FAAL: Pseudocode

July 2, 2018

```
1  max_value : array storing the highest values identified by the algorithm in the matrix
2  comp_matrix : matrix with the number of features shared by each pair of the two words; see article
3  x and y : coordinates of the matrix

4  //search for max values
5  foreach letter x of Word A do
6      foreach letter y of Word B do
7          foreach item a of the array max_val do
8              if comp_matrix[x][y] > 0 comp_matrix[x][y] Equals max_val[a] then
9                  break loop
10             if comp_matrix[x][y] < 0 then
11                 break loop
12                 //this in case some values of the comparative matrix have been set to < 0 values due to the presence of possible penalties
13             if comp_matrix[x][y] > max_val[a] then
14                 foreach item i of the array max_val by going backward do
15                     max_val[i] ← max_val[i − 1]
16                 max_val[a] ← comp_matrix[x][y]
17                 break loop
18 //store max values
19 Outerloop: foreach item i of the array max_val do
20     foreach letter x of Word A do
21         foreach letter y of Word B do
22             //A flag may be added here to deal with cases in which an excessively high number of pairs with the same top scoring highest value are present in the matrix
23             if comp_matrix[x][y] Equals max_val[i] then
24                 //Store the coordinates and position in the aligment of the identified pair as follow:
25                 define seq_data as a multidimentional object.
26                 //seq_data contains the following items:
27                 //0 : index of the matching sequence being built
28                 //1 : coord. x in the comp_matrix
29                 //2 : coord. y in the comp_matrix
30                 //3 : value of comp_matrix[x][y]
31                 //4 : index of the pair in the matching sequence
32                 //5 : index of the previous pair in the matching sequence being built, set by default to -1 for the initial pairs as they are the first values considered
33                 //6 : flag indicating the edges of the matrix have been reached
34                 seq_data ← [seq_count, x, y, comp_matrix[x][y], index, prev_index, flag_false]
```

```
                                                                    //save the high value found as first item of the head sequence:
                                                                    add seq_data to the multidimentional array sequence_head
                                                                    //save the high value found as first item of the tail sequence:
                                                                    add seq_data to the multidimentional array sequence_tail
                                                                    //Update indicators as follow:
                                                                    seq_count ← seq_count + 1
                                                                    seq_count_tail ← seq_count_tail + 1
                                                                    index ← index + 1
                                                                    index_tail ← index_tail + 1
// loop to build the head of the matching sequence (namely sequence_head), i.e. the segmente to the left of the initial max value
foreach item a of the array max_val do
    //reset array max_val[a]
    max_val[a] ← 0
foreach multidimentional item z of the array sequence_head do
    //This IF...ELSE checks if the edges of the matrix have been reached.
    //If the edges have not been reached, do as follow:
    if sequence_head[z][1], namely coord. x in the comp_matrix of the stored pair z > 0 sequence_head[z][2] namely coord. y in the comp_matrix of the stored pair z > 0 then
        //search for max values along the y branch of the L-shaped area to the left of the stored pair z
        for y = 0 to sequence_head[z][2] − 1, namely coord. y − 1 in the comp_matrix of the stored pair z do
            x ← sequence_head[z][1] − 1, namely coord. x − 1 in the comp_matrix of the stored pair z
            foreach item a of the array max_val do
                if comp_matrix[x][y] > 0 comp_matrix[x][y] Equals max_val[a] then
                    break loop
                if comp_matrix[x][y] < 0 then
                    break loop
                if comp_matrix[x][y] > max_val[a] then
                    foreach item i of the array max_val by going backward do
                        max_val[i] ←3 max_val[i − 1]
                    max_val[a] ← comp_matrix[x][y]
                    break loop
```

```
65        for x = 0 to sequence_head[z][1] − 1, namely coord. x − 1 in the
          comp_matrix of the stored pair z do
66            y ← sequence_head[z][2] − 1, namely coord. y − 1 in the
              comp_matrix of the stored pair z
67            foreach item a of the array max_val do
68                if comp_matrix[x][y] > 0 comp_matrix[x][y] Equals
                  max_val[a] then
69                    break loop
70                if comp_matrix[x][y] < 0 then
71                    break loop
72                if comp_matrix[x][y] > max_val[a] then
73                    foreach item i of the array max_val by going backward
                      do
74                        max_val[i] ← max_val[i − 1]
75                    max_val[a] ← comp_matrix[x][y]
76                    break loop
77        //Store max values
78        for x = 0 to sequence_head[z][1] − 1, namely coord. x − 1 in the
          comp_matrix of the stored pair z do
79            for y = 0 to sequence_head[z][2] − 1, namely coord. y − 1 n
              the comp_matrix of the stored pair z do
80                foreach item i of the array max_val do
81                    if comp_matrix[x][y] Equals max_val[i] then
82                        //Store the coordinates and position in the
