Dynamic Programming and Optimal Sequence Alignment

David M. Trujillo, TRUCOMP, Fountain Valley, CA.

June, 2004

Abstract

By casting the sequence alignment problem into a routing process, the method of dynamic programming can be used to solve the optimal routing problem and hence, the optimal sequence alignment problem. This note explains how the simplest alignment problem can be cast into a routing one and then a description of dynamic programming as applied to the optimal routing problem.

The Sequence Alignment Problem

The alignment of two sequences involves a spacing of the symbols to produce a correspondence between the two sequences. For example, one particular alignment of the two sequences AAGCAAA and AGCTACA is: (with a dash _ denoting a space)

```
A _ A _ G C A  _ A A
A G _ C _ T _   A C A
```

Another possible alignment is

```
A A G C A A _ A
A _  G C T A C A
```

In the second alignment more symbols match(5) between the sequences than in the first (2). The optimization problem is to find, from all possible alignments of the sequences, the one that produces the **maximum** number of matches between the

two sequences.

Alignment as a Routing Process

One way to solve the optimization problem is to cast the alignment as a routing process and then use the method of dynamic programming to solve the optimal routing problem. To illustrate how a routing process can produce alignments, consider the two sequences AGA and GA. The figure below shows a routing process that starts from the upper left hand corner and ends at the lower right hand corner. The sequences are used to create a two dimension grid with one sequence defining the columns and the other the rows. Moving from one state to another along any leg generates a portion of an alignment, either a space and a symbol or two symbols. The portions associated with each leg are identified in the figure.

Starting at the upper left hand corner (initial state), the route at each state has three choices (policy):

1) go Right

2) go Down

or 3)go Diagonally

Suppose it is decided that the first choice is to go Right, then the process generates the first part of the alignment associated with that leg

_
A

This takes us to the next state and if the second decision is to go diagonally, the process then generates the second part of the alignment

_ G

A G

The process continues, making a decision at each state, until the lower right hand state is reached.

