More Edit Distance Algorithms

Lecture 6

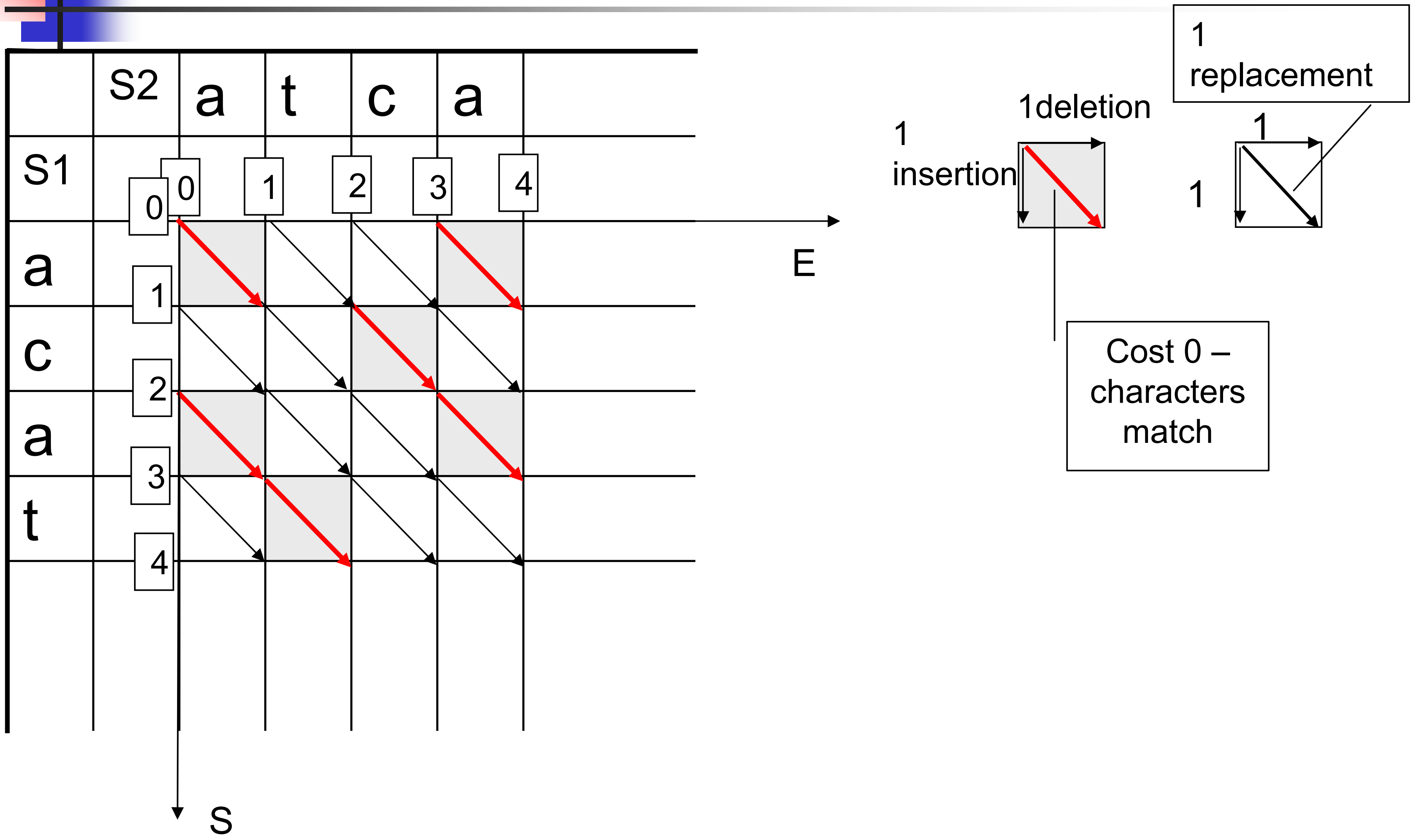
Edit distance

- The *edit distance* between two strings is defined as the minimum number of edit operations needed to transform one string into another
- Edit operations are
  - insertion of 1 symbol,
  - deletion of 1 symbol or
  - replacement (substitution) of 1 symbol

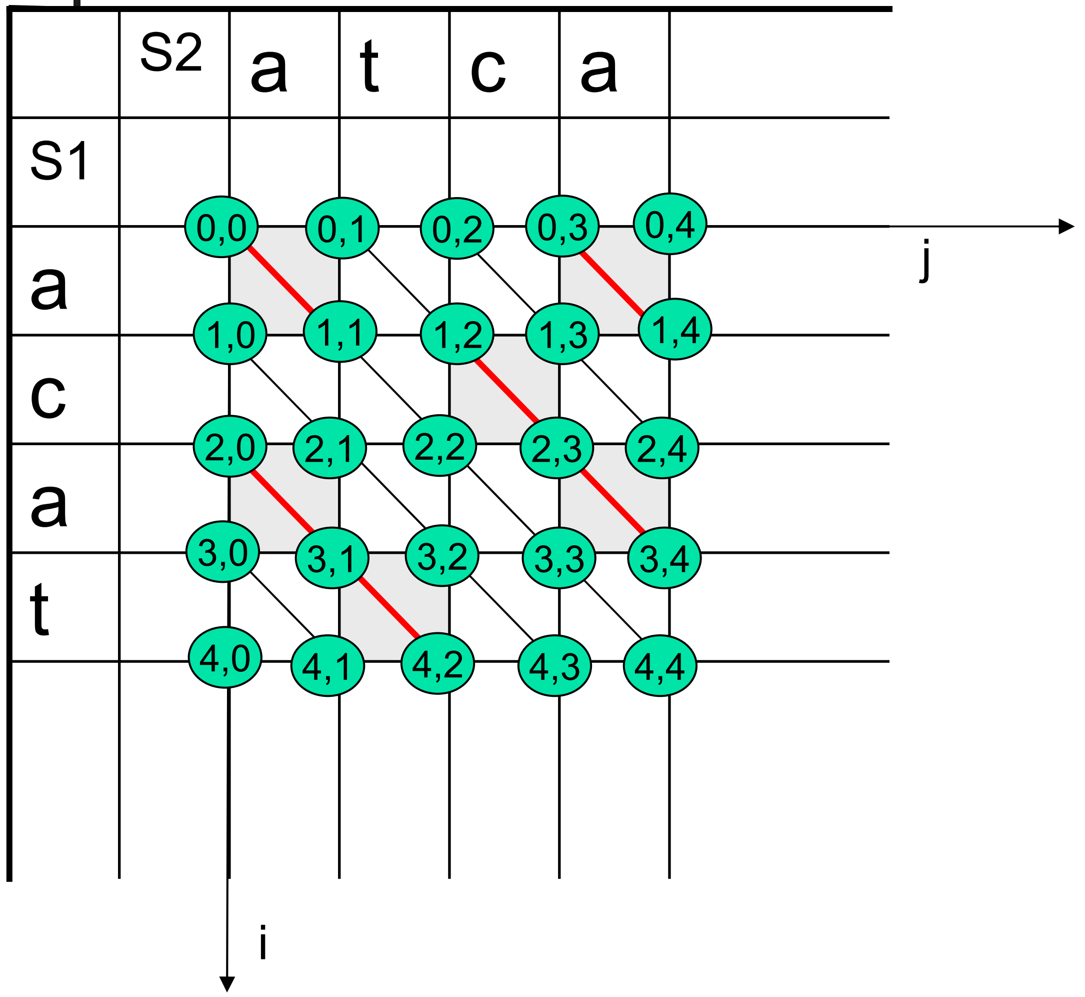
The edit distance problem

- Compute the edit distance between two strings along with a sequence of the operations which describe the transformation

Analogy with the cheapest path in a grid



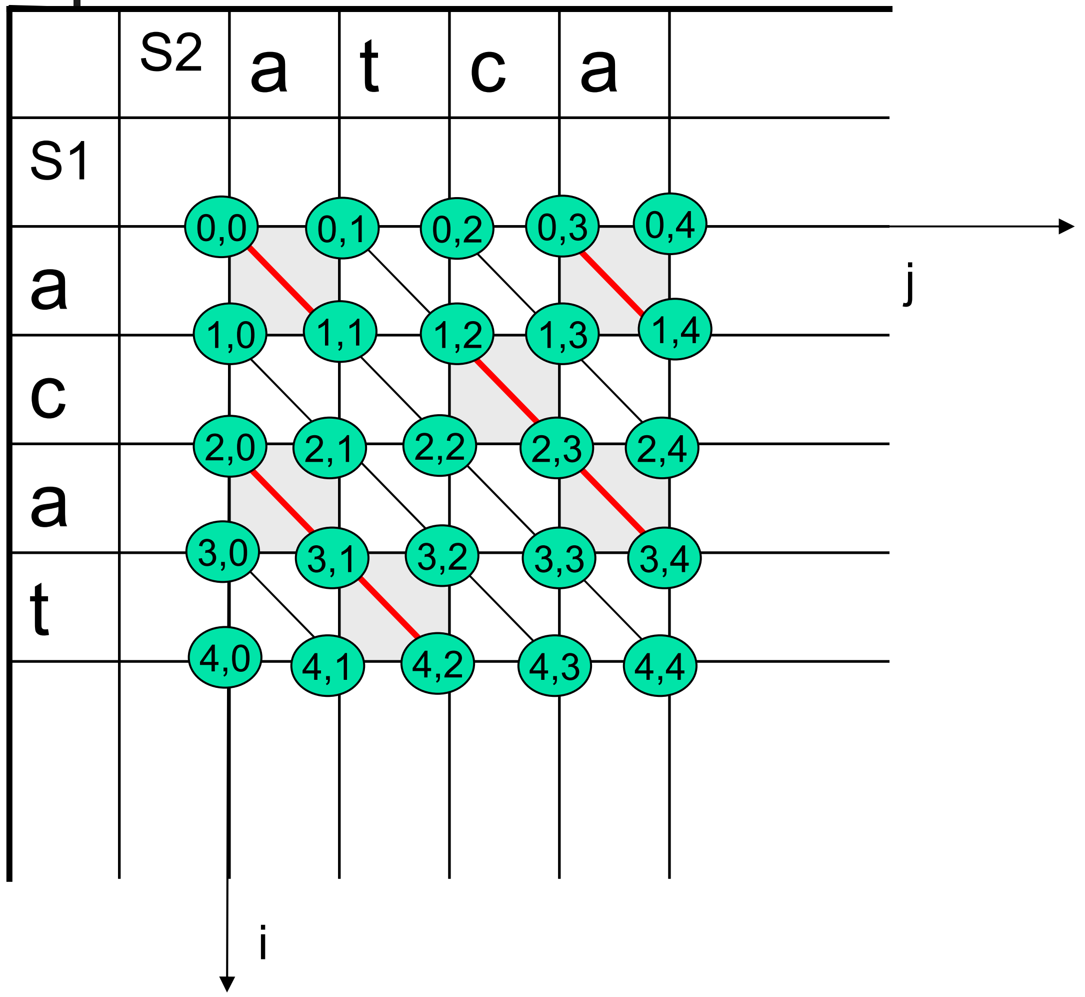
An edit graph



An edit graph for a pair of strings S_1 and S_2 has $(N+1)*(M+1)$ vertices, each labeled with a corresponding pair (i,j), $0 \le i \le N$, $0 \le j \le M$

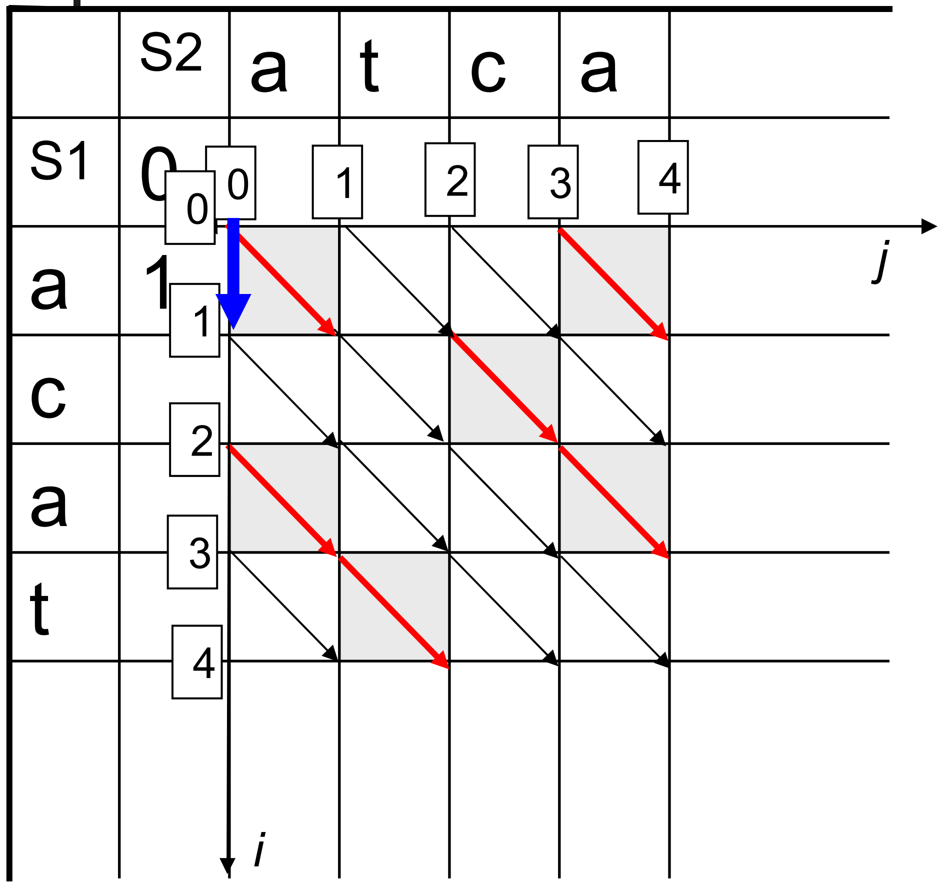
The edges are directed and their weight depends on the specific string problem: for the edit distance problem – red edges have cost 0, black edges have cost 1

