Computer Science Vol. 4. 2002 004.43:519.712

*Adam Drozdek**

HIRSCHBERG'S ALGORITHM FOR APPROXIMATE MATCHING

The Hirschberg algorithm was devised to solve the longest common subsequence problem. The paper discusses the way of adopting the algorithm to solve the string matching problem in linear space to determine edit distance for two strings and their alignment.
***Keywords**: pattern matching, string processing, edit distance, Hirschberg's algorithm*

ALGORYTM HIRSCHBERGA DLA PROBLEMU PRZYBLIŻONEGO WYSZUKIWANIA WZORCA
Algorytm Hirschberga został podany w celu rozwiązania problemu najdłuższego wspólnego podciągu. Niniejszy artykuł prezentuje sposób zaadoptowania tego algorytmu do rozwiązania przy liniowych wymogach pamięciowych problemu wyszukiwania wzorca w celu znalezienia odległości edycyjnej dwóch tekstów i ich wyrównania.
***Słowa kluczowe:** wyszukiwanie wzorca, przetwarzanie tekstów, odległość edycyjna, algorytm Hirschberga*

1. Introduction

For two strings s_1 and s_2, a common subsequence is the sequence of characters that occur in both strings, not necessarily in consecutive order. For example, *es*, *ece* and *ee* are common subsequences in *predecessor* and *descendant*. There exists a connection between the longest common subsequence (*lcs*) and edit distance expressed by the formula

$$d(s_1, s_2) = |s_1| + |s_2| - 2lcs(s_1, s_2)$$

where $d(s_1, s_2)$ is a restricted Levenshtein edit distance which refers only to deletion and insertion with the cost equal to 1, so that substitution is, in effect, replaced by deletion followed by an insertion and therefore, the cost of substitution is equal to 2 ([1], 240–241, already suggested in [5], 173). Sometimes the Hirschberg algorithm is presented in the context of discussing Levenshtein edit distance, with any cost of edit operations, without any attempt to adopt it to this situation ([4], 57–62). One author makes an effort to adopt the Hirschberg algorithm to find the edit distance with the cost of all the three edit operations equal to 1 ([2], 258). In this paper a version of the Hirschberg algorithm is presented to the find edit distance of two strings and determine an alignment.

* Department of Mathematics and Computer Scince, Duquesne University, Pittsburgh

2. The Wagner – Fisher algorithm

A classical algorithm to find edit distance was presented by Wagner and Fischer [5]. The algorithm relies on the following definitions. Let $D(i,j) = d(s_1(1..i), s_2(1..j))$ be the edit distance between the prefixes $s_1(1..i)$ of s_1 and $s_2(1..j)$ of s_2. Then

$$D(i,j) = \min(D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+c(i,j))$$

where: $c(i,j) = 0$ if $s_{1,i} = s_{2,j}$ and 1 otherwise, $D(i,0) = i$, and $D(0,j) = j$.

The algorithm itself is as follows:

```
WagnerFischer(D, s1, s2)
    for i := 0  to |s1|
        D(i,0) := i;
    for j := 0 to |s2|
        D(0,j) := j;
    for i := 1 to |s1|
        for j := 1 to |s2|
            x := D(i-1,j)+1; //upper
            y := D(i,j-1)+1; //left
            z := D(i-1,j-1); //diagonal
            if s1i ≠ s2j
                z := z+1;
            D(i,j) := min(x,y,z);
```

For example, for $s_1 =$ *capital* and $s_2 =$ *apple* the algorithm produces the following edit matrix:

		a	p	p	l	e
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	3	4	5
p	3	2	1	2	3	4
i	4	3	2	2	3	4
t	5	4	3	3	3	4
a	6	5	4	4	4	4
l	7	6	5	5	4	5

so that d(*capital, apple*) = $D(7, 5) = 5$. It is clear from the use of nested `for` loops that the algorithm runs in $O(|s_1||s_2|)$ time and space. With an additional algorithm

```
WagnerFisherPrint(D, s1, s2)
    i = |s1|;
    j = |s2|;
    while i ≠ 0 or j ≠ 0
        output pair (i, j);
        if i  > 0 and D (i - 1, j)  < D (i, j)          //up
```

```
            ss1.push (s1_i);
            ss2.push ('-');
            i := i - 1;
        else if j  >  0 and D (i, j - 1)  <  D(i, j)    //left
            ss1.push ('-');
            ss2.push (s2_j);
            j := j - 1;
        else // if  i  > 0  and  j  > 0 and          //diagonally
            //  (D (i - 1, j - 1) = D (i, j)  and s1_i  =  s2_j  or
            //  D (i - 1, j - 1)  <  D (i, j)  and s1_i ≠ s2_j)
            ss1.push (s1_i);
            ss2.push (s2_j);
            i := i-1;
            j := j-1;
    print stack ss1;
    print stack ss2;
```

one possible alignment can be printed:

```
    path:    [7 5] [6 5] [5 4] [4 3] [3 2] [2 1] [1 0]
    capital
    -apple-
```

Since at least one of the indexes i and j is decremented in each iteration of the `while` loop, the algorithm runs in $O(|s_1|+|s_2|)$ time.