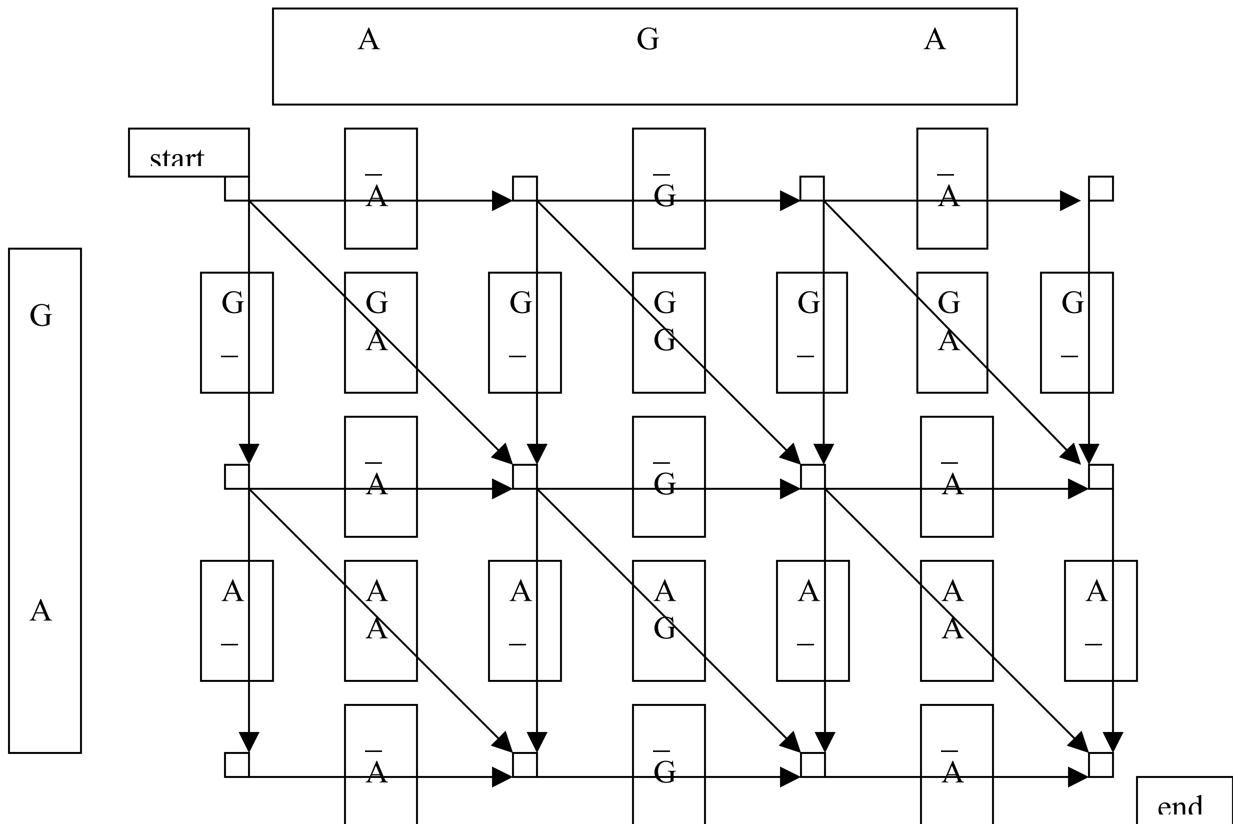


Figure 1 A Routing Process that Generates Alignments

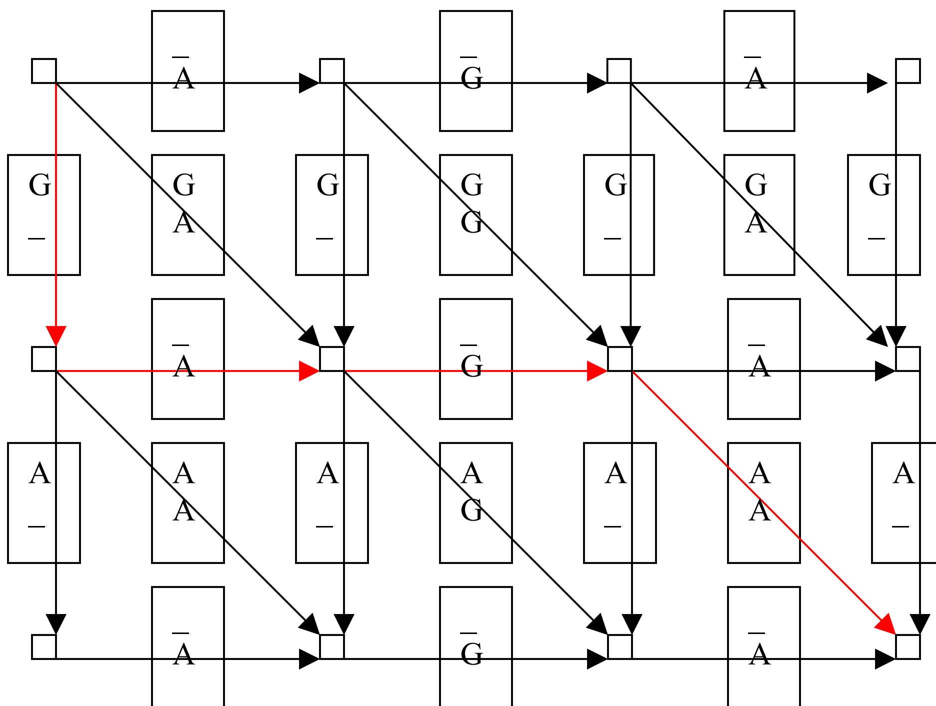
As another example, consider a Policy of decisions(in red) shown in the figure below