The cheapest path in the edit graph



The cost of a cheapest path from a vertex (0,0) to vertex (*N*,*M*) in this edit graph corresponds to the edit distance between S1 and S2, and the path itself represents a series of edit operations and an optimal alignment of S1 with S2

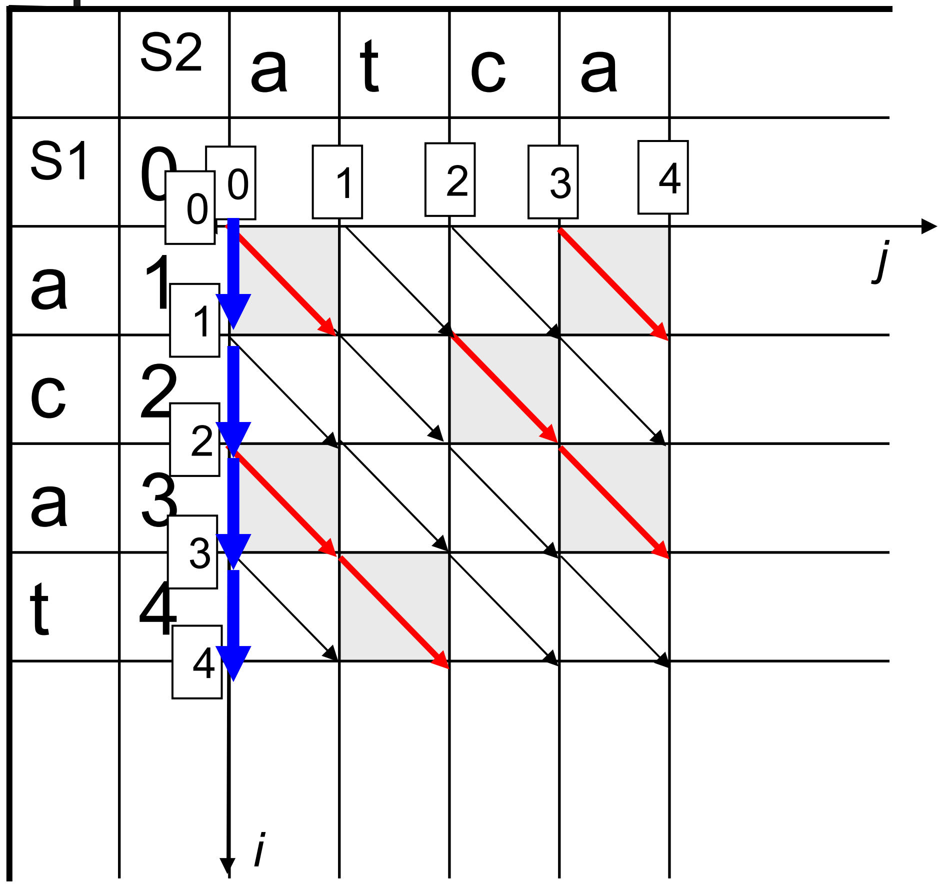
Calculating the edit distance. Base condition



The minimum number of edit operations we need in order to transform an empty string (of length 0) into string *a* is 1 (insertion)

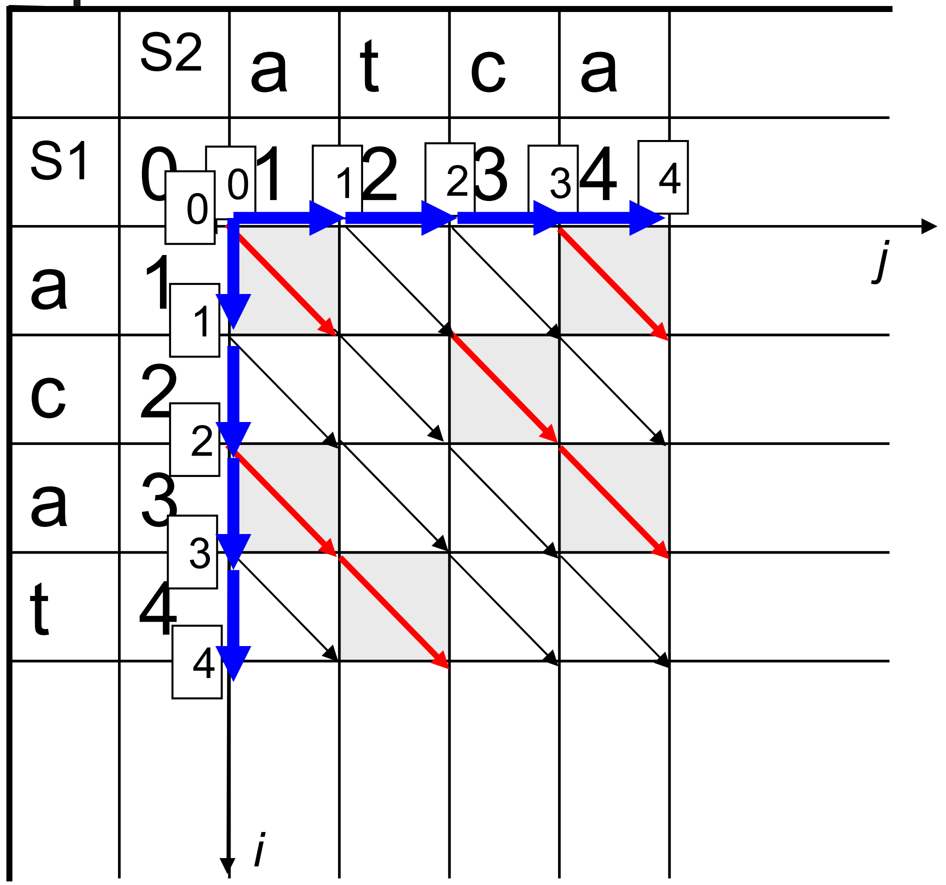
Therefore the minimum edit distance between ε and *a* is 1

Calculating the edit distance. Base condition



The same is true for ε and *ac, aca, acat*

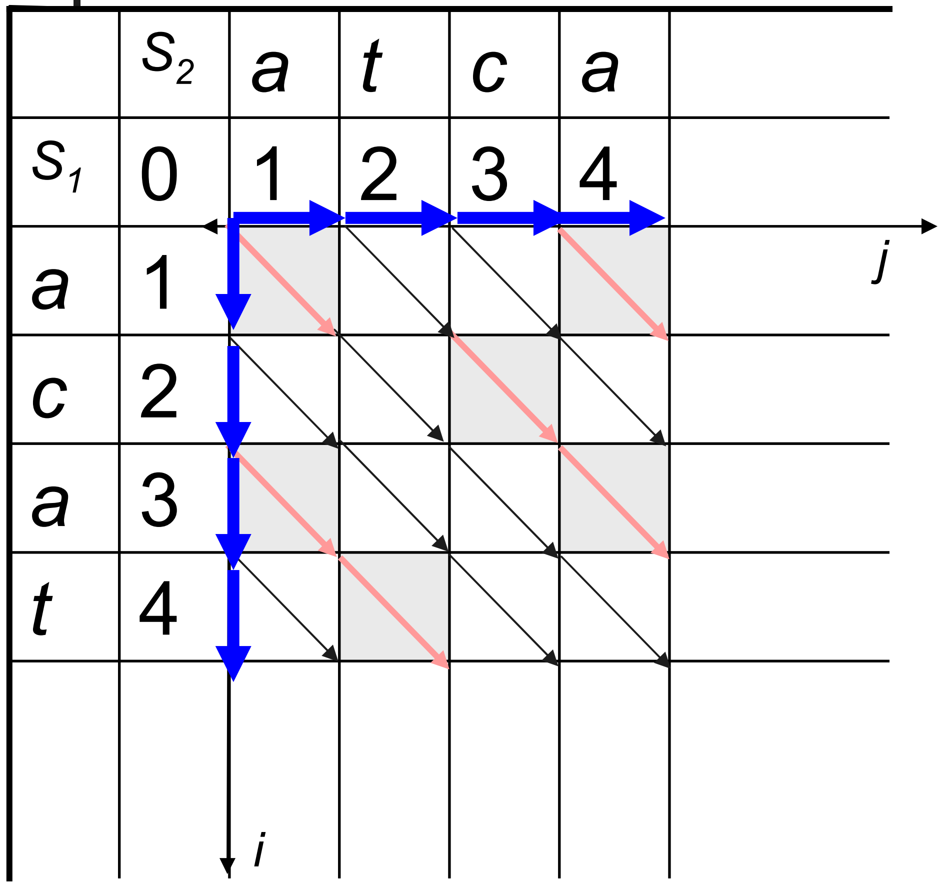
Calculating the edit distance. Base condition



In order to transform *a* into ε, we need to delete 1 character. This is the best way to do it, there is no other way.

The same for transforming *at, atc, atca* into ε with 2, 3, 4 deletions respectively

Calculating the edit distance. Recursion for i>0 and j>0



There are only 3 different ways to move trough the next cell in the grid, namely:

- Increase both *i* and *j* (diagonal)

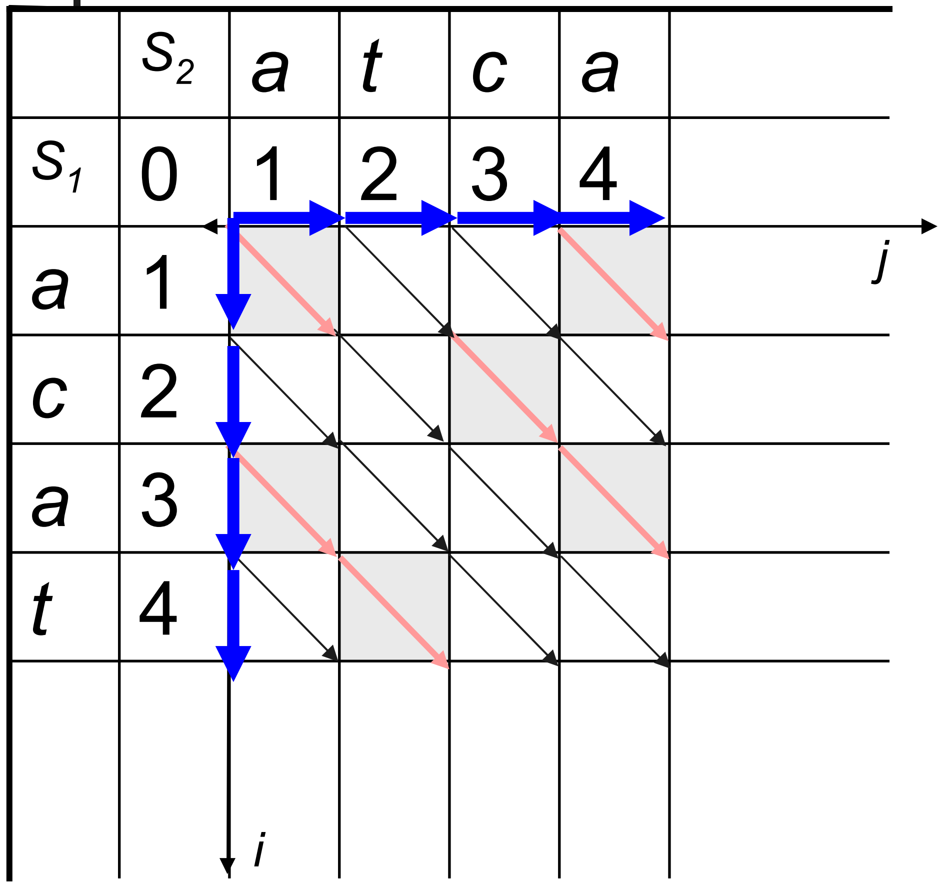
  with 1 edit operation if $S_1[i] \neq S_2[j]$

  with 0 cost if $S_1[i] = S_2[j]$
- Increase only *i* (insertion of $S_1[i]$ into S_2)

  with the cost 1
- Increase only *j* (deletion of $S_2[j]$ from S_2)

  with the cost 1

Calculating the edit distance. Recursion for i>0 and j>0

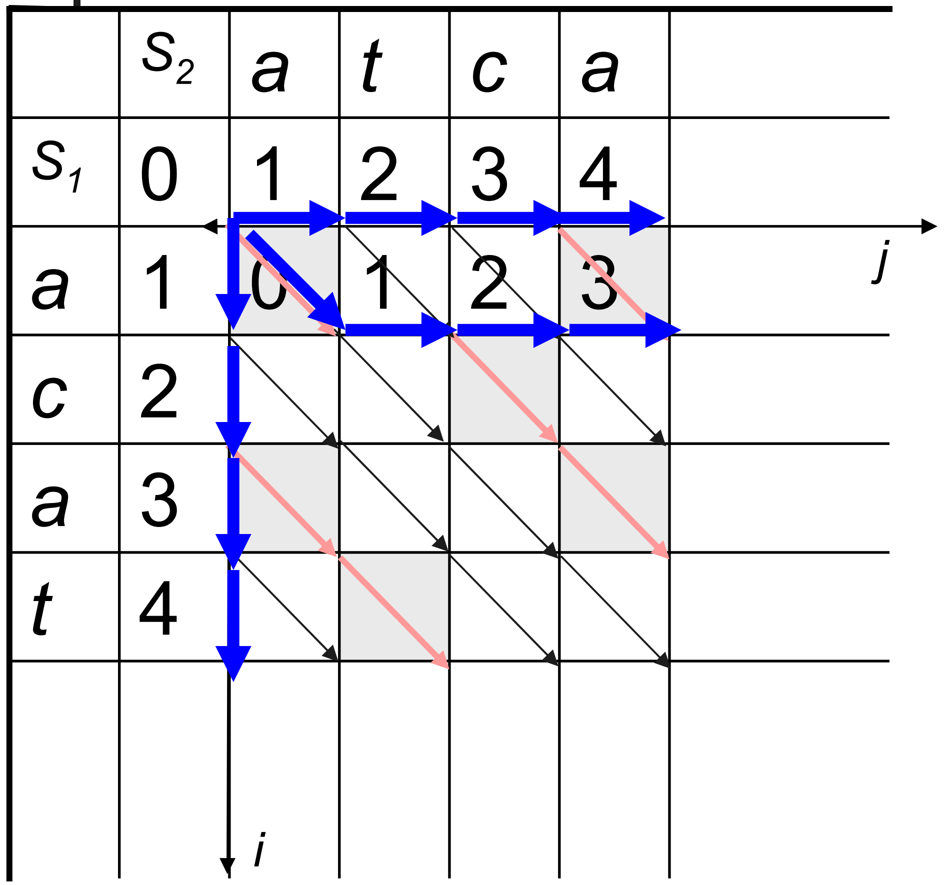


Thus, if we know the edit distance

D[*i*-1,*j*-1], *D*[*i*-1,*j*] and *D*[*i*,*j*-1], we can correctly calculate *D*[*i*,*j*]