The algorithm `WagnerFischer()` can be improved by using only linear space $O(|s_2|)$:

```
WagnerFischer1D (current, previous, following, s1, s2, i1, i2, j1, j2)
    n  := i2 - i1+1;  m  := j2 - j1 + 1;
    for j  :=  0 to m
        previous [j]  := j;
    for i  := 1 to n + 1
        if i = n+1
            following[0] := i;
        else current[0] := i;
        for j := 1 to m
            if i = n+1
                x  := current[j] + 1;
                y  := following[j - 1] + 1;
                z  := current[j - 1];
            else x := previous[j] + 1;
                y := current[j - 1] + 1;
                z := previous[j - 1];
```

```
        if s2[j+j1-1] ≠ s1[i+i1-1]
            z  := z + 1;
        if  i = n + 1
            following[j]  :=  min (x,y,z);
        else current[j]  := min (x,y,z);
    if i  <  n  // don't overwrite previous from last iteration;
        for j := 0 to m
            previous[j]  :=  current[j];
```

This improvement, however, gives only the edit distance, but does not produce the alignment and that is what Hirschberg's algorithm does[3].

3. The Hirshberg algorithm

The Hirschberg algorithm recursively divides a string s_1 into two even (or almost even) parts s_{11} and s_{12} and for each part it finds a prefix s_{21} and suffix s_{22} of s_2 so that concatenation of alignments for s_{21} and s_{11} and then for *reverse*(s_{12}) and *reverse*(s_{22}) renders an alignment for s_1 and s_2. If needed, the algorithm is reapplied to the two halves of s_{11} and two halves of s_{12} and is continued until $|s_{1i}| \leq 2$ and $|s_2^j| \leq 2$ at which point the generating of subalignments cannot be delayed any longer.

String s_1 is bisected, but the procedure of dividing s_2 relies on the following statement. For any i dividing s_1 into two parts,

$$d(s_1, s_2) = \min\{d(s_1(1..i), s_2(1..j)) + d(reverse(s_1(i+1..|s_1|)), reverse(s_2(j+1..|s_2|))) : 1 \leq j \leq |s_2|\} \quad (1)$$

That is, the edit distance for the entire strings s_1 and s_2 equals the minimum among all combined edit distances for the first two parts of these strings and their second parts. If the minimum edit distance for the entire strings is known, then an alignment corresponding to the edit distance can be constructed; for a j dividing s_2, the alignment can be divided into two subalignments, one for strings $s_1(1..i)$ and $s_2(1..j)$ and one for strings *reverse*($s_1(i+1..|s_1|)$) and *reverse*($s_2(j+1..|s_2|)$). By the definition of the minimum edit distance $d(s_1, s_2)$, the combined edit distance corresponding to the two subalignments $d(s_1(1..i), s_2(1..j)) + d(reverse(s_1(i+1..|s_1|)), reverse(s_2(j+1..|s_2|)) = d(s_1, s_2)$.

The thrust of the algorithm is in finding an index j that renders equation (1) true. That is, our task is to find a prefix $s_2(1..j)$ of s_2 that can be matched with the first half of s_1 – and thereby the suffix $s_2(j+1..|s_2|)$ of s_2 that can be matched with the second half of s_1 so that the combined match is minimum. Because the position (i, j) contains the minimum edit distance for the two strings, the edit distance is included in the path of edit distances from $(|s_1|, |s_2|)$ to (0, 0).