go Down

go Right

go Right

go Diagonally



This produces the alignment

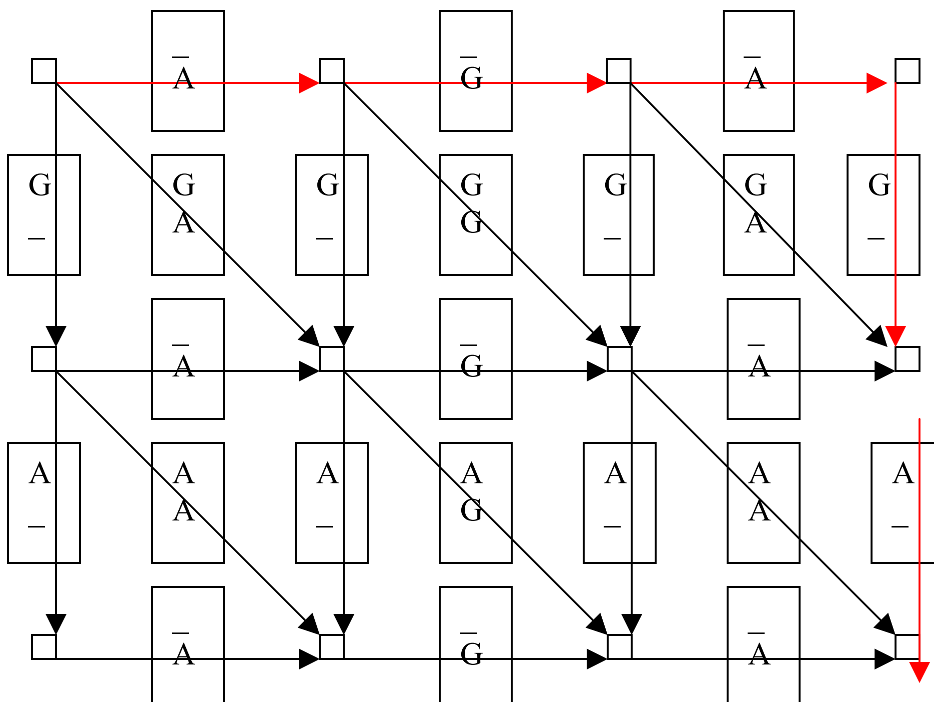
G _ _ A
_ A G A

Consider still another Policy of decisions, shown in red below, of

go Right
go Right
go Right
go Down
go Down

This would give the alignment

_ _ _ G A
A G A _ _



Using this routing process, all of the possible alignments can be produced. For most of the states there are the 3 choices: Right, Down, and Diagonally. However, as seen in the figure, the states on the extreme right edge have only 1 choice, **Down**. And for the states on the extreme lower edge, there is only 1 choice, **Right**.

In order to complete the routing process it is only necessary to assign a cost for each leg of the route. The optimization problem then becomes one of finding the route that maximizes the cost. This could just as easily be one that minimizes cost, but it seems more natural to maximize the number of symbols that match. For the sequence alignment problem, one set of costs are zero (0) for a leg that has a dash, one (1) for a leg with two mismatched symbols, and two (2) for one with matched symbols. These costs produces an alignment with the most matches.

For example, applying the above costs to the legs gives the figure below. And, for the policy of following the red arrows, the costs would be

Diagonal	Cost of 1
Right	Cost of 0
Diagonal	Cost of 2

For a total cost of 1 + 0 + 2 = 3. Of course , the problem is to find the policy (route) that maximizes this cost.

