j_u \) does not belong to \( \overline{b}(j, n+1, n-u) \) (at step \((n+1)\)).

But by (2) (at step \((n+1)\)) we have that

\[
b(j, n+1, n-u) = \overline{a}(j, n+1, n-u) \cap \text{projPlane} \subseteq \overline{a}(j, n+1, n-u)
\]

and since

\( j_u \) does not belong to \( \overline{a}(j, n+1, n-u) \)

we have that

\( j_u \) does not belong to \( b(j, n+1, n-u) \) \hspace{1cm} (20).

Also by (4) we have that

\[
\overline{b}(j, n+1, n-u) \subseteq b(j, n+1, n-u)
\]

which by (20) gives that

\( j_u \) does not belong to \( \overline{b}(j, n+1, n-u) \).

So, for every ill path that could be contained in sets \( \overline{a}(j, n+1, n-u) \) or \( \overline{b}(j, n+1, n-u) \) we have proved that it is not contained and this holds for \( u < n+1 \). So, \( n-u \in \{0, \ldots , n\} \). Hence, the sets \( \overline{a}(j, n+1, m), \overline{b}(j, n+1, m) \) \((m = n-u \in \{0, \ldots , n\})\) contain no ill paths, and this completes the induction.

**Theorem**: Algorithm match2 performs 2-D string matching exactly.

**Proof**

By lemmas 1 and 5, we have that all the healthy paths and no ill paths are stored into structure 'a'. So, if a query picture matches a stored picture, then there is at least one healthy path from an element of the first group of the unmatched instance, to an element of the last group, and it is stored in structure 'a'. This causes the algorithm to respond 'yes' which is the correct answer in this case. On the other hand if the query picture does not match the stored picture, then there is no healthy path, from an element of the first partition to an element of the last partition. So, no such path is stored into structure 'a', since the algorithm does not store ill paths to 'a'. This causes the algorithm to respond 'no' which is the correct answer again. Hence in any case the algorithm responds correctly.

**References**