Here is the algorithm:

```
Hirschberg(s1, s2, i1, i2, j1, j2)
    if j1 = j2  // one column case;
        for i := i1 to i2
```

```
            push the pair (i, j2) onto output;
        return;
    if  i1 = i2    // one row case;
        for j := j1 to j2
            push the pair (i2, j) onto output;
        return;
    row := (i2 + i1) / 2; // bisect substring s2(i1..i2);
    n := j2 - j1;
    if  j1  > 0
        n := n + 1;
WagnerFischer1D(current1,previous1,following s1,s2,max(1,i1), row, max
(1,j1),j2);
    if  i1 + 1 = i2    // two rows case;
      i := i2;
      j := j2;
      while (1)
         jj := j - j1 + 1;
         if  i2 = 1                    // if rows 0 or 1;
            x := previous1[jj];        //          j1      j2
            y := current1[jj - 1];     // 0:       e1 e3 ... e2(j2-j1)     - previous1
            z := previous1[jj - 1];    // 1 = row: e2 e4 ... e2(j2-j1+1) - current1
         else  x := current1[jj]       //          j1      j2
            y := following[jj - 1];    // i1 = row: e1 e3 ... e2(j2-j1)     - current1
            z := current1[jj - 1];     // i1+1:     e2 e4 ... e2(j2-j1+1) - following
         push the pair (i, j) onto tmpStack;
         if j = j1
            break;
         if(x ≤ y or z ≤ y) and (i = i2)
            i := i - 1;
         if z ≤ x or y ≤ x
            j := j - 1;
      transfer tmpStack to output;
      return;
    s11 := s1(row+1..i2 - row+1);
    s22 := s2(max(1, j1)..j2 -  max(1, j1));
```

```
    WagnerFischer1D(current2,dummy,dummy,reverse(s11),reverse(s22),1,|s11|,1,|s22|);
    Min := current1[0] + current2[n];
    col := 0;
    for k := 1 to n
        if Min > current1[k] + current2[n - k]
            Min := current1[k] + current2[n - k];
            col := k;
    x := previous1[max(1,col)];     // up
    y := current1[max(1,col) - 1]; // left
    z := previous1[max(1,col) -1]; // upper left diagonal
    if x ≤ y or z ≤ y               // choose diagonal neighbor over left
                                    neighbor
        row1 := row - 1;            // to have one less row to process
                                    in the next recursive call;
    else row1 := row;

        col := col + j1;
    if j1 > 0
        col := max(j1,col - 1);     // don't go into the area already
                                    processed;
    if z ≤ x or y ≤ x               // choose diagonal neighbor over upper
                                    neighbor
        col1 := col - 1;            // to have one less column to
                                    process;
    else col1 := col;
    Hirschberg(s1, s2, i1, row1, j1, max(j1, col1));
    Hirschberg(s1, s2, row, i2, col, j2);
```

The algorithm is called twice for each half of $s_1(i_1..i_2)$, so that it is called $2\lg|s_1|$ times in total and in each iteration it finds $(i_2 - i_1)(j_2 - j_1)$ edit distances. At the first level of recursion (very first call), the number of edit distances found is $|s_1||s_2|$; at the second level of recursion, the number equals $j|s_1|/2 + (|s_2| - j)|s_1|/2 = |s_1||s_2|/2$; at the third level of recursion, it is $j_1|s_1|/4 + (j - j_1)|s_1|/4 + (j_2 - j)|s_1|/4 + (|s_2| - j_2)|s_1|/4 = |s_1||s_2|/4$, etc., which is

$$\sum_{k=0}^{\lg|s_2|} \frac{|s_1||s_2|}{2^k} = 2\,|s_1||s_2|\left(1 - \frac{1}{|s_1|}\right) < 2|s_1||s_2|$$

in total, that is, about twice the number of operations required by `WagnerFischer()`. However, the latter uses $O(|s_1||s_2|)$ space, whereas `Hirschberg()` requires $O(|s_2|)$ space.

The algorithm uses a stack `output` to store all path positions generated by `Hirschberg()`. The stack is used later to generate an alignment.

3. An example

Let us apply `Hirschberg()` to strings s_1 = *capital* and s_2 = *apple* by calling `Hirschberg`(*capital*, *apple*, 0, 7, 0, 5). First, *capital* is bisected into *cap* and *ital* (*row* = 0 + 7 = 3) and `WagnerFischer1D`(*current*1, *previous*1, *following*, *cap*, *apple*, 1, 3, 1, 5) initializes arrays. Along the way, it generates values that `WagnerFischer()` puts in the upper half of the 2D array *D*; a full trace of execution of `WagnerFischer1D()` is as follows:

```
    a p p l e
  0 1 2 3 4 5
c 1 1 2 3 4 5
a 2 1 2 3 4 5
p 3 2 1 2 3 4      =    current1
```

Next, reverses of strings *tal* and *apple* are processed with the call `WagnerFischer1D` (*current*2, *dummy*, *dummy*, *lat*, *elppa*, 1, 3, 1, 5):

```
    e l p p a
  0 1 2 3 4 5
l 1 1 1 2 3 4
a 2 2 2 2 3 3
t 3 3 3 3 3 4
i 4 4 4 4 4 4      =    current2
```

To find the position *col* of edit distance that meets the condition (1), we use only arrays *current1* and *current*2, as in:

```
k = 0 1 2 3 4 5
    3 2 1 2 3 4  =  current1
    4 4 4 4 4 4  =  reverse(current2)
    7 6 5 6 7 8  =  current1[k] + current2[n-k] = current1[k] +
                    + reverse(current2[k])
```

We find that *Min* = 5 and *col* = 2.