                          aligment of the identified pair as follow:
83                        define seq_data as a multidimentional object.
84                        //As above, seq_data contains the following items:
85                        //0 : index of the head matching sequence being
                          built
86                        //1 : coord. x in the comp_matrix
87                        //2 : coord. y in the comp_matrix
88                        //3 : value of comp_matrix[x][y]
89                        //4 : index of the pair in the head matching
                          sequence
90                        //5 : index of the previous (i.e. to the right) pair in
                          the head matching sequence being built
91                        //6 : flag indicating the edges of the matrix have
                          been reached
92                        seq_data ← [seq_count, x, y, comp_matrix[x][y],
                          index, prev_index, flag_false]
93                        add seq_data to the multidimentional array
                          sequence_head
94                        //Update indicators as follow:
95                        seq_count ← seq_count + 1
96                        index ← index + 1
97        foreach item a of the array max_val do
98            //reset array max_val[a]
99            max_val[a] ← 0
```

```
100     else
101         //If the edges of the matrix for the head sequence have been
            reached, flag the indicator 6 of the pair z as follow:
102         sequence_head[z][6] ← flag_true_head
103 //=========
104 // loop to build the tail of the matching seguence, i.e. the segmente to
    the right of the initial max value
105 foreach item a of the array max_val do
106     //reset array max_val[a]
107     max_val[a] ← 0
108 for z = 0 to sequence_tail.size() − 1 do
109     //This IF...ELSE check if the edges of the matrix have been reached.
110     //If the edges have not been reached, do as follow:
111     if sequence_tail[z][1] + 1, namely coord. x + 1 in the comp_matrix
        of the stored pair z < length of Word A sequence_tail[z][2] + 1,
        namely coord. y + 1 in the comp_matrix of the stored pair z < length
        of Word B then
112         //search for max values along the x branch of the L-shaped area
            at the right of the stored pair z
113         for x = sequence_tail[z][1] + 1, namely coord. x + 1 in the
            comp_matrix of the stored pair z to length of Word A do
114             y ← sequence_tail[z][2] + 1, namely coord. y + 1 in the
                comp_matrix of the stored pair z
115             foreach item a of the array max_val do
116                 if comp_matrix[x][y] > 0 comp_matrix[x][y] Equals
                    max_val[a] then
117                     break loop
118                 if comp_matrix[x][y] < 0 then
119                     break loop
120                 if comp_matrix[x][y] > max_val[a] then
121                     foreach item i of the array max_val by going backward
                        do
122                         max_val[i] ← max_val[x − 1]
123                     max_val[a] ← comp_matrix[x][y]
124                     break loop
```

```
125     //search for max values along the y branch of the L-shaped area
        at the right of the stored pair z
126     for y = sequence_tail[z][2] + 1, namely coord. y + 1 in the
        comp_matrix of the stored pair z to length of Word B do
127         x ← sequence_tail[z][1] + 1, namely coord. x + 1 in the
            comp_matrix of the stored pair z
128         foreach item a of the array max_val do
129             if comp_matrix[x][y] > 0 comp_matrix[x][y] Equals
                max_val[a] then
130                 break loop
131             if comp_matrix[x][y] < 0 then
132                 break loop
133             if comp_matrix[x][y] > max_val[a] then
134                 foreach item i of the array max_val by going backward
                    do
135                     max_val[i] ← max_val[i − 1]
136                 max_val[a] ← comp_matrix[x][y]
137                 break loop
138     // store max values
139     for x = sequence_tail[z][1] + 1, namely coord. x + 1 in the
        comp_matrix of the stored pair z to length of Word A do
140         for y = sequence_tail[z][2] + 1, namely coord. y + 1 in the
            comp_matrix of the stored pair z to length of Word B do
141             foreach item i of the array max_val do
142                 if comp_matrix[x][y] Equals max_val[i] then
143                     //Store the coordinates and position in the
                        aligment of the identified pair as follow:
144                     define seq_data_tail as a multidimentional object.
145                     //seq_data_tail contains the following items:
146                     //0 : index of the tail matching sequence being
                        built
147                     //1 : coord. x in the comp_matrix
148                     //2 : coord. y in the comp_matrix
149                     //3 : value of comp_matrix[x][y]
150                     //4 : index of the pair in the tail matching sequence
151                     //5 : index of the previous (i.e. to the left) pair in
                        the matching tail sequence being built, which
                        corresponds to z
152                     //6 : flag indicating the edges of the matrix have
                        been reached
153                     seq_data_tail ← [seq_count_tail, x, y,
                        comp_matrix[x][y], index_tail, z, flag_false]
154                     add seq_data_tail to the multidimentional array
                        sequence_tail
155                     //Update indicators as follow:
156                     seq_count_tail ← seq_count_tail + 1
157                     index_tail ← index_tail + 1
```