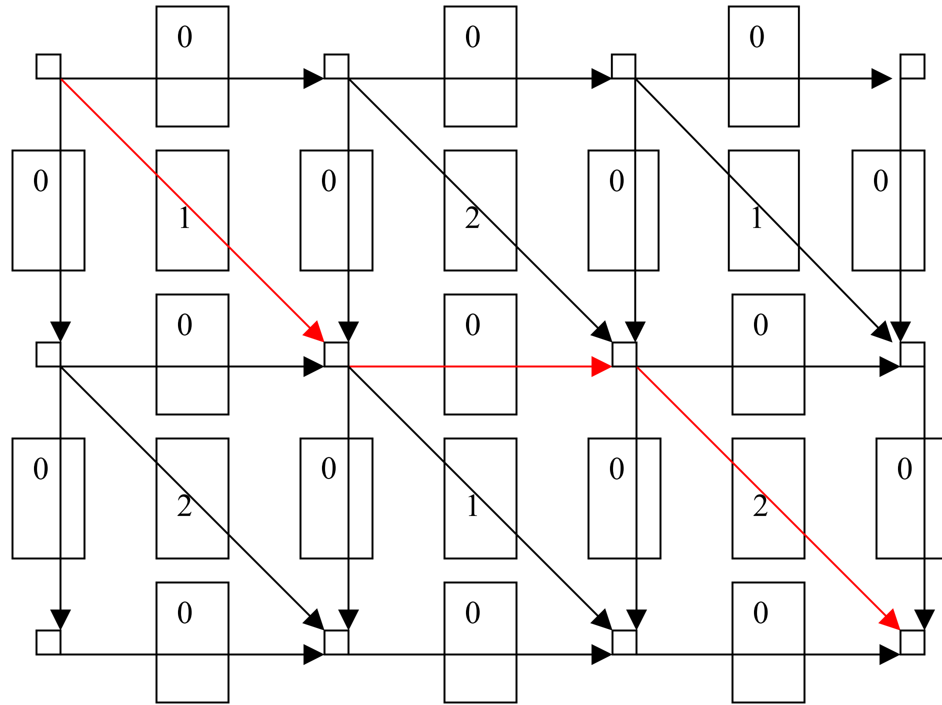


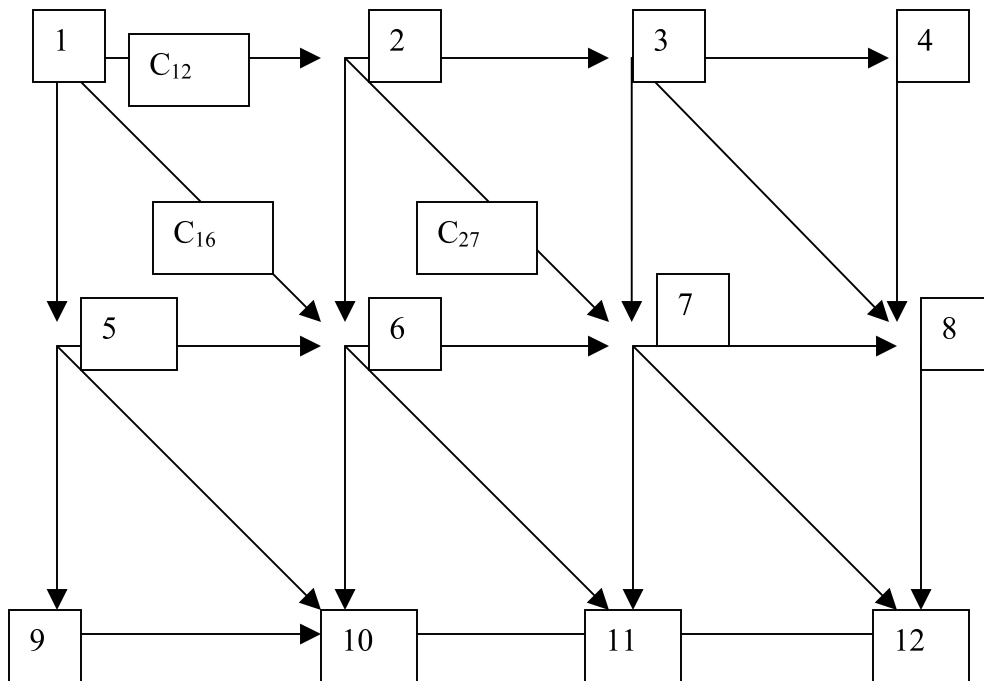
Figure 2 Cost Assignments for each Leg

With these cost assignments, the sequence alignment problem is cast into an optimal routing problem which can be solved very efficiently with dynamic programming.

For more information on this extraordinary relationship between optimal sequence alignment and the routing problem, see the references Needleman and Wunsch and Myers and Miller.