This is true since there are no other ways of moving through cell [*i*][*j*], and reaching the top, left and top-left corners by different paths cannot produce a better value than is already in these 3 cells, since they contain the minimum cost by definition

Calculating the edit distance. Recursion for i>0 and j>0

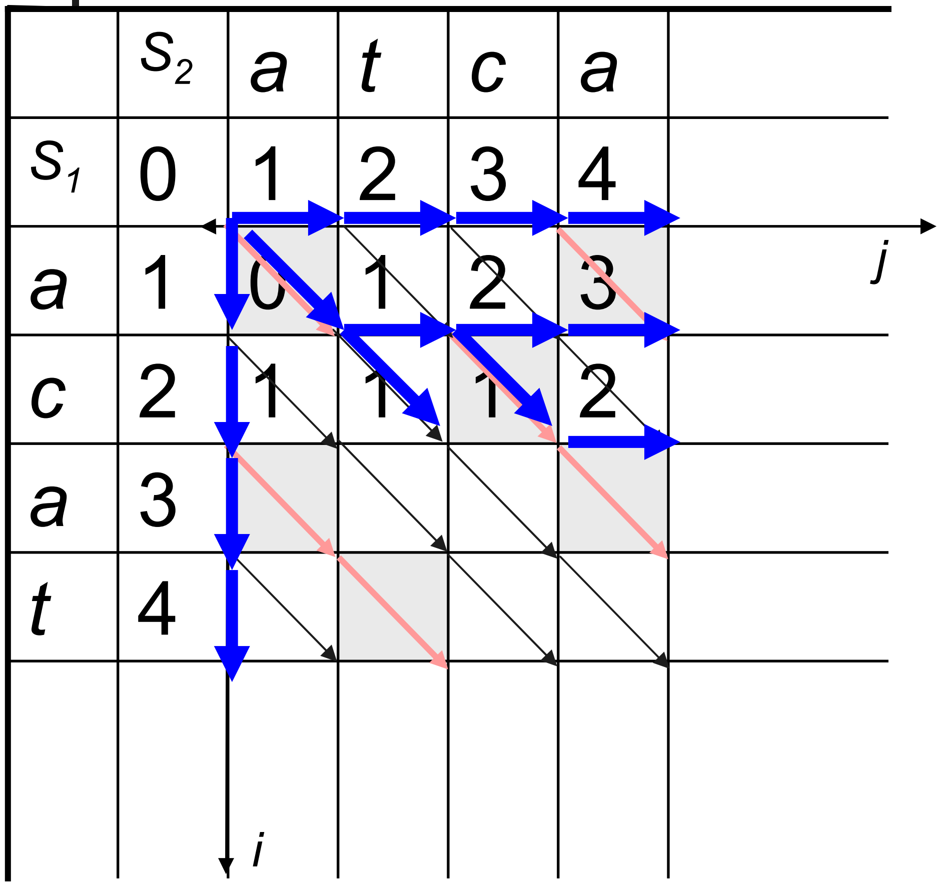


	S_2	*a*	*t*	*c*	*a*
S_1	0	1	2	3	4
a	1	0	1	2	3
c	2				
a	3				
t	4				

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) = \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating the edit distance. Recursion for i>0 and j>0

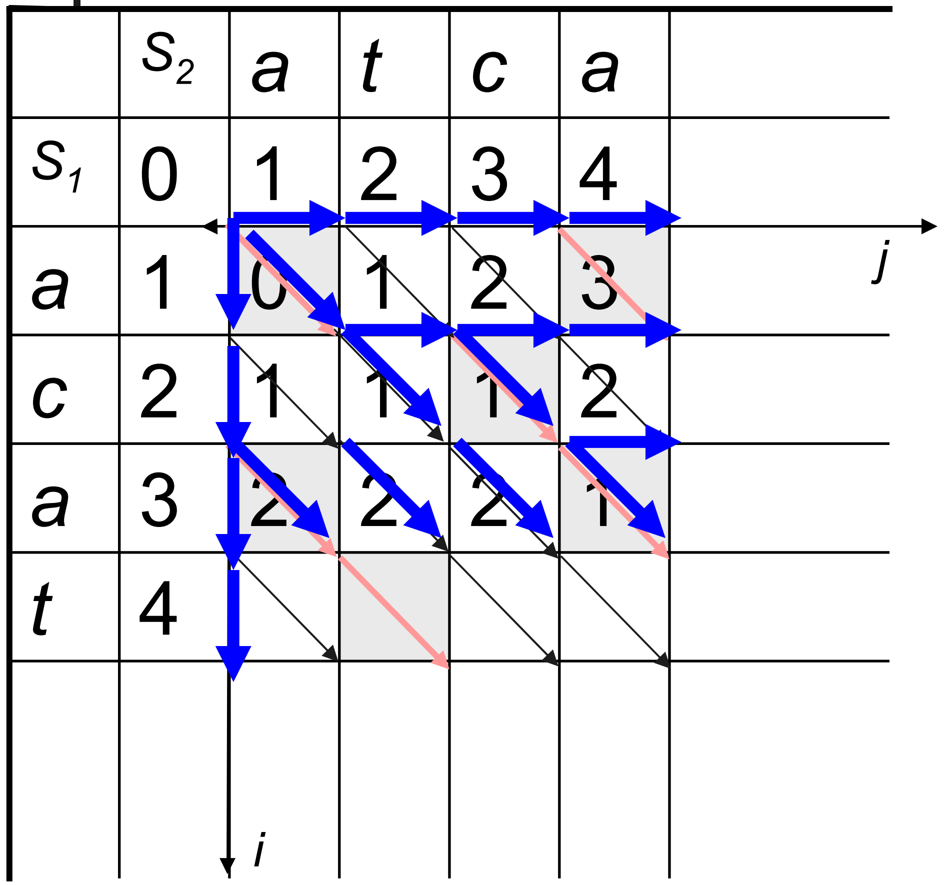


	S_2	*a*	*t*	*c*	*a*
S_1	0	1	2	3	4
a	1	0	1	2	3
c	2	1	1	1	2
a	3				
t	4				

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) = \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating the edit distance. Recursion for i>0 and j>0

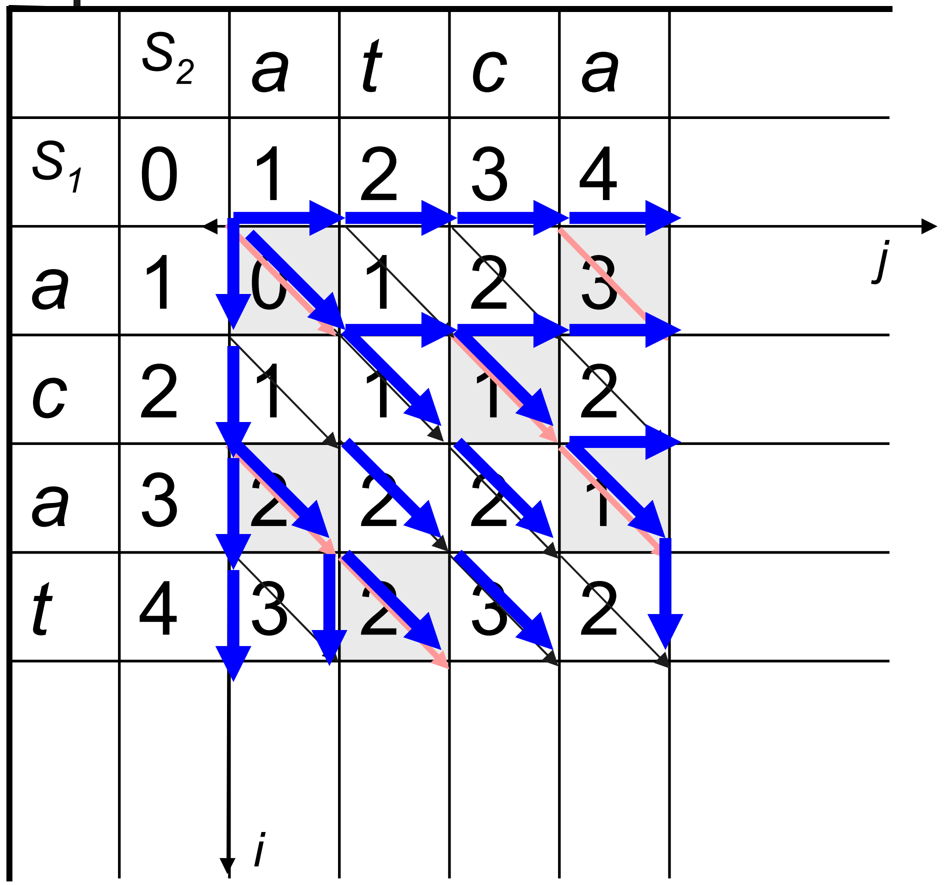


	S_2	*a*	*t*	*c*	*a*	
S_1	0	1	2	3	4	
a	1	0	1	2	3	
c	2	1	1	1	2	
a	3	2	2	2	1	
t	4					

$$D(i,j) \quad = \quad \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) \quad = \quad \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

Calculating the edit distance. Recursion for i>0 and j>0

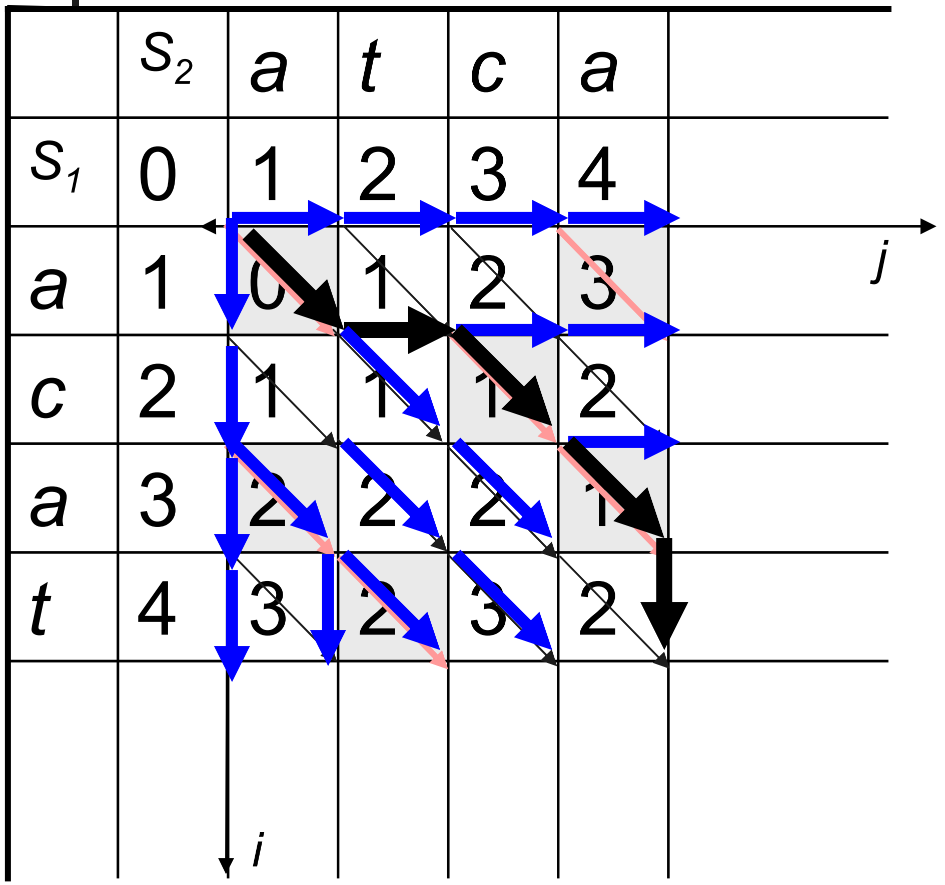


	S_2	*a*	*t*	*c*	*a*
S_1	0	1	2	3	4
a	1	0	1	2	3
c	2	1	1	1	2
a	3	2	2	2	1
t	4	3	2	3	2

$$D(i,j) \quad = \quad \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+c(i,j) \end{cases}$$

$$\text{where } c(i,j) \quad = \quad \begin{cases} 0 \text{ if } S1[i]=S2[j] \\ 1 \text{ if } S1[i] \neq S2[j] \end{cases}$$

The sequence of edit operations



	S_2	*a*	*t*	*c*	*a*
S_1	0	1	2	3	4
a	1	0	1	2	3
c	2	1	1	1	2
a	3	2	2	2	1
t	4	3	2	3	2

↘	Place a character in S1 opposite to a character in S2
↓	Place a character in S1 opposite to a gap in S2
→	Place a character in S2 opposite to a gap in S1

S1	*a*	-	*c*	*a*	*t*
S2	*a*	*t*	*c*	*a*	-

Optimal alignment for two strings

S1	*a*	*-*	*c*	*a*	*t*
S2	*a*	*t*	*c*	*a*	*-*

Evolutionary explanation:

S2 evolved from S1 by a series of the following mutations:

Insertion of nucleotide T at position 2

Deletion of nucleotide T at position 5

An optimal alignment is not unique

S1	-	*a*	*t*	*t*	*a*	*a*	*g*
S2	*t*	*a*	-	*t*	*c*	*a*	*g*

S1	-	*a*	*t*	*t*	*a*	*a*	*g*
S2	*t*	*a*	*t*	*c*	*a*	-	*g*

2 different alignments with the optimal minimal cost 3

The exact sequence of mutations cannot be determined

Edit distance as a measure of a similarity

S1	*a*	*-*	*c*	*a*	*t*
S2	*a*	*t*	*c*	*a*	*-*

If the number of basic evolutionary events is small, we infer that the divergence between S1 and S2 happened not so long time ago, and that the two strings are still *similar*

The smaller is the edit distance between 2 strings, the more similar they are

Complexity of the edit distance computation

- Quadratic - O(*NM*),

Where *N* is the length of S_1, *M* is the length of S_2

What if *N* and *M* are very large?