foreach *item a of the array max_val* **do**
//reset array $max\_val[a]$
$max\_val[a] \leftarrow 0$
else
//If the edges of the matrix for the tail sequence have been reached, flag the indicator 6 of the pair z as follow:
$sequence\_tail[z][6] \leftarrow flag\_true\_tail$

//Build alignments
//———
//Build head sequence
//The head sequence will be built starting from the edge of the matrix and going backward up to the initial highest value.
foreach *item n in sequence_head* **do**
//find starting points for the head sequence on the edges of the matrix
if *flag indicating the edges of the matrix have been reached is positive for the left edge, namely sequence_head*[n][6] **Equals** $flag\_true\_head$ **then**
coord_x_pair_head : array containing the x coord. of the aligned head sequence of pairs (without gaps) for Word A
coord_y_pair_head : array containing the y coord. of the aligned head sequence of pairs (without gaps) for Word B
align_W_A_head : array containing the aligned head sequence of phonemes (including gaps) for Word A; phonemes are indicated by their corresponding x coord. in the comparative matrix
align_W_B_head : array containing the aligned head sequence of phonemes (including gaps) for Word B; phonemes are indicated by their corresponding y coord. in the comparative matrix
add the coord. x of pair n, namely *sequence_head*[n][1], to the array *coord_x_pair_head*
add the coord. y of pair n, namely *sequence_head*[n][2], to the array *coord_y_pair_head*
$next\_char \leftarrow$ index of the previous (i.e. to the right) pair in the head matching sequence being built, namely *sequence_head*[n][5]
foreach *item i in sequence_head by going backward* **do**
if *index of the pair i in the matching head sequence corresponds to next_char, namely sequence_head*[i][4] **Equals** *next_char* **then**
$next\_char \leftarrow$ index of the previous (i.e. to the right) pair in the head matching sequence being built, namely *sequence_head*[i][5]
add the coord. x of pair i, namely *sequence_head*[i][1], to the array *coord_x_pair_head*
add the coord. y of pair i, namely *sequence_head*[i][2], to the array *coord_y_pair_head*