At that point we know that one path to generate an alignment leads through number 1 shown in bold face in position 2 of *current*1. Having still access to *previous*1, we can determine with the same rules as used in `WagnerFisherPrint()` that this path leads to number 1 in position (2, 1). The remaining parts of the path are still undetermined. Therefore, we continue processing for strings *ca* and *a* with number 1, the first part of the path, in the lower right corner of the part of the edit matrix that corresponds to these strings (marked with pluses) and then for *pital* and *pple* with number 1, the second part of the path, in the upper left

right corner of the edit submatrix associated with the two strings (marked with asterisks); the other parts of the matrix can be disregarded from further processing:

```
    a p p l e
  + +
c + +
a + 1
p     1 * * *
i     * * * *
t     * * * *
a     * * * *
l     * * * *
```

Processing of *ca* and *a*, with *row* = 2, gives *col* = 0 and *Min* = 1. This means that the sought path goes through position (1, 0):

```
    a p p l e
  +
c 1 +
a + 1
p     1 * * *
i     * * * *
t     * * * *
a     * * * *
l     * * * *
```

From this, the call `Hirschberg` (*capital*, *apple*, 0, 0, 0, 0) amounts to the one column case and to including position(0, 0) in the path. Also, position (2, 1) marks the lower right corner and position(1, 0) marks the upper left corner of edit submatrix for next recursive invocation of `Hirschberg()`, `Hirschberg` (*capital*, *apple*, 1, 2, 0, 1), for-strings *ca* and *a*. This is another base case: the two rows case. For this case, we know that the path leads through the upper left and lower right corner, so the problem is in determining the part of the path between these corners. We must start from the lower right corner. Along the way, the positions are stored on a temporary stack and after reaching the upper left corner, they are transferred to the stack `output`. This case utilizes the array *following* which stores edit distances corresponding to the second row (the first row is in *current*1). It should be also clear why two rows case is a base case and it is not reduced to two applications of the one row case: the upper left corner has been determined by `WagnerFischer1D()` in a previous call to `Hirschberg()` and it will also be chosen in the current call leading to an infinite loop.

Now, recursion returns to strings *pital* and *pple*. This leads to processing of *pi* and *p*, another one column base case, and thus to outputting positions (3, 2) and (4, 2).

Also, `Hirschberg()` is called for the same string *tal* and string *pple*:

```
    a p p l e
  0
c 1
a   1
p     1
i     2
t     3 * * *
a     * * * *
l     * * * *
```

This, in turn, results in the call of `Hirschberg()` for *t* and *p*, and outputting (5, 2), and to the call for *al* and *ple*:

```
    a p p l e
  0
c 1
a   1
p     1
i     2
t     3
a       4 * *
l       * * *
```

This is also a two rows case that produces now positions (6, 3), (7, 4), and (7, 5). This concludes the processing by `Hirschberg()`. Now, stack `output` holds all the positions of a path corresponding with an alignment. The alignment is generated by another routine quite similar to `WagnerFischerPrint()`.

In effect, the path:

```
[7 5]  [7 4]  [6 3]  [5 2]  [4 2]  [3 2]  [2 1]  [1 0]  [0 0]
```

that represents the situation:

```
    a p p l e
  0
c 1
a   1
p     1
i     2
t     3
a       4
l         4 5
```

is used to generate the alignment:

```
capital-
-ap--ple
```

References

[1] Crochemore M., Rytter W.: *Text algorithms.* New York, Oxford University Press 1994

[2] Gusfield D.: *Algorithms on strings, trees, and sequences.* New York, Cambridge University Press 1997

[3] Hirschberg D.S.: *A linear-space algorithm for computing maximal common subsequences.* Communications of the ACM, 18, 1975, 341–343

[4] Stephen G.A.: *String searching algorithms.* Singapore, World Scientific 1994

[5] Wagner R.A., Fischer M.J.: *The string-to-string correction problem.* Journal of the ACM, 21, 1974, 168–173