2 main problems:

Quadratic running time

Quadratic space

ED computation in a linear space. The algorithm by Hirschberg

- The time complexity is proportional to the number of edges in the edit graph: O(*NM*)
- The space complexity is proportional to the number of vertices in the edit graph, since for each vertex we need to store the best incoming edge: O(*NM*)

ED computation. Space

- For very long strings, the quadratic computation time is not as bad as the quadratic space required in order to store all the traceback pointers
- The quadratic space is a bottleneck of these algorithms

If we only want the value D[N][M]

		a	t	c	a	t	g
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
c							
a							
t							
a							
g							

If we only want the value D[N][M]

		a	t	c	a	t	g
a	1	0	1	2	3	4	5
c	2	1	1	1	2	3	4
a							
t							
a							
g							

We don’t need row 0 for computing values in row 3

If we only want the value D[N][M]

		a	t	c	a	t	g
a			╲	╲		╲	╲
c	2	╲ 1	╲ 1	1	╲ 2	╲ 3	╲ 4
a	3	2	╲ 2	╲ 2	1	╲ 2	╲ 3
t		╲		╲	╲		╲
a			╲	╲		╲	╲
g		╲	╲	╲	╲	╲	

If we only want the value D[N][M]

		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a	3	2	╲ 2	╲ 2	1	╲ 2	╲ 3
t	4	╲ 3	2	╲ 3	╲ 2	1	╲ 2
a			╲	╲		╲	╲
g		╲	╲	╲	╲	╲	

If we only want the value D[N][M]

		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a			╲	╲		╲	╲
t	4	╲ 3	2	╲ 3	╲ 2	1	╲ 2
a	5	4	╲ 3	╲ 3	3	╲ 2	╲ 2
g		╲	╲	╲	╲	╲	

If we only want the value D[N][M]

		a	t	c	a	t	g
a							
c							
a							
t							
a	5	4	3	3	3	2	2
g	6	5	4	4	4	3	2

Then we don't need more space than to store 2 rows of a table

Since for computing row *i* we only need to know values in the row *i*-1, so the values in rows before *i*-1 can be discarded

This computation can be performed in linear space O(*N*)

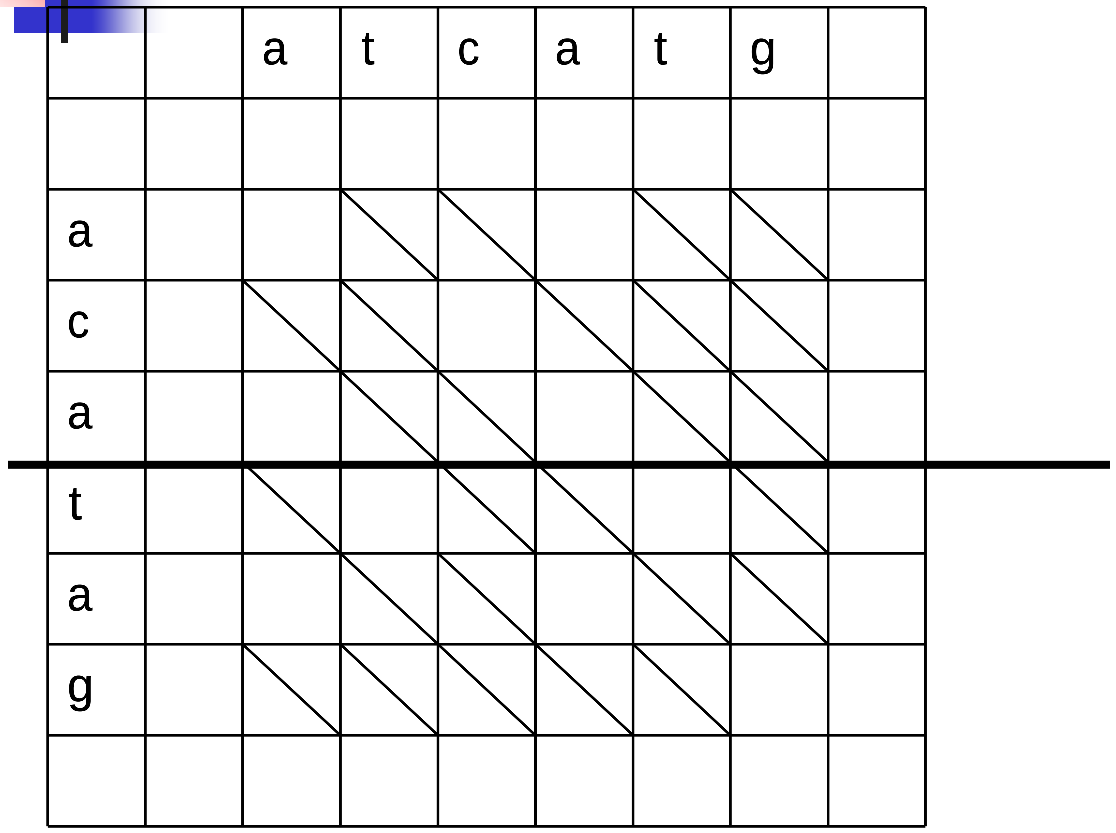
But in order to actually find a series of edit operations

		a	t	c	a	t	g
a			╲	╲		╲	╲
c		╲	╲		╲	╲	╲
a			╲	╲		╲	╲
t		╲		╲	╲		╲
a	5	4	╲ 3	╲ 3	3	╲ 2	╲ 2
g	6	╲ 5	╲ 4	╲ 4	╲ 4	╲ 3	🟥 2

We need to store the pointers for each vertex in the entire graph in order to be able to trace the path back

How did we get there with D=2?

The median border in the graph

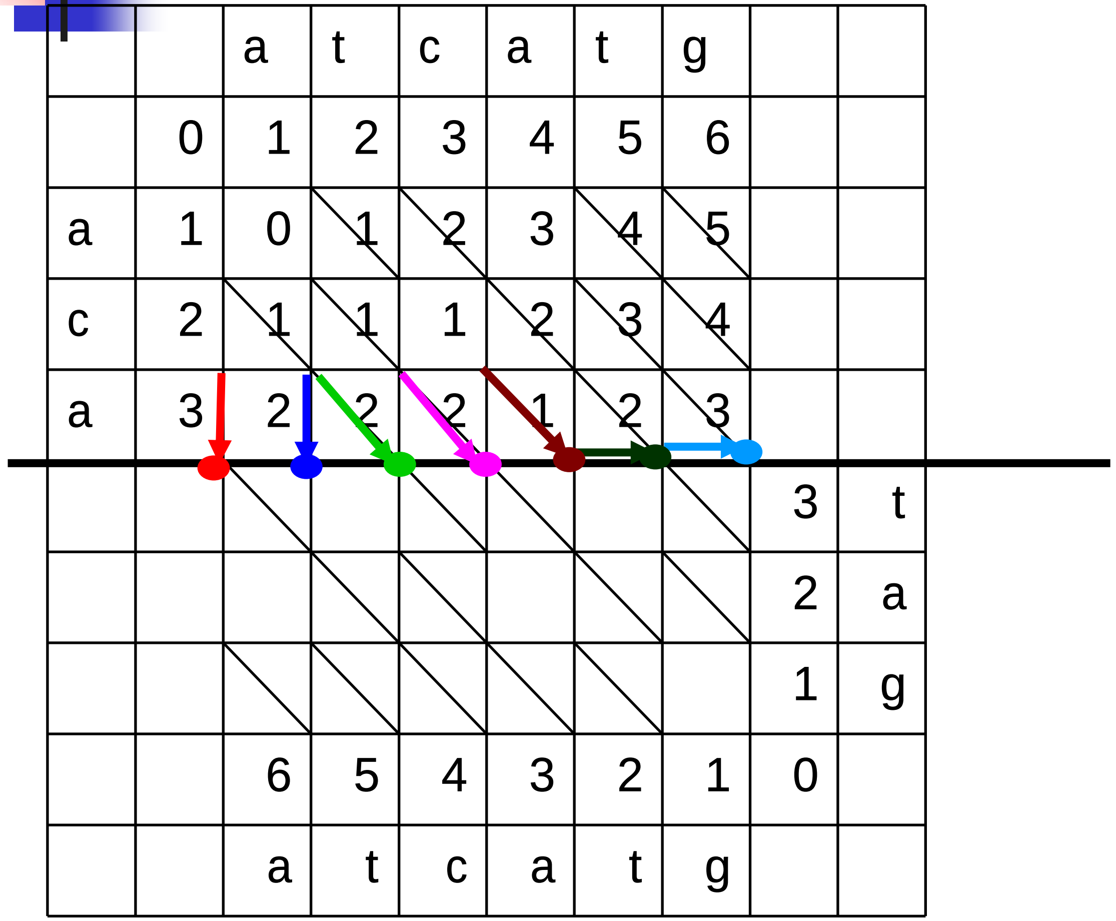


Let us set the median line after the row N/2

Each path, including the optimal path we are looking for, crosses the median line

The goal – to find the point in the median line, where an optimal path crosses it