Dynamic Programming and the Optimal Routing Problem

In this section, the method of dynamic programming will be used to solve the routing problem. The general routing problem can be described by first assigning an id to each state. In the figure below the states have been assigned ids from 1 through 12. State 1 is at the upper left hand corner and state 12 is at the lower right hand corner. The cost of going along any particular leg from state i to state j is C_{ij}. The possible paths are denoted by arrows.



The problem statement is: Starting at state 1 and ending at state 12, find the route that maximizes the sum of the accrued costs. At each state the choice for the route is either right, down, or diagonally (except at the right and bottom edges). The cost of traveling each leg is Cij. Thus, the total cost of any particular route is the sum of the Cij's.

Dynamic programming approaches this problem by dividing the overall optimization problem into a smaller number of optimal problems. It

then uses previous optimal solutions to solve for more optimal solutions. To this end, let us define the following function Fi.

Fi is the cost of starting at any state i and ending at state 12, **using the optimal policy** .

In other words, Fi is the maximum cost if one started at state i and traveled to state 12 (the end of the route), using the best route. Notice that there are two functions defined: the optimal cost and the optimal policy. The optimal policy is the collection of decisions at each state. The optimal policy at state i will be denoted by Pi.

For example, starting at state 11, the optimal policy is Right, since there is no other choice, and the optimal cost is $C_{11\text{-}12}$

$$F11 = C_{11\text{-}12}$$

and $$P11 = \text{Right}$$

Incidentally, F12 = 0 since starting and ending at state 12 contributes no cost and there is no policy.

Starting at state 8, the optimal policy is Down, again there is no other choice for this state, and the optimal cost is $C_{8\text{-}12}$

$$F8 = C_{8\text{-}12}$$

$$P8 = \text{Down}$$

Now consider starting at state 7. For this state there are 3 choices: Right takes us to state 8, Down takes us to state 11, and Diagonally takes us to state 12. We need to decide which of these is the optimal policy, i.e. which one maximizes the cost. If we decide to go Right we arrive at state 8, where we have already determined the optimal cost in going from state 8 to 12 (F8). Therefore the cost of going Right is the sum of the cost going from 7 to

8 plus the cost of going from 8 to the end

$$C_{7\text{-}8} + F8 \qquad \text{(Right)}$$

Similarly, the cost of going Down is

$$C_{7\text{-}11} + F11 \qquad \text{(Down)}$$

and the cost of going diagonally is

$$C_{7\text{-}12} + F12 \qquad \text{(Diagonally)}$$

Since F7 represents the maximum cost starting from state 7 it must then be the maximum of the three costs above. In dynamic programming notation:

$$F7 = \text{MAX} \; \{C_{7\text{-}8} + F8,\; C_{7\text{-}11} + F11,\; C_{7\text{-}12} + F12\}$$

and the optimal policy P7 (right, down, or diagonally) is the one associated with the maximum of the three choices.

The equation above represents the essence of dynamic programming in that:

(1) It constructs an optimal solution from previous optimal solutions.

(2) It finds optimal solutions for all the states including the one of interest, F1, the optimal solution starting from state 1 and ending in state 12.

(3) It defines the optimal policies Pi for all the states.

(4) It constructs the optimal solutions starting from the end of the process and works backwards.

(5) The optimal route is found by starting at state 1 and following the optimal policies forward to the end.

The optimal solution starting at state 1 is found after all the Fi's and Pi's are calculated. This forward pass only uses the policies and starts with P1. For example, if P1 = Right, then this takes us to state 2 where P2 gives the optimal policy for state 2. If P2 = Diagonally, then this takes us to state 7, where P7 holds the optimal policy for starting at state 7. This process

continues until the end state is reached.

The functional equation is an application of the Principle of Optimality developed by mathematician Richard Bellman (see references 3 and 4).

Principle of Optimality: *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

For this routing problem this Principle of Optimality is expressed as

$Fi = \text{MAX} \ \{C_{ij} + Fj, \ C_{ik} + Fk, \ C_{im} + Fm\}$

and Pi = the decision associated with the maximum, i.e. going to to state j, k, or m.