$diff\_coords$: difference between coordinates x and y
$dist\_pairs$: distance between two next pairs or between a pair and the edge of the matrix counting any phoneme that are not part of any pair between them. For instance in the alignment $tkaso/tk–o$ the $dist\_pairs$ between the pairs t/t and k/k is 1, while the $dist\_pairs$ between k/k and o/o is 3

foreach *item i in* $coord\_x\_pair\_head$ **do**
    if *i is the first item* **then**
        $diff\_coords \leftarrow$ subtract coord. y of first pair of the alignment in the head sequence from coord. x of first pair of the alignment in the head sequence, namely $coord\_x\_pair\_head[first - item] - coord\_y\_pair\_head[first - item]$
    else
        $diff\_coords \leftarrow$ subtract the difference between coord. y of the pairs i and $i-1$ of the alignment in the head sequence from the difference between coord. x of the pairs i and $i-1$ of the alignment in the head sequence, namely $(coord\_x\_pair\_head[i] - coord\_x\_pair\_head[i-1]) - (coord\_y\_pair\_head[i] - coord\_y\_pair\_head[i-1])$
    //if there is no gap between the pairs, as in an alignment tk/tk, or if the gap involve the same number of unaligned phonemes in both words, like in an alignment $ta-k/t-sk$ or $t-ak/ts-k$ (which are equivalent)
    if $diff\_coords$ **Equals** 0 **then**
        if i **Equals** 0 **then**
            //if there is no gap because it is the first phoneme of the sequence
            add $coord\_x\_pair\_head[i]$ to $align\_W\_A\_head$
            add $coord\_y\_pair\_head[i]$ to $align\_W\_B\_head$
        else
            $dist\_pairs \leftarrow$ difference between coord. x of the pairs i and $i-1$ of the alignment in the head sequence, namely $coord\_x\_pair\_head[i] - coord\_x\_pair\_head[i-1]$
            //if there is no gap:
            if $dist\_pairs$ **Equals** 1 **then**
                add $coord\_x\_pair\_head[i]$ to $align\_W\_A\_head$
                add $coord\_y\_pair\_head[i]$ to $align\_W\_B\_head$

```
                    //if there is a gap involving the same number of phonemes:
                    if dist_pairs > 1 then
                        for e = 1 to dist_pairs − 1 do
                            add phoneme of Word A corresponding to the gap in Word B, namely (coord_x_pair_head[i − 1] + e), to align_W_A_head
                            add flag indicating the gap in Word A corresponding to the phoneme in Word B to align_W_A_head
                            add flag indicating the gap in Word B corresponding to the phoneme in Word A to align_W_B_head
                            add phoneme of Word B corresponding to the gap in Word A, namely (coord_y_pair_head[i − 1] + e), to align_W_B_head
                        add coord_x_pair_head[i] to align_W_A_head
                        add coord_y_pair_head[i] to align_W_B_head
                //if there is a gap either in Word A or in Word B
                if diff_coords > 0 then
                    //if total number of unaligned phonemes is greater in Word B, whether because of a single continuous gap in Word A, e.g. k − t/kat or because the gap sequence involving alternating gaps in Word A and B, e.g. k − s − t/ka − et, with the total gap corresponding to −s − /a − e, which results in more single gaps in Word A and more unaligned phonemes in Word B
                    if i > 0 then
                        dist_pairs ←
                        coord_y_pair_head[i] − coord_y_pair_head[i − 1]
                    //if dist_pairs > 1, namely if the gap sequence involves alternating gaps
                    //First process the alternating gaps