All the paths running from the source …

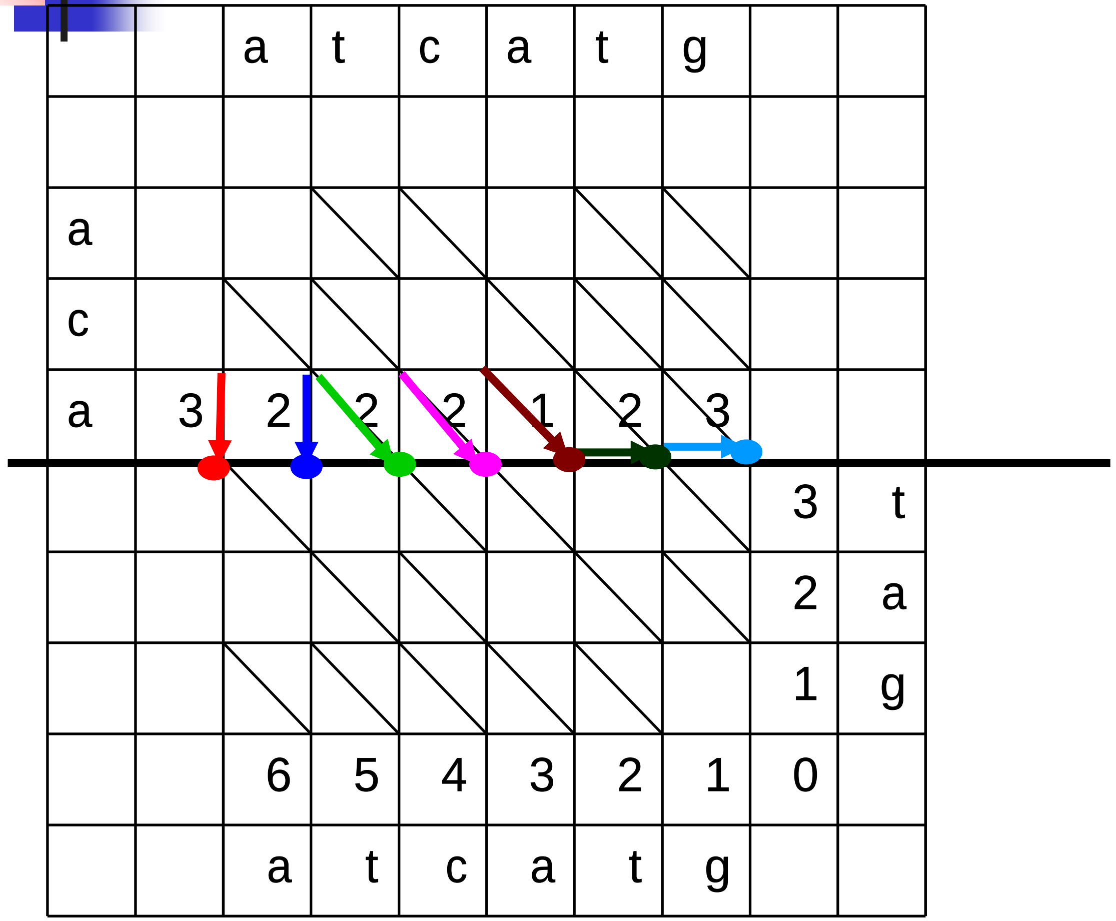


Compute the values of D for the row above the median line

Do not store all the pointers, use only space for 2 rows

Mark the last row with the traceback pointers to the previous row

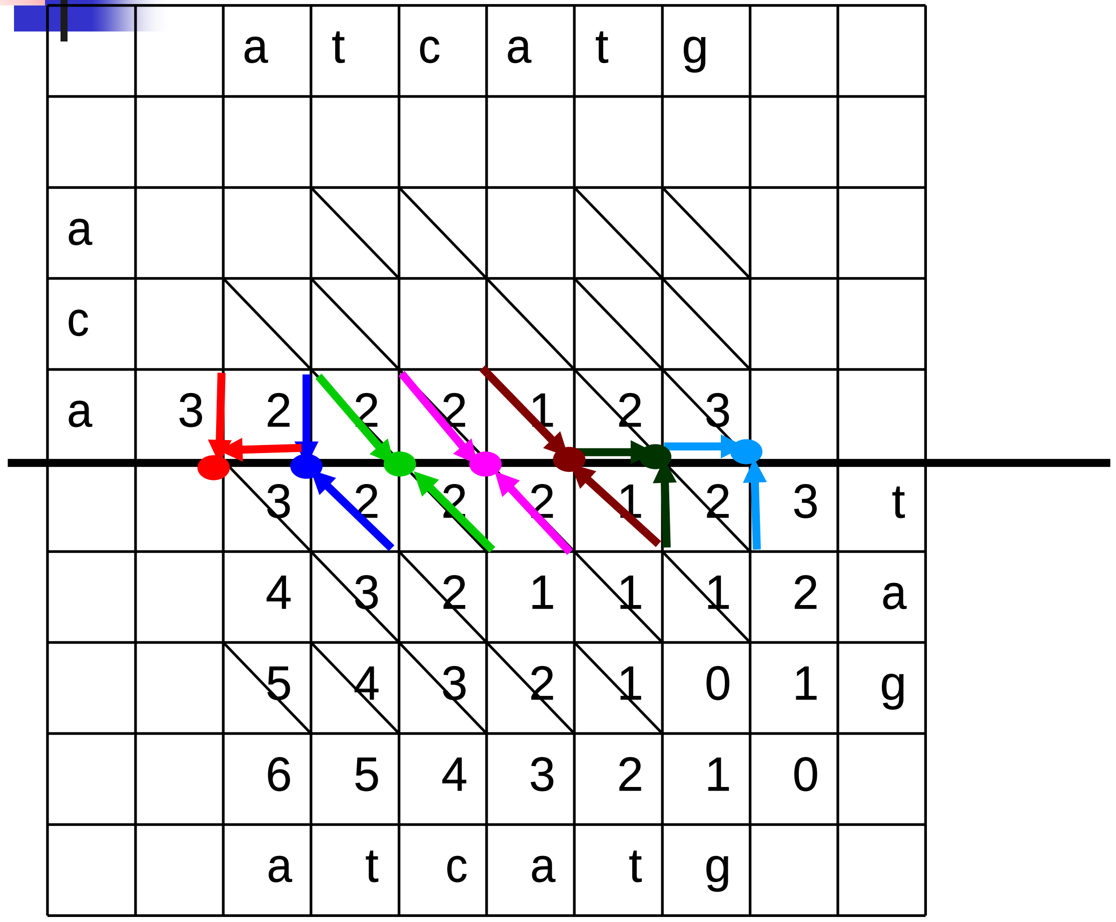
All the paths running from the source ...



We have obtained the set of values of the best paths which run from the source till each point in the median line

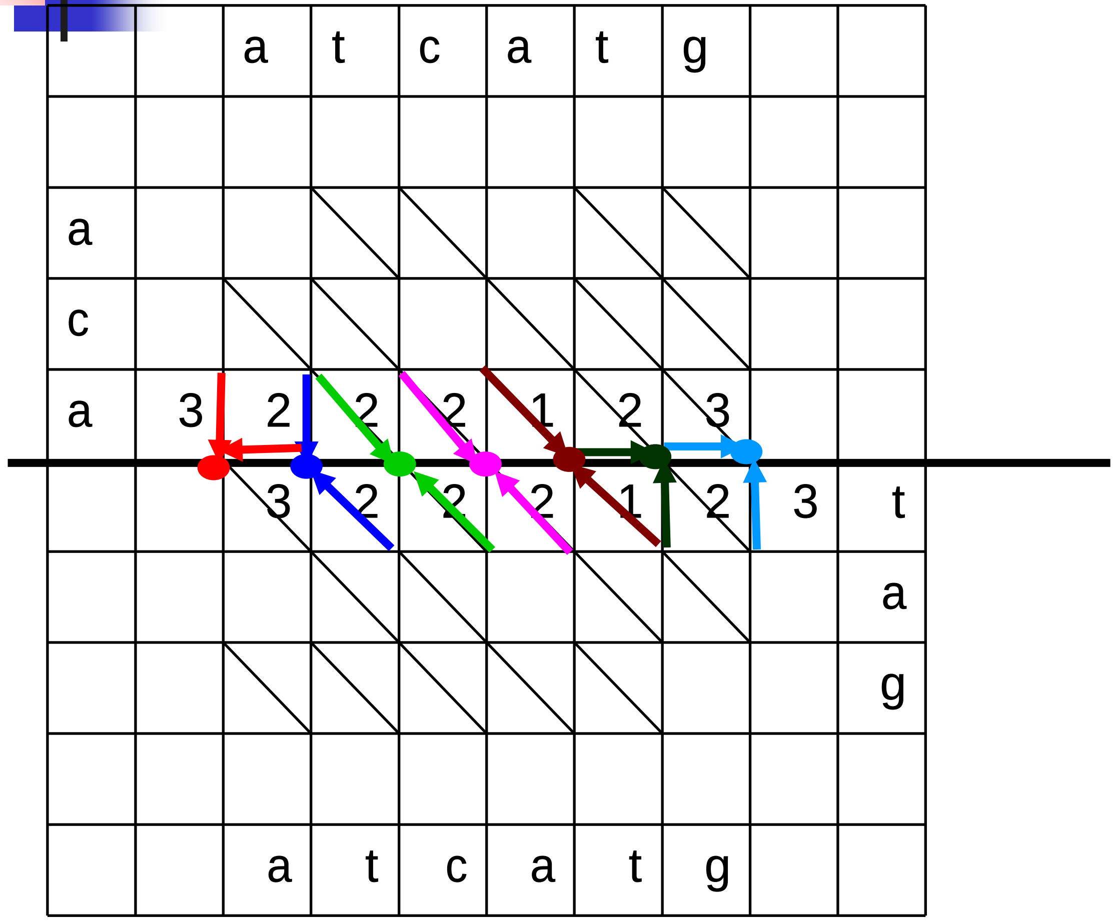
But we cannot choose yet which of these points belong to an overall cheapest path

All the paths running from the source and from the destination



Next, we compute all the best paths running from the destination point (6,6) till each point on the median line, considering the same strings in the opposite direction

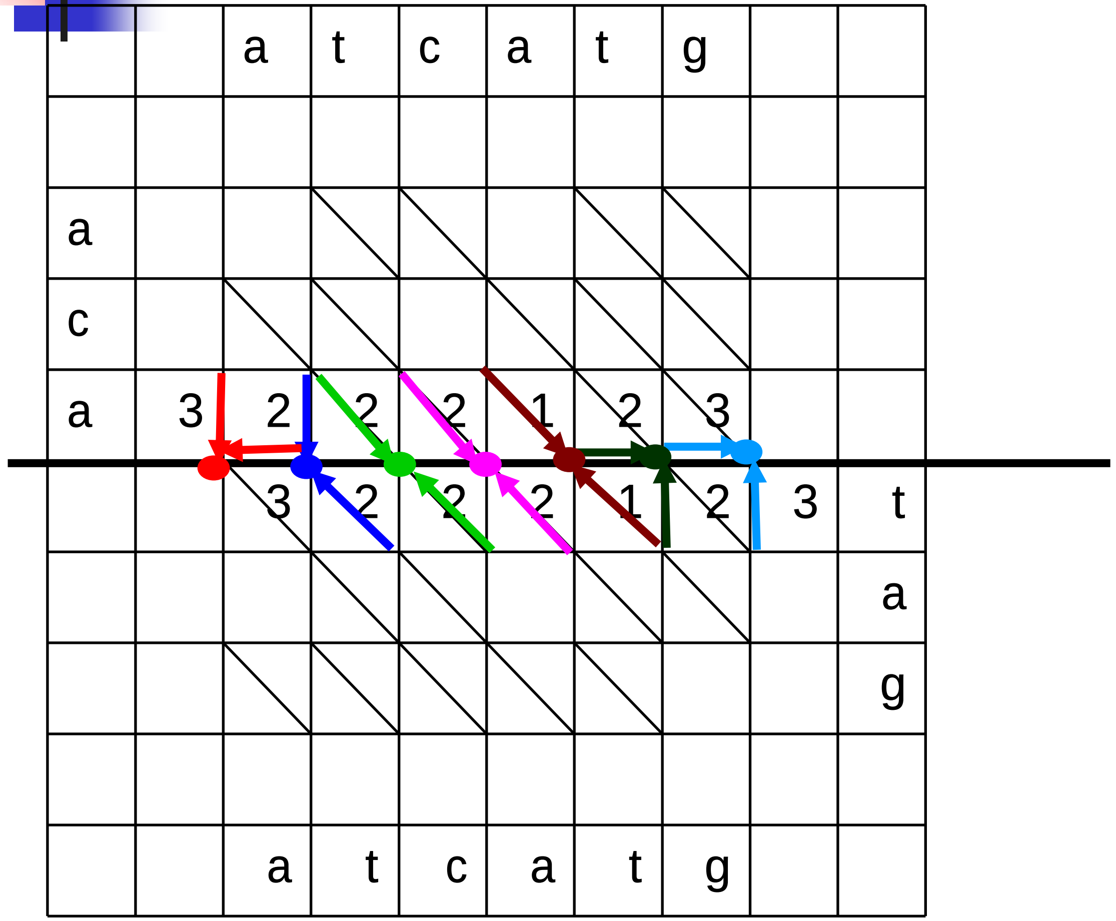
All the paths running from the source and from the destination



This is enough information to compute the total cost of the cheapest path passing though each point on the median line:

●	3+3=6
●	2+2=4
●	2+2=4
●	2+2=4
●	1+1=2
●	2+2=4
●	3+3=6

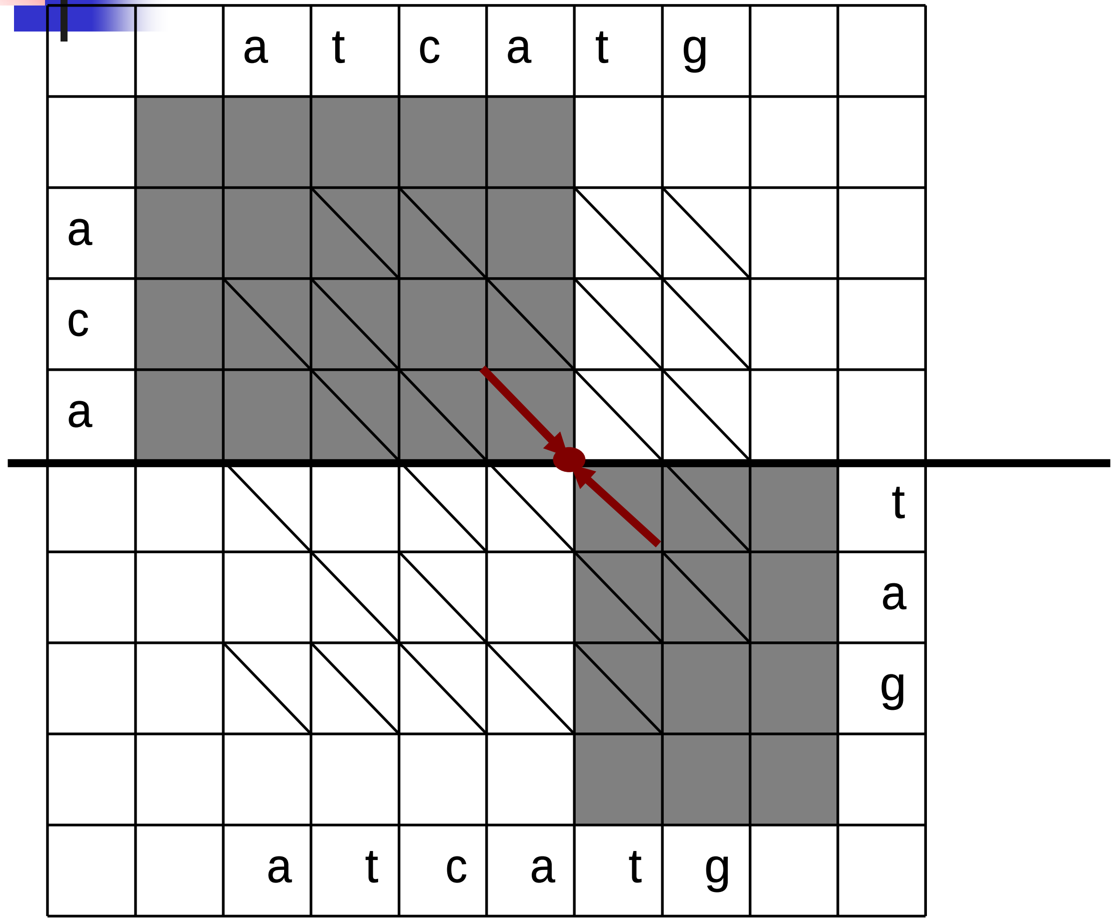
The median line hit point of the best path