Let us determine the optimal costs and decisions for the assigned costs shown in figure 2. As outlined above, the dynamic programming process starts at the end state 12, where

$F12 = 0$ and $P12$ = irrelevant

Since determining the next optimal costs requires a previously determined optimal cost it is natural for this routing process to proceed backward along each row. Thus the next state is

$F11 = \{ C_{11\text{-}12} + F12\}$ P11 = Right

since for this state there is no choice and the decision must be to go right.

$F11 = 0$ P11 = Right

Proceeding backwards along the bottom edge gives

$F10 = \{ C_{10\text{-}11} + F11\} = 0$ P10 = Right

$F9 = \{ C_{9\text{-}10} + F10\} = 0$ P9 = Right

Moving up one row puts us at state 8, which also has no choice but to go down, thus

$F8 = \{ C_{8-12} + F12\} = 0 \qquad P8 = \text{Down}$

The next state 7 offers the first choices, namely

$F7 = MAX \{C_{7-8} + F8, C_{7-11} + F11, C_{7-12} + F12 \}$

or $F7 = MAX \{0 + 0, 0 + 0, 2 + 0 \}$

$F7 = 2 \qquad P7 = \text{Diagonally}$

For state 6

$F6 = MAX \{C_{6-7} + F7, C_{6-10} + F10, C_{6-11} + F11 \}$

$F6 = MAX \{ 0 + 2, 0 + 0, 1 + 0 \}$

$F6 = 2 \qquad P6 = \text{Right}$

State 5

$F5 = MAX \{C_{5-6} + F6, C_{5-9} + F9, C_{5-10} + F10 \}$

$F5 = MAX \{ 0 + 2, 0 + 0, 2 + 0 \}$

For this state there are two decisions that give the same optimal cost, right and diagonally. It does not matter which is chosen so arbitrarily choosing Right gives

$F5 = 2 \qquad P5 = \text{Right}$

Moving up to the last row puts us at state 4 which only has the decision to go down

$F4 = \{C_{4-8} + F8 \}$

$F4 = \{ 0 + 0 \} = 0 \qquad P4 = \text{Down}$

For state 3

$F3 = MAX \{C_{3-4} + F4, C_{3-7} + F7, C_{3-8} + F8 \}$

$F3 = MAX \{ 0 + 0, 0 + 2, 1 + 0 \}$

$F3 = 2 \qquad P3 = \text{Down}$

For state 2

$F2 = MAX \{C_{2-3} + F3, C_{2-6} + F6, C_{2-7} + F7 \}$

$F2 = MAX \{ 0 + 2, 0 + 2, 2 + 2 \}$

F2 = 4 P2 = Diagonally

Finally the beginning state 1

$$F1 = MAX\ \{C_{1\text{-}2} + F2,\ C_{1\text{-}5} + F5,\ C_{1\text{-}6} + F6\}$$

$$F1 = MAX\ \{0 + 4,\ 0 + 2,\ 1 + 2\}$$

F1 = 4 P1 = Right

Let us **summarize** the costs and decisions for all the states

F1 = 4	P1 = Right
F2 = 4	P2 = Diagonally
F3 = 2	P3 = Down
F4 = 0	P4 = Down
F5 = 2	P5 = Right
F6 = 2	P6 = Right
F7 = 2	P7 = Diagonally
F8 = 0	P8 = Down
F9 = 0	P9 = Right
F10 = 0	P10 = Right
F11 = 0	P11 = Right
F12 = 0	P12 = Not Required

Thus in order to **construct the optimal route** starting from point 1, we start with the optimal decision at state 1

P1 = Right

which takes us to state 2 where the optimal decision is

P2 = Diagonally

which takes us to state 7 where

P7 =Diagonally

which takes us to the end state 12. Notice that constructing the optimal route

only requires the optimal decisions.

Discussion

The most simple alignment problem was used to illustrate how dynamic programming can be used. There can be variations on this problem, mostly by using different sets of costs assigned to the various legs. This does not change the dynamic programming algorithm at all. Another aspect to be explored later is to use an iterative dynamic programming to attack large sequences. The above method could easily solve sequences of thousands, but would have difficulties if the sequences had 100,000 symbols.