                    if dist_pairs > 1 then
                        for e = 1 to dist_pairs − 1 do
                            add phoneme of Word A corresponding to the gap in Word B, namely (coord_x_pair_head[i − 1] + e), to align_W_A_head
                            add flag indicating the gap in Word A corresponding to the phoneme in Word B to align_W_A_head
                            add flag indicating the gap in Word B corresponding to the phoneme in Word A to align_W_B_head
```

```
224                 //Then process the remaining non alternating
                    gaps/unaligned phonemes
225                 for e = |diff_coords| to e > 0 by going backward do
226                     add a flag indicating the gap to align_W_B_head
227                     add phoneme of Word A corresponding to the gap in
                        Word B, namely (coord_x_pair_head[i] − e), to
                        align_W_A_head
228                 //finally add coordinates pair:
229                 add coord_x_pair_head[i] to align_W_A_head
230                 add coord_y_pair_head[i] to align_W_B_head
231             if diff_coords < 0 then
232                 //if the gap is instead in word A
233                 if i > 0 then
234                     dist_pairs ←
                        coord_x_pair_head[i] − coord_x_pair_head[i − 1]
235                 if dist_pairs > 1 then
236                     for e = 1 to dist_pairs − 1 do
237                         add phoneme of Word A corresponding to the gap
                            in Word B, namely
                            (coord_x_pair_head[i − 1] + e), to
                            align_W_A_head
238                         add flag indicating the gap in Word A
                            corresponding to the phoneme in Word B to
                            align_W_A_head
239                         add flag indicating the gap in Word B
                            corresponding to the phoneme in Word A to
                            align_W_B_head
240                         add phoneme of Word B corresponding to the gap
                            in Word A, namely
                            (coord_y_pair_head[i − 1] + e), to
                            align_W_B_head
241                 for e = |diff_coords| to e > 0 by going backward do
242                     add a flag indicating the gap to align_W_A_head
243                     add phoneme of Word B corresponding to the gap in
                        Word A, namely (coord_y_pair_head[i] − e), to
                        align_W_B_head
244                 //finally add coordinates pair:
245                 add coord_x_pair_head[i] to align_W_A_head
246                 add coord_y_pair_head[i] to align_W_B_head
247         dist_pairs ← 0
248         diff_coords ← 0                    10
249         list_A_head : array storing the head sequences for word A
            obtained
250         list_B_head : array storing the head sequences for word B
            obtained
251         //check if the head sequence so obtained is new, and if so save it
            into the list_A_head and list_B_head arrays for later
```

```
253     foreach item g in list_A_head do
254         if list_A_head[g] Equals align_W_A_head
            list_B_head[g] Equals align_W_B_head then
255             already_stored_head ← true
256             break loop
257     if already_stored_head Equals false then
258         add align_W_A_head to list_A_head
259         add align_W_B_head to list_B_head
260         already_stored_head ← true
```