We infer that the overall cheapest path of cost 2 hits the median line in point (3,4):

●	3+3=6
●	2+2=4
●	2+2=4
●	2+2=4
●	1+1=2
●	2+2=4
●	3+3=6

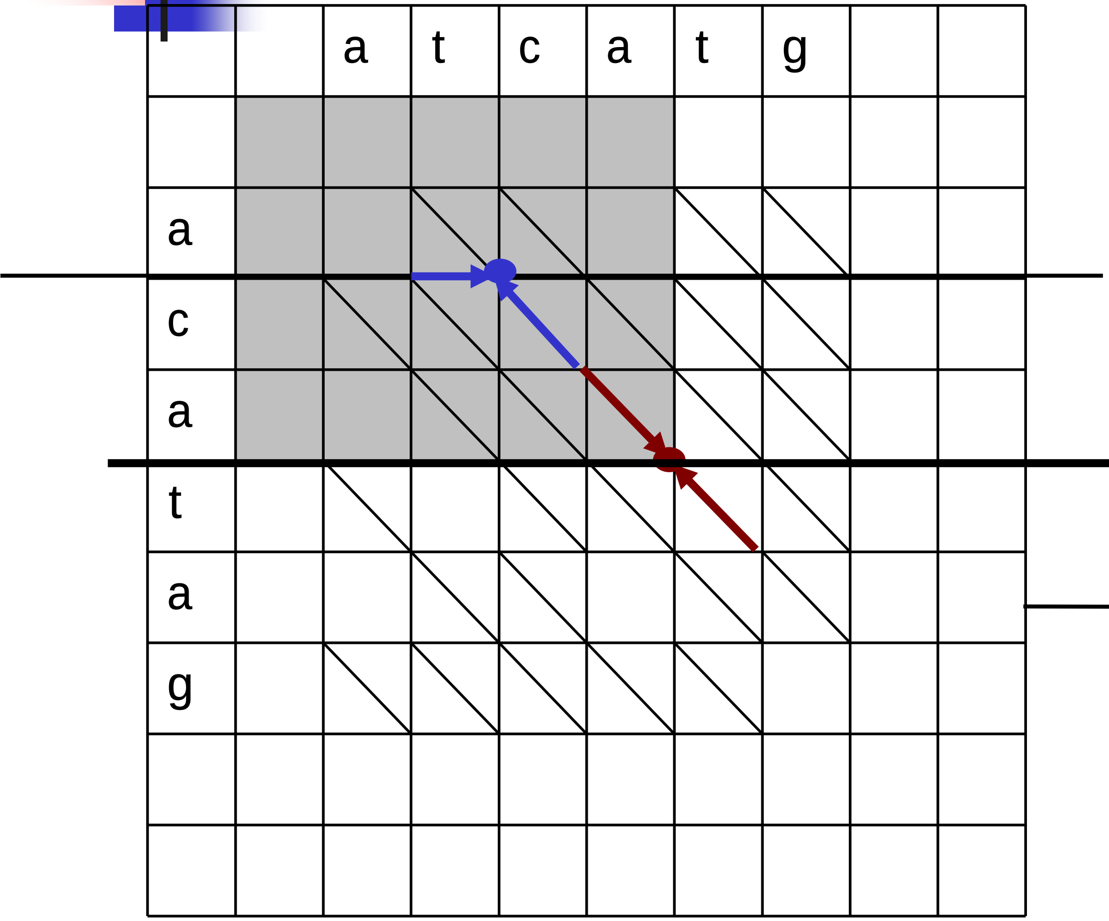
The remaining parts of the best path



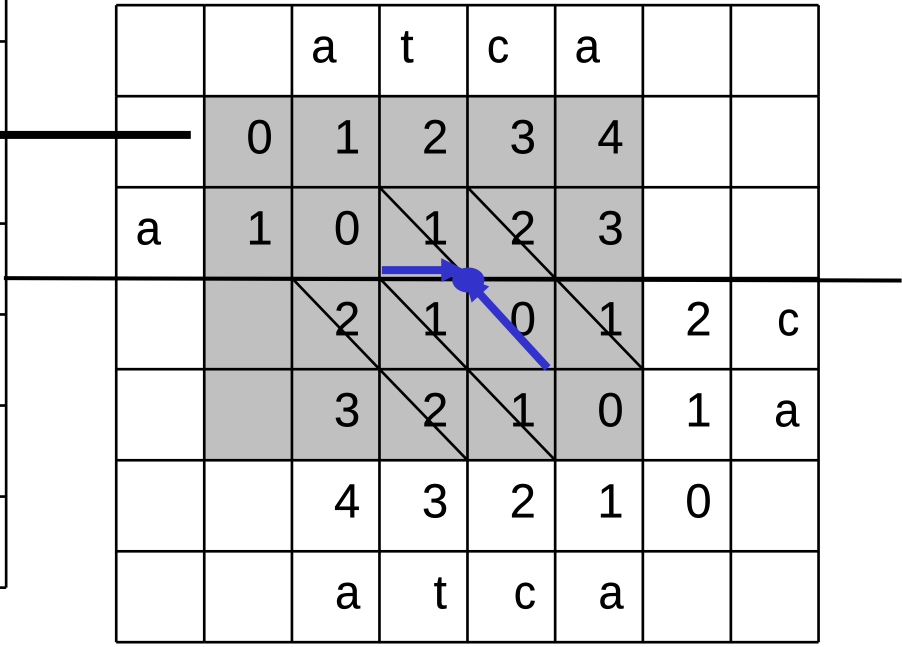
Thus, we have found a piece of the best path

Next, we need only to find the remaining parts of the path which can pass only inside grey areas of the grid

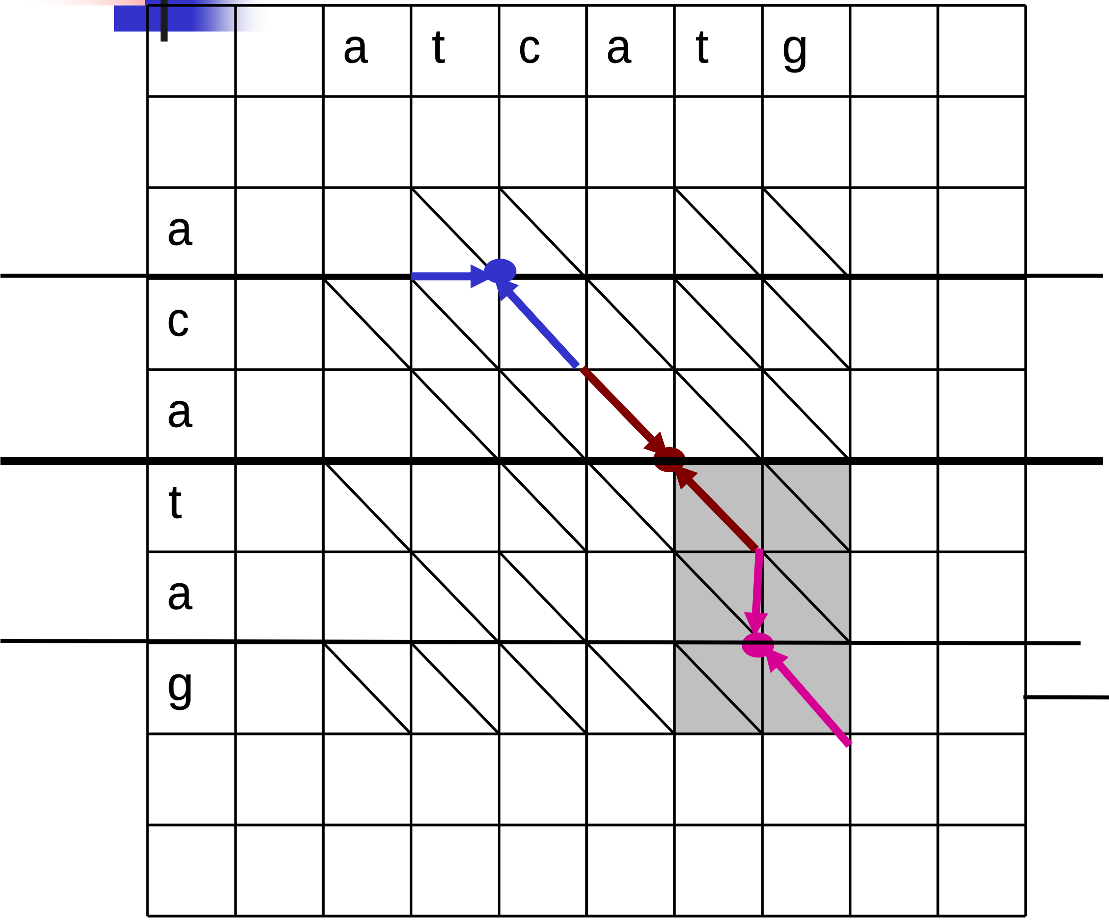
Recursive computation for NM/2 cells



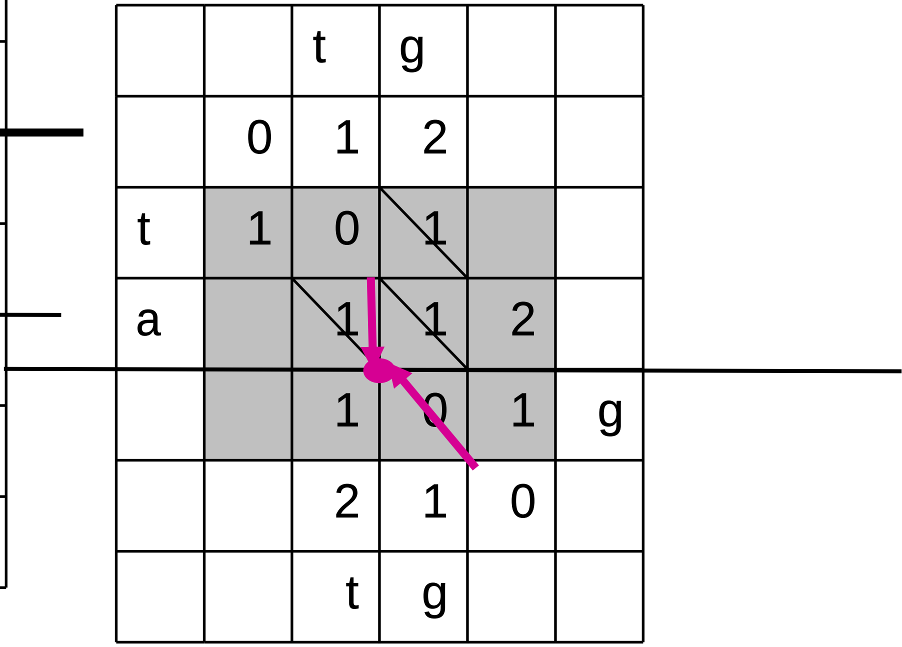
By the same bi-directional algorithm we compute the piece of the path which hits the median line of the upper-left and …