A listing of a program written on the ideas above is included at the end of this note.

References

(1) Needleman,S.B. and C.D. Wunsh, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins”, J.Mol.Biol, 48, 443-453, 1970.

(2) Myers.E. and W.Miller ,” Optimal Alignments in Linear Space”, CABIOS, 4, 11-17 , 1988.

(3) Bellman, R. and Dreyfus, S, Applied Dynamic Programming, Princeton University Press, New Jersey, 1962.

(4) Trujillo, D.M. and H.R. Busby, Practical Inverse Analysis in Engineering, CRC Press, Boca Raton, Florida, 1997.

```
/*  Program: DP_sequence.c
 *  Author:  David M. Trujillo, TRUCOMP, Fountain Valley, CA.
 *  Purpose: Dynamic Programming solution to
 *           the Optimal Sequence Alignment Problem
 *
 *  Input:   A file with two sequences, each on a line. For example
 *          GGATCATCGAA
 *          GAACTT
 *  Output:  The optimal alignment, that maximizes the alignments,e.g.
 *           GGATCATCGAA
 *           _GA__A_C_TT
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXBUFF 256     // maximum buffer length
#define MAXSIZE 70000   // maximum array length


int maximum( int a, int b, int c);

//*********************MAIN PROGRAM ********************

void main( void )
{
  FILE *in_file, *io_file;
  char  buffer[MAXBUFF];
  char  sequence1[MAXBUFF];
  char  sequence2[MAXBUFF];
  int   len1,len2;      // lengths of the sequences
  int   M,N;            // number of horizontal and vertical grids
  int   MTOTAL;         // total number of states
  int   i,j,k;
  int   decision;
  int   dcost[MAXSIZE];  // the diagonal costs for each state
                         // the horizontal and vertical costs are zero
  int   f[MAXSIZE];      // costs start from  i and using the optimal
                         // policies p[i]
  int   p[MAXSIZE];



// open the input file

  printf("\n\nPlease enter the INPUT file name.\n");
  gets(buffer);

  if(( in_file = fopen( buffer, "r")) == NULL )
   {
    printf("\n\nError opening input file.\n");
    exit(99);
   }
```

```
//open an output file

  printf("\n\nPlease enter the OUTPUT file name.\n");
  gets(buffer);
  if(( io_file = fopen( buffer, "w+")) == NULL )
   {
    printf("\n\nError opening output file.\n");
    exit(97);
   }


// read in the sequences from the input file.
// first sequence1, then sequence2.


  if( fgets(sequence1, MAXBUFF, in_file) == NULL)
  {
   printf(" Error in reading sequence 1\n");
   exit(96);
  }
  else
  {
   fprintf(io_file," Sequence 1 \n %s\n",sequence1);
  }

  if( fgets(sequence2, MAXBUFF, in_file) == NULL)
  {
   printf(" Error in reading sequence 2\n");
   exit(95);
  }
  else
  {
   fprintf(io_file," Sequence 2 \n %s\n",sequence2);
  }


// determine the lengths of the sequences

  len1 = strlen(sequence1)-1;
  len2 = strlen(sequence2)-1;

// the grid representing the route is two dimensional with
// sequence1 as the horizontal axis and sequence2 as the vertical.
// the number of states in the horizontal axis is M = len1 +1,
// which includes the far right edge. Similarly the vertical
// axis is N = len2 +1, which includes the bottom edge.

  M = len1 + 1;
  N = len2 + 1;

// the total number of states is MTOTAL. The state will be numbered
// consecutively starting from the upper left hand corner
// along each row.

  MTOTAL = M*N;
```

```
  for( i=1; i<= MTOTAL; i++)
  {
   dcost[i] = 0;
  }

// fill in the diagonal costs, equal to 1 if dissimilar
// and 2 if similar.
// fill in 1 row at a time, starting at the top

  k=1;      // index for states

  for( i=1; i<N; i++)  // row i
  {
   for( j=1; j<=M; j++) // column j
   {
    if( k%M != 0)   // right edge has no diagonal