261 //build tail sequence

262 //as in the case of the head sequence, the tail sequence will be built starting from the edge of the matrix and going backward up to the initial highest value. Note that this means the sequence will be first built backward, and therefore will need to be reversed.

263 **foreach** *item n in sequence_tail* **do**

264 //find starting points for the head sequence on the edges of the matrix

265 **if** *flag indicating the edges of the matrix have been reached is positive for the right edge, namely sequence_tail*[*n*][6] **Equals** *flag_true_tail* **then**

266 *coord_x_pair_tail_bckwrd* : array containing the x coord. of the aligned tail sequence of pairs (without gaps) for Word A in backward order

267 *coord_y_pair_tail_bckwrd* : array containing the y coord. of the aligned tail sequence of pairs (without gaps) for Word B in backward order

268 *coord_x_pair_tail* : array containing the x coord. of the aligned tail sequence of pairs (without gaps) for Word A in regular order

269 *coord_y_pair_tail* : array containing the y coord. of the aligned tail sequence of pairs (without gaps) for Word B in regular order

270 *align_W_A_tail* : array containing the tail aligned sequence of phonemes (including gaps) for Word A; phonemes are indicated by their corresponding x coord. in the comparative matrix.

271 *align_W_B_tail* : array containing the tail aligned sequence of phonemes (including gaps) for Word B; phonemes are indicated by their corresponding y coord. in the comparative matrix.

272 add the coord. x of pair n, namely *sequence_tail*[*n*][1], to the array *coord_x_pair_tail_bckwrd*

273 add the coord. y of pair n, namely *sequence_tail*[*n*][2], to the array *coord_y_pair_tail_bckwrd*

274 *next_char_tail* ← index of the previous (i.e. to the right) pair in the head matching sequence being built, namely *sequence_tail*[*n*][5]

//Build tail sequence backward

foreach *item i in sequence_tail by going backward* **do**

  if *index of the pair i in the tail matching sequence corresponds to next_char, namely sequence_tail*$[i][4]$ **Equals** *next_char* **then**

    next_char ← index of the previous (i.e. to the right) pair in the tail matching sequence being built, namely *sequence_tail*$[i][5]$

    add the coord. x of pair i, namely *sequence_tail*$[i][1]$, to the array *coord_x_pair_tail_bckwrd*

    add the coord. y of pair i, namely *sequence_tail*$[i][2]$, to the array *coord_y_pair_tail_bckwrd*

//reverse tail sequence from backward to the correct direction

foreach *item f in coord_x_pair_tail_bckwrd by going backward* **do**

  add *coord_x_pair_tail_bckwrd*$[f]$ to *coord_x_pair_tail*

  add *coord_y_pair_tail_bckwrd*$[f]$ to *coord_y_pair_tail*

$diff\_coords$: difference between coordinates x and y, as above

dist_pairs : distance between two next pairs or between a pair and the edge of the matrix counting any phoneme that are not part of any pair between them, as above. For instance in the alignment *tkaso*/*tk–o* the *dist_pairs* between the pairs t/t and k/k is 1, while the *dist_pairs* between k/k and o/o is 3.

foreach *item i in coord_x_pair_tail* **do**

  if *i is the first item* **then**

    $diff\_coords$ ← subtract coord. y of first pair of the alignment in the tail sequence from coord. x of first pair of the alignment in the tail sequence, namely $coord\_x\_pair\_tail[first - item] - coord\_y\_pair\_tail[first - item]$