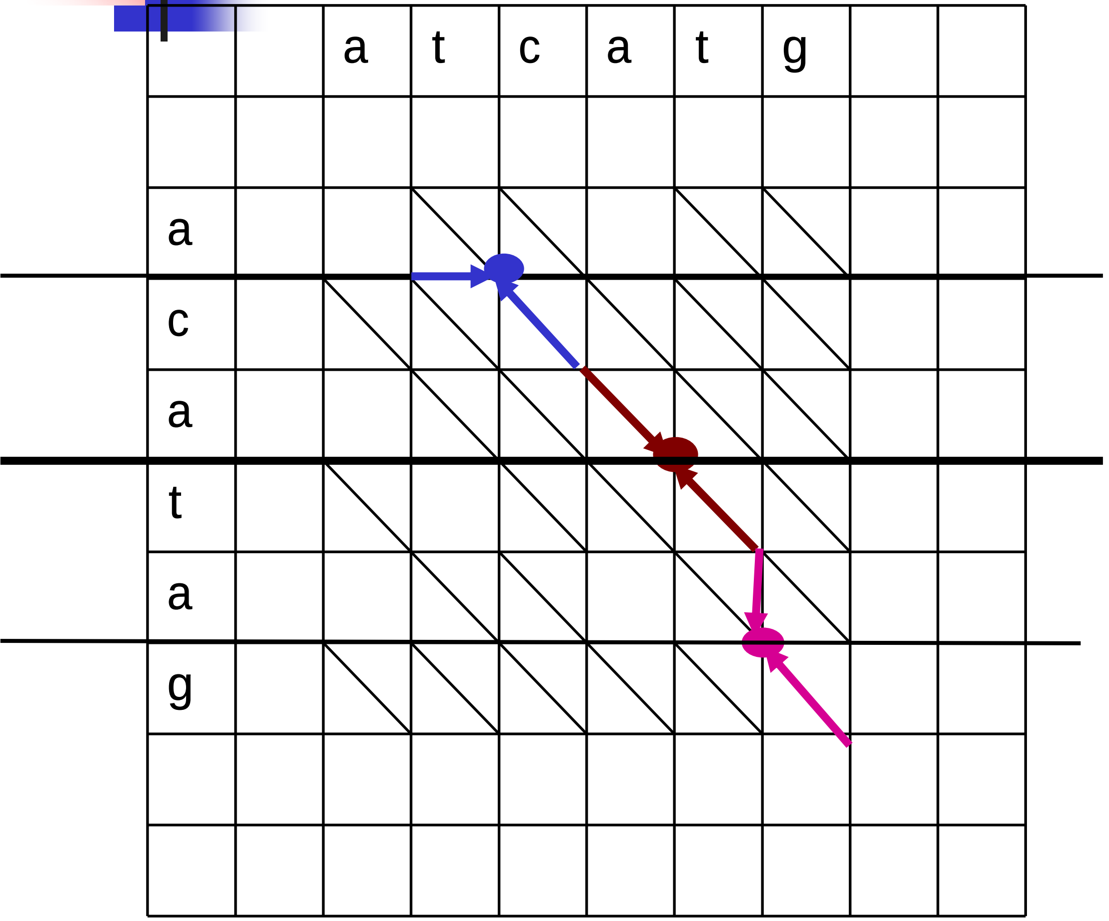
Recursive computation for NM/2 cells



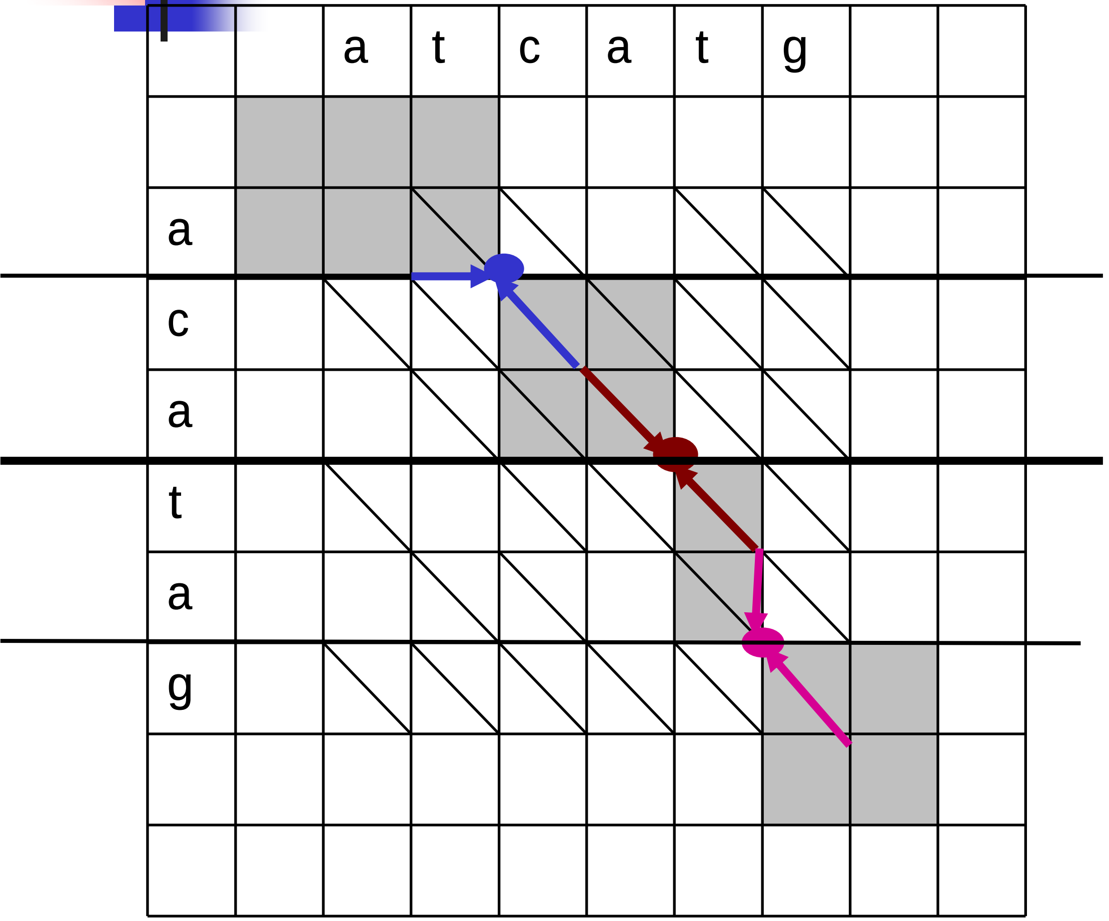
By the same bi-directional algorithm we compute the piece of the path which hits the median line of the upper-left and … the bottom-right squares



We know 1+2 pieces of the best path

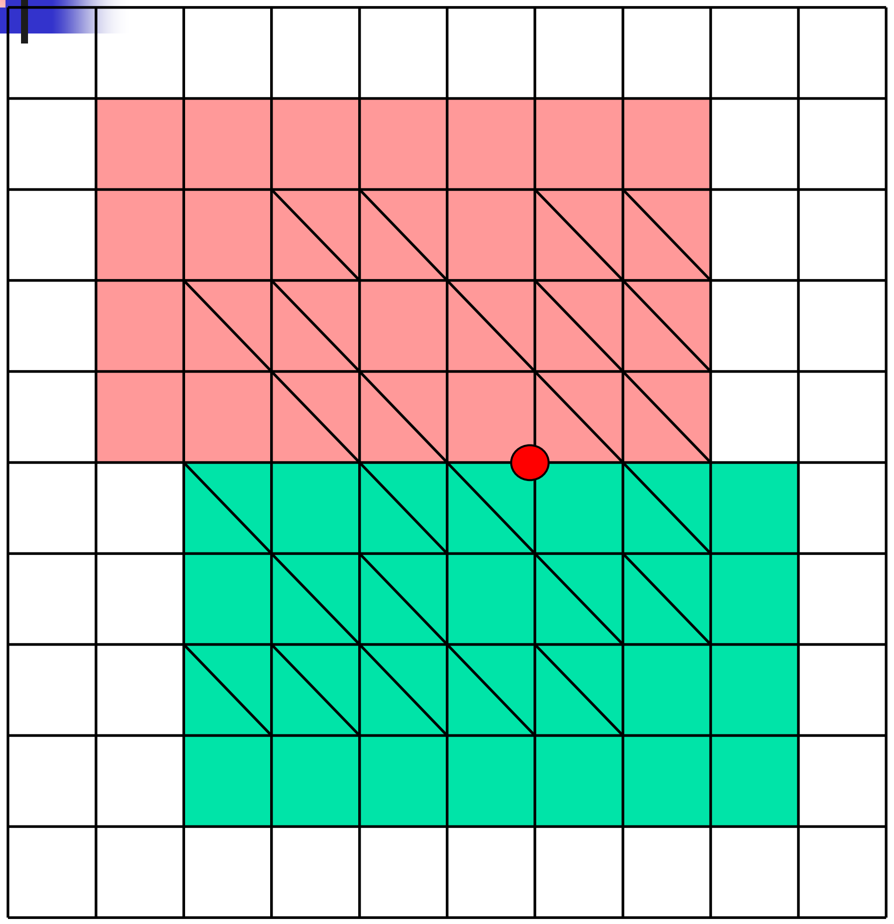


Recursive computation for NM/4



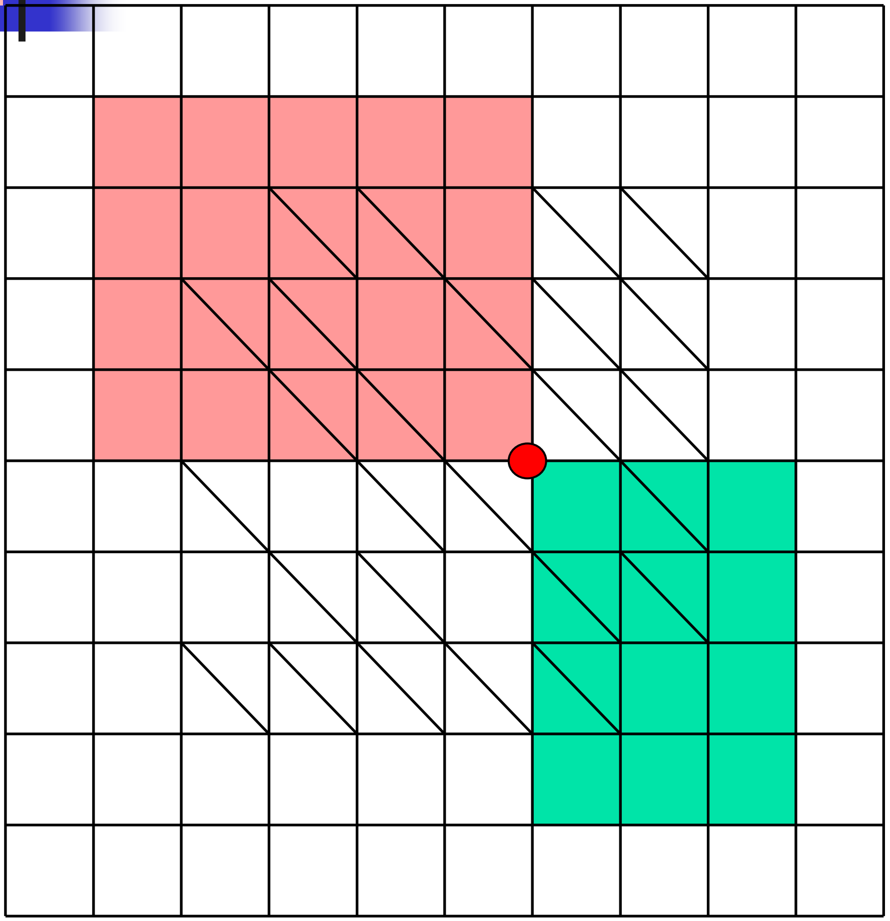
The areas remaining for the computation are in grey

The Hirschberg's algorithm. Step 1. Computed NM cells



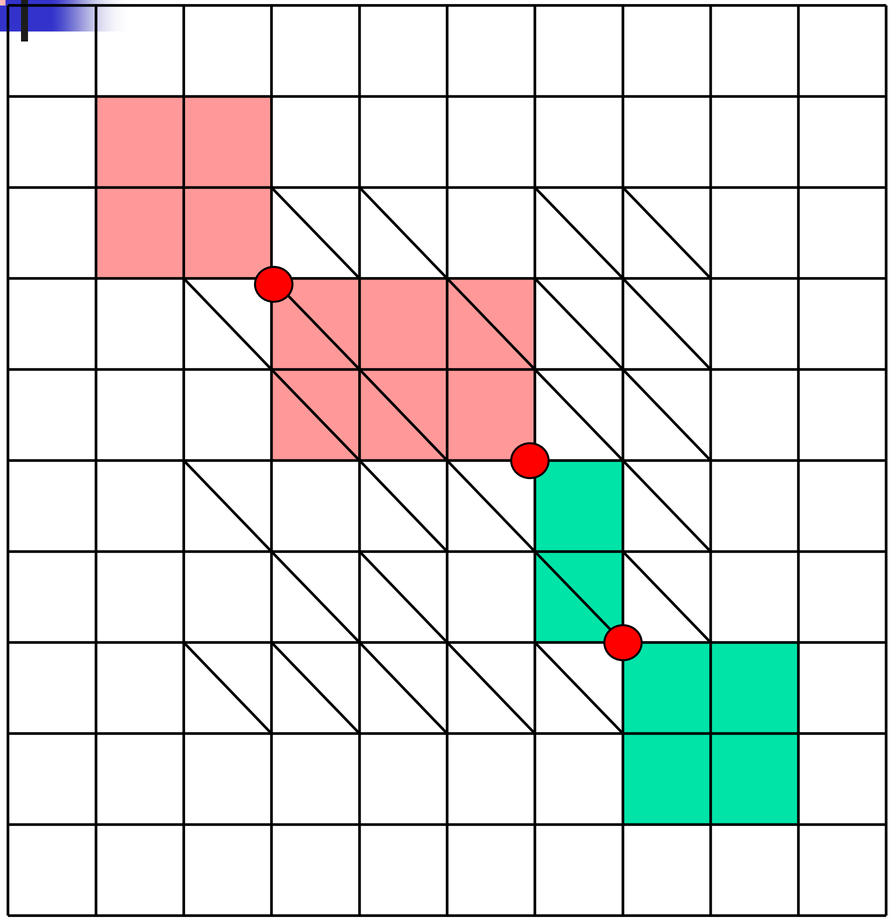
Each time we compute two tables, whose total size is 2 times smaller than in the previous step. In each computation we find an additional point belonging to the cheapest path, and recording it