    {
    dcost[k] = 1;
    if( sequence2[i-1] == sequence1[j-1] ) dcost[k] = 2;
    }
    k = k+1;
   }
  }

// working backwards, find the optimal costs and policies.
// end state is zero and end policy is not used.

  f[MTOTAL] = 0;
  p[MTOTAL] = 0;

// the decisions making up the optimal policy
// will be assigned the following values
//        right = 1
//        down  = 2
//     diagonal = 3

// bottom edge(row) of the two dimensional grid is special in that
// the route must move right( no choice).


  for( k=MTOTAL-1; k>MTOTAL-M; k--)  // bottom edge row
  {
    f[k] =  0 + f[k+1];  // cost of each leg = 0
    p[k] =  1;           // optimal decision = right
  }

// starting at next to last row, work backwards along each row
// find the optimal costs and policies.
// the index k will identify the state. offset M locates
// next adjacent states.

  for( k=MTOTAL-M; k>=1; k--)
  {

// check to see if k is on the right edge
```

```
   if(k % M == 0)
   {
    f[k] = f[k+M];  // actually 0 + f[k+M]
    p[k] = 2;       // must go down
   }
   else
   {

//        costs are:
//                   right= 0+f[k+1]         maximum returns 1
//                   down = 0+f[k+M]                 returns 2
//             diagonally = dcost[k]+f[k+M+1]        returns 3

// find the decision that maximizes f[k]
   p[k] = maximum( f[k+1], f[k+M], dcost[k] + f[k+M+1] );

// place the maximum into f[k]

   f[k] = f[k+1]; // default p[k]=1
   if(p[k] == 2 ) f[k]=f[k+M];
   if(p[k] == 3 ) f[k]=dcost[k] + f[k+M+1];

   } // end of if

  }  // end of for


// print out optimal costs and policies

  for( k=1; k<= MTOTAL; k++)
  {
   fprintf(io_file," state %d  cost  %d  decision %d \n",k,f[k],p[k]);
  }

// using the optimal policies, print out the optimal alignments,
// by following the optimal policy.

  fprintf(io_file,"  \n  Optimal Alignment \n\n");

// print out sequence1. A symbol is advanced if p[k]=right or diagonal
// otherwise print a blank(dash).

  k = 1; // index on sequence1[]
  i = 1; // index on policy
  decision = p[1]; // first  optimal decision

  while( k< M)
  {
   if(decision == 2)  // go down does not advance sequence1
   {
    fprintf(io_file,"_");
   }
   else
   {
    fprintf(io_file,"%c",sequence1[k-1]);
    k = k+1;  // advance sequence1
   }
```

```
// find next state that was directed by p[]
   if(decision == 1) i = i+1;    // go right
   if(decision == 2) i = i+M;    // go down
   if(decision == 3) i = i+M+1;  // go diagonally

   decision = p[i];     // next state and next decision

  }

  fprintf(io_file,"\n");  // end of sequence line

// print out sequence2. a symbol is advanced if p[k]=down or diagonal
// otherwise print a blank

  k = 1; // index on sequence2[]
  i = 1; // index on policy
  decision = p[1]; // first  decision

  while( k< N)
  {
   if(decision == 1)  // go right- does not advance sequence2
   {
    fprintf(io_file,"_");
   }
   else
   {
    fprintf(io_file,"%c",sequence2[k-1]);
    k = k+1;  // advance sequence2
   }

// find next state
   if(decision == 1) i = i+1;    // go right
   if(decision == 2) i = i+M;    // go down
   if(decision == 3) i = i+M+1;  // go diagonally

   decision = p[i];     // next state and next decision
  }

  fprintf(io_file,"\n");  // end of sequence line

  printf("  NORMAL TERMINATION\n");

  exit(0);

} // end of main

//***************************************************
int maximum( int a, int b, int c)
{
 int i;

  i = 1; // maximum is a

  if( b > a )
  {
   i = 2;
```

```
   if( c > b ) i = 3;
   return i;
  }
  else
  {        // a is > b check c
   if( c > a ) i = 3;
   return i;
  }

}
```