  else

    $diff\_coords$ ← subtract the difference between coord. y of the pairs i and $i-1$ of the alignment in the tail sequence from the difference between coord. x of the pairs i and $i-1$ of the alignment in the tail sequence, namely $(coord\_x\_pair\_tail[i] - coord\_x\_pair\_tail[i-1]) - (coord\_y\_pair\_tail[i] - coord\_y\_pair\_tail[i-1])$

```
292     //if there is no gap between the pairs, as in an alignment tk/tk, or if the gap involve the same number of unaligned phonemes in both words, like in an alignment ta − k/t − sk or t − ak/ts − k (which are equivalent)
293     if diff_coords Equals 0 then
294         if i Equals 0 then
295             //if there is no gap because it is the first phoneme of the sequence
296             add coord_x_pair_tail[i] to align_W_A_tail
297             add coord_y_pair_tail[i] to align_W_B_tail
298         else
299             dist_pairs ← difference between coord. x of the pairs i and i − 1 of the alignment in the tail sequence, namely coord_x_pair_tail[i] − coord_x_pair_tail[i − 1];
300             //if there is no gap:
301             if dist_pairs Equals 1 then
302                 add coord_x_pair_tail[i] to align_W_A_tail
303                 add coord_y_pair_tail[i] to align_W_B_tail
304             //if there is a gap involving the same number of phonemes:
305             if dist_pairs > 1 then
306                 for e = 1 to dist_pairs − 1 do
307                     add phoneme of Word A corresponding to the gap in Word B, namely (coord_x_pair_tail[i − 1] + e), to align_W_A_tail
308                     add flag indicating the gap in Word A corresponding to the phoneme in Word B to align_W_A_tail
309                     add flag indicating the gap in Word B corresponding to the phoneme in Word A to align_W_B_tail
310                     add phoneme of Word B corresponding to the gap in Word A, namely (coord_y_pair_tail[i − 1] + e), to align_W_B_tail
311                 add coord_x_pair_tail[i] to align_W_A_tail
312                 add coord_y_pair_tail[i] to align_W_B_tail
```

```
313     //if there is a gap either in Word A or in Word B
314     if diff_coords > 0 then
315         //if total number of unaligned phonemes is greater in
            Word B, weather because of a single continuous gap in
            Word A, e.g. k − t/kat or because the gap sequence
            involving alternating gaps in Word A and B, e.g.
            k − s − t/ka − et, with the total gap corresponding to
            −s − /a − e, which results in more single gaps in Word A
            and more unaligned phonemes in Word B
316         if i > 0 then
317             dist_pairs ←
                coord_y_pair_tail[i] − coord_y_pair_tail[i − 1]
318         //if dist_pairs > 1, namely if the gat sequence involves
            alternating gaps
319         //First process the alternating gaps
320         if dist_pairs > 1 then
321             for e = 1 to dist_pairs − 1 do
322                 add phoneme of Word A corresponding to the gap
                    in Word B, namely (coord_x_pair_tail[i − 1] + e),
                    to align_W_A_tail
323                 add flag indicating the gap in Word A
                    corresponding to the phoneme in Word B to
                    align_W_A_tail
324                 add flag indicating the gap in Word B
                    corresponding to the phoneme in Word A to
                    align_W_B_tail
325                 add phoneme of Word B corresponding to the gap
                    in Word A, namely (coord_y_pair_tail[i − 1] + e),
                    to align_W_B_tail
326         //Then process the remaining non alternating
            gaps/unaligned phonemes
327         for e = |diff_coords| to e > 0 by going backward do
328             add a flag indicating the gap to align_W_B_tail
329             add phoneme of Word A corresponding to the gap in
                Word B, namely (coord_x_pair_tail[i] − e), to
                align_W_A_tail
330         //finally add coordinates pair
331         add coord_x_pair_tail[i] to align_W_A_tail
332         add coord_y_pair_tail[i] to align_W_B_tail
```

if $diff\_coords < 0$ **then**

//if the gap is in word A

if $i > 0$ **then**

$dist\_pairs \leftarrow$
$coord\_x\_pair\_tail[i] - coord\_x\_pair\_tail[i-1]$

if $dist\_pairs > 1$ **then**

for $e = 1$ **to** $dist\_pairs - 1$ **do**

add phoneme of Word A corresponding to the gap in Word B, namely $(coord\_x\_pair\_tail[i-1] + e)$, to $align\_W\_A\_tail$

add flag indicating the gap in Word A corresponding to the phoneme in Word B to $align\_W\_A\_tail$

add flag indicating the gap in Word B corresponding to the phoneme in Word A to $align\_W\_B\_tail$

add phoneme of Word B corresponding to the gap in Word A, namely $(coord\_y\_pair\_tail[i-1] + e)$, to $align\_W\_B\_tail$

for $e = |diff\_coords|$ **to** $e > 0$ *by going backward* **do**

add a flag indicating the gap to $align\_W\_A\_tail$

add phoneme of Word B corresponding to the gap in Word A, namely $(coord\_y\_pair\_tail[i] - e)$, to $align\_W\_B\_tail$

//finally add coordinates pair

add $coord\_x\_pair\_tail[i]$ to $align\_W\_A\_tail$

add $coord\_y\_pair\_tail[i]$ to $align\_W\_B\_tail$

//In contrast with the head sequence, which always ends with a match (corresponding to the starting highest value), the tail sequence may end with a gap. This gap and the corresponding phoneme(s) are added to the sequence as follow:

//If the gap is in word B

if *the coord. x of the last pair of the tail sequence, namely the last item in* $align\_W\_A\_tail$*, is smaller than the number of phonemes of Word A* **then**

for f = *coord. x of the last pair* $+1$ **to** *last phoneme of Word A* **do**

add f to $align\_W\_A\_tail$

add flag indicating the gap in Word B corresponding to the phoneme in Word A to $align\_W\_B\_tail$

//If the gap is in word A

if *the y coord. of the last pair of the tail sequence, namely the last item in* $align\_W\_B\_tail$*, is smaller than the number of phonemes of Word B* **then**

for f = *coord. y of the last pair* $+1$ **to** *last phoneme of Word B* **do**

add flag indicating the gap in Word A corresponding to the phoneme in Word B to $align\_W\_A\_tail$

add f to $align\_W\_B\_tail$

```
        dist_pairs ← 0
        diff_coords ← 0
        list_A_tail : array storing the tail sequences for word A obtained
        list_B_tail : array storing the tail sequences for word B obtained
        //check if the tail sequence so obtained is new, and if so save it
          into the list_A_tail and list_B_tail arrays for later
        already_stored_tail ← false
        foreach item g in list_A_tail do
            if list_A_tail[g] Equals align_W_A_tail list_B_tail[g]
              Equals align_W_B_tail then
                already_stored_tail ← true
                break loop
        if already_stored_tail Equals false then
            add align_W_A_tail to list_A_tail
            add align_W_B_tail to list_B_tail
            already_stored_tail ← true
//join initial and final sequences
list_matches_A : array storing the valid alignments obtained by the
  algorithm for Word A
list_matches_B : array storing the corresponding valid alignments
  obtained by the algorithm for Word B
foreach item e in list_A_head do
    foreach item i in list_A_tail do
        comb_head_tail_A : array in which the combined alignment of
          head and tail for Word A are stored
        comb_head_tail_B : array in which the combined alignment of
          head and tail for Word B are stored
        //if the x coord. of the last phoneme of the head sequence of A
          corresponds to the x coord. of the first phoneme of the tail
          sequence of A and the y coord. of the last phoneme of the head
          sequence of B corresponds to the y coord. of the first phoneme of
          the tail sequence of B, namely:
        if last value stored in item e of list_A_head Equals first value
          stored in item i of list_A_tail last value stored in item e of
          list_B_head Equals first value stored in item i of list_B_tail
          then
            //add head sequences to comb_head_tail_A
            foreach item a in list_A_head[e] do
                add list_A_head[e][a] to comb_head_tail_A
                add list_B_head[e][a] to comb_head_tail_B
```

```
387         / /add tail sequences to comb_head_tail_A. Note that the
              first item has to be left off, as it is the same as the last of the
              head sequence, and therefore it has already been added.
388         foreach item a in list_A_tail[i], except first item do
389             add list_A_tail[i][a] to comb_head_tail_A
390             add list_B_tail[i][a] to comb_head_tail_B
391         //check if it is a new alignment, and is so store it in
              list_matches_B and list_matches_B
392         new_item ← true
393         if list_matches_A and list_matches_B are empty then
394             //add the first alignment
395             add comb_head_tail_A to list_matches_A
396             add comb_head_tail_B to list_matches_B
397         else
398             foreach item m in list_matches_A do
399                 //check if the same alignment has already been stored
400                 if list_matches_A[m] Equals comb_head_tail_A
                    list_matches_B[m] Equals comb_head_tail_B
                    then
401                     new_item ← false
402             //if it is a new alignment, store it
403             if new_item Equals true then
404                 add A_pre_post to list_matches_A
405                 add B_pre_post to list_matches_B
406 //The results of the algorithm, namely the alignments obtained through
    it, are stored in the arrays:
407 list_matches_A for Word A
408 list_matches_B for Word B
```