The Hirschberg’s algorithm. Step 2. Computed NM/2 cells



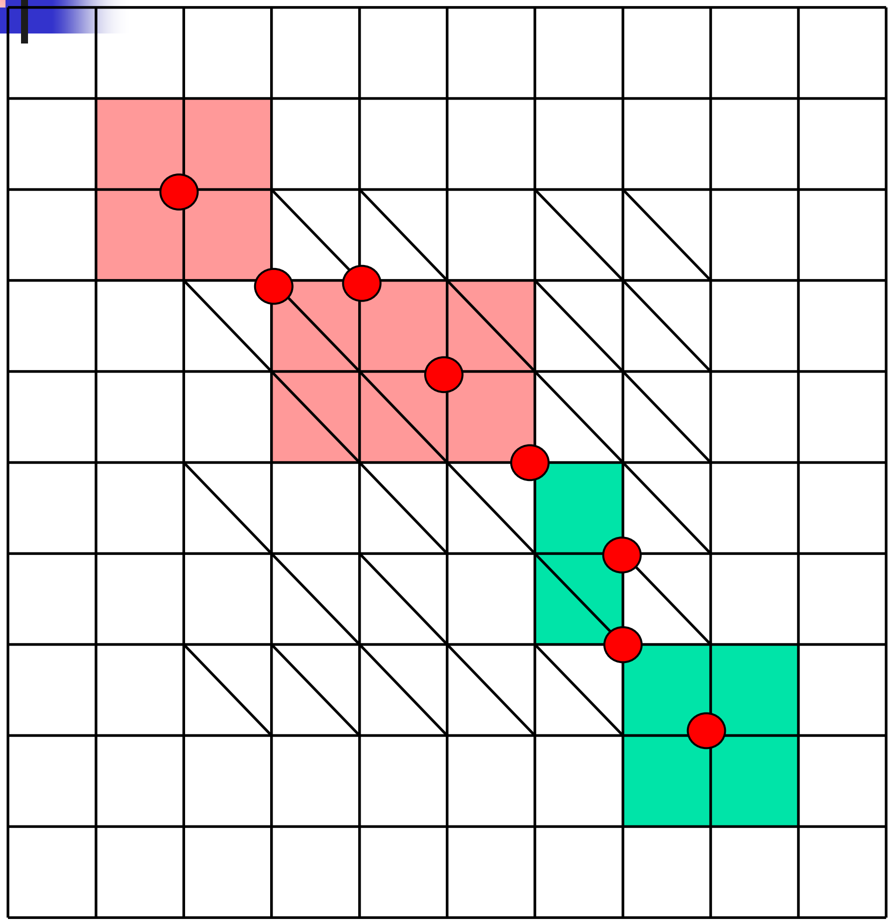
Each time we compute two tables, whose total size is 2 times smaller than in the previous step. In each computation we find an additional point belonging to the cheapest path, and recording it

The Hirschberg's algorithm. Step 3. Computed NM/4 cells



Each time we compute two tables, whose total size is 2 times smaller than in the previous step. In each computation we find an additional point belonging to the cheapest path, and recording it

The Hirschberg’s algorithm. Termination



When only 2 rows left to be computed, we can find the best path using only 2 rows of each table

The total path is complete

The Hirschberg's algorithm. Time complexity

- The algorithm computes values of $NM + NM/2 + NM/4 + \ldots . NM/(NM/2) =$
  $= NM(1+1/2+1/4+\ldots) = 2NM$ cells

The time complexity is still $O(NM)$

The Hirschberg’s algorithm. Space complexity

- The algorithm never uses the space more than for 2 rows of the table

The space complexity is O(N)

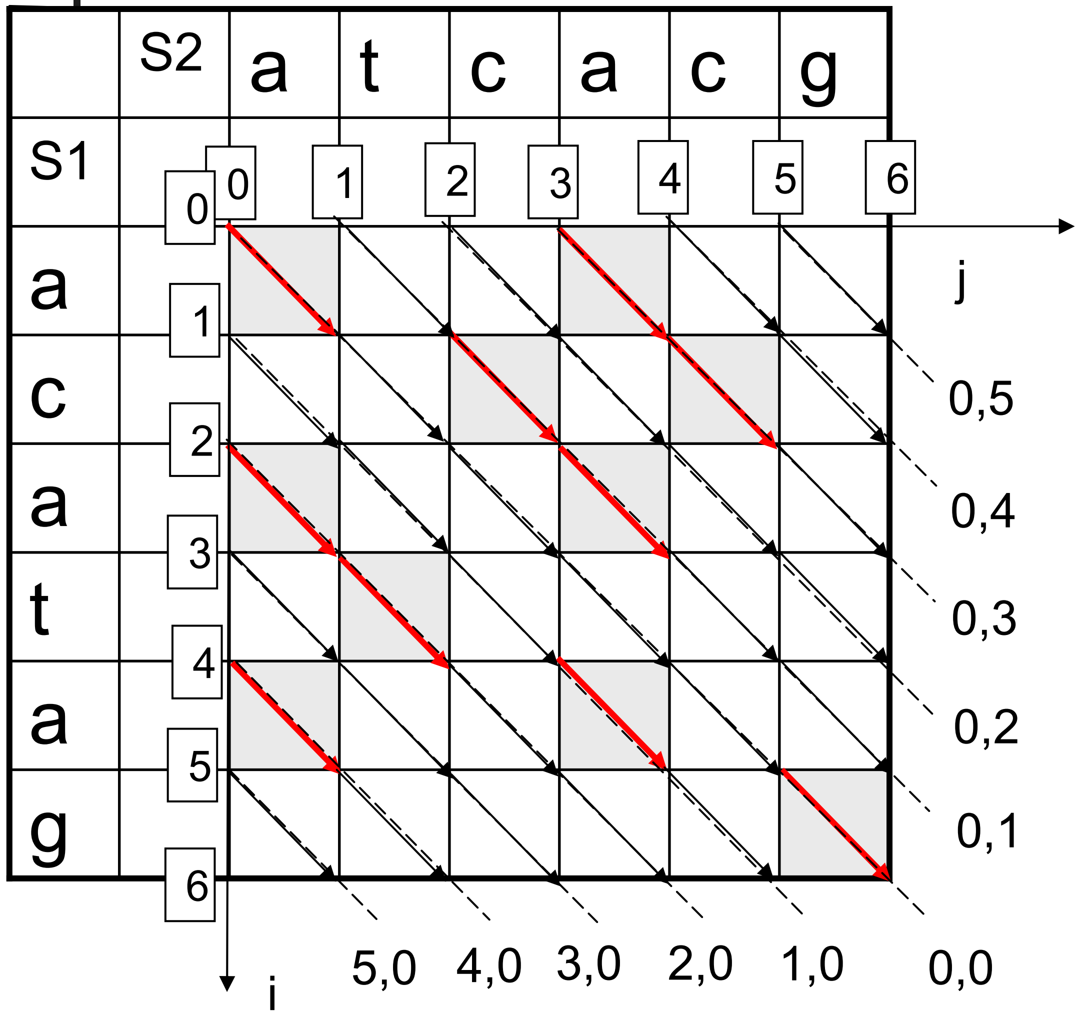
The pseudocode of the Hirschberg’s algorithm can be found at:

http://en.wikipedia.org/wiki/Hirschberg%27s_algorithm

Time complexity

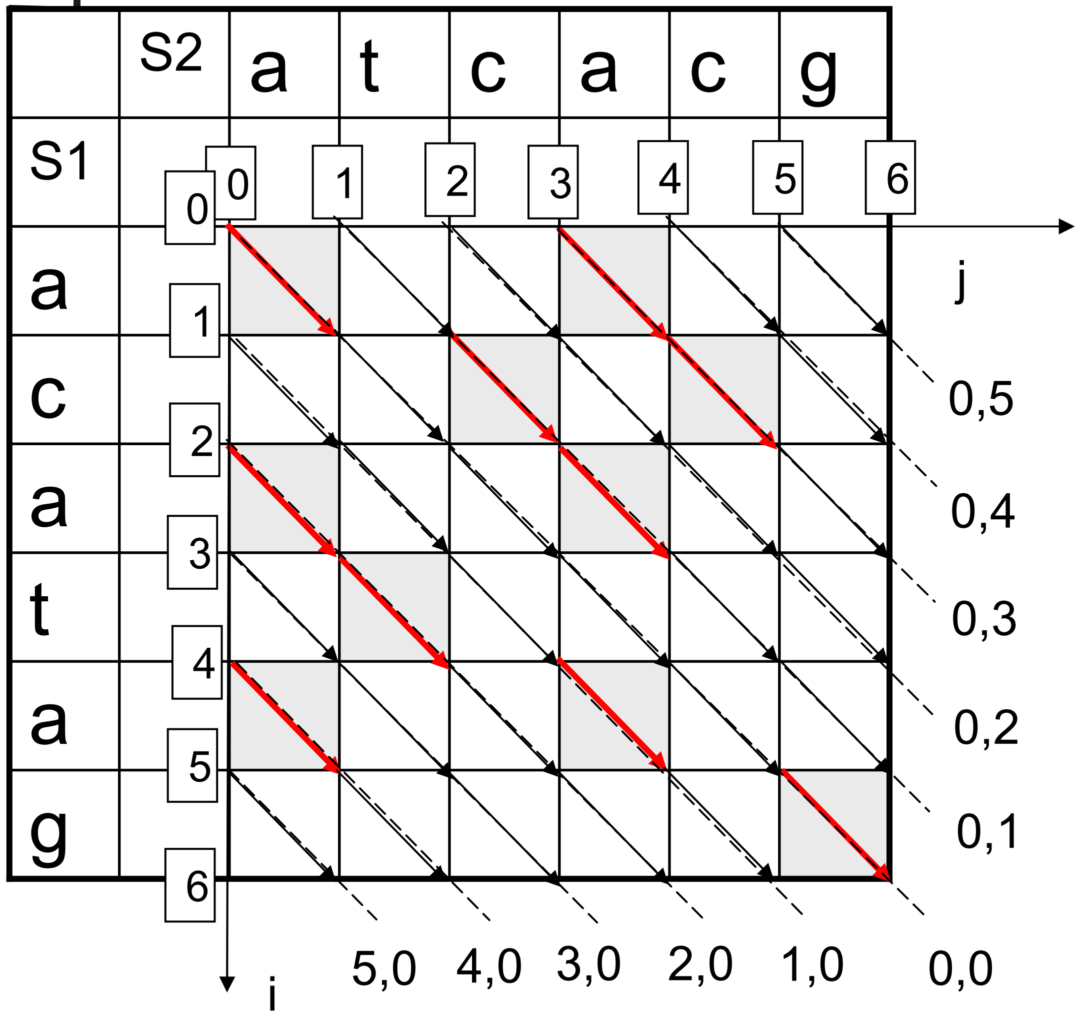
- Improving the O(NM) running time

Algorithm by Miller & Myers (The MM algorithm)



The main idea of the MM algorithm is to move as far as possible through a given diagonal of the grid graph, following the sequence of matches

The MM algorithm: definitions



Diagonals:

Name each diagonal according to the coordinates of its starting point

The 2 *neighbor diagonals* of diagonal (0,0) are:

diagonal (1,0)

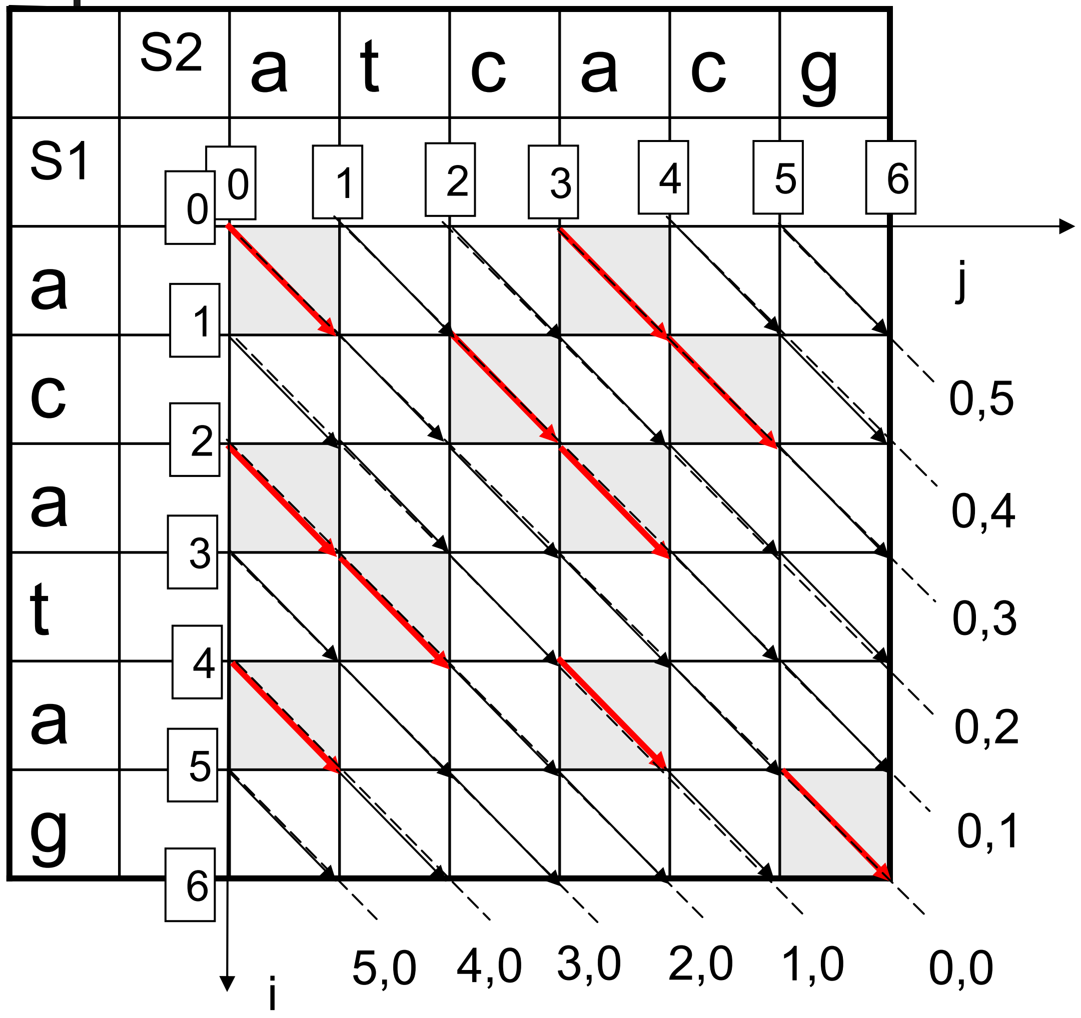
and diagonal (0,1)

The 2 neighbor diagonals of diagonal (0,2) are

diagonal (0,1)

and diagonal (0,3)

The MM algorithm: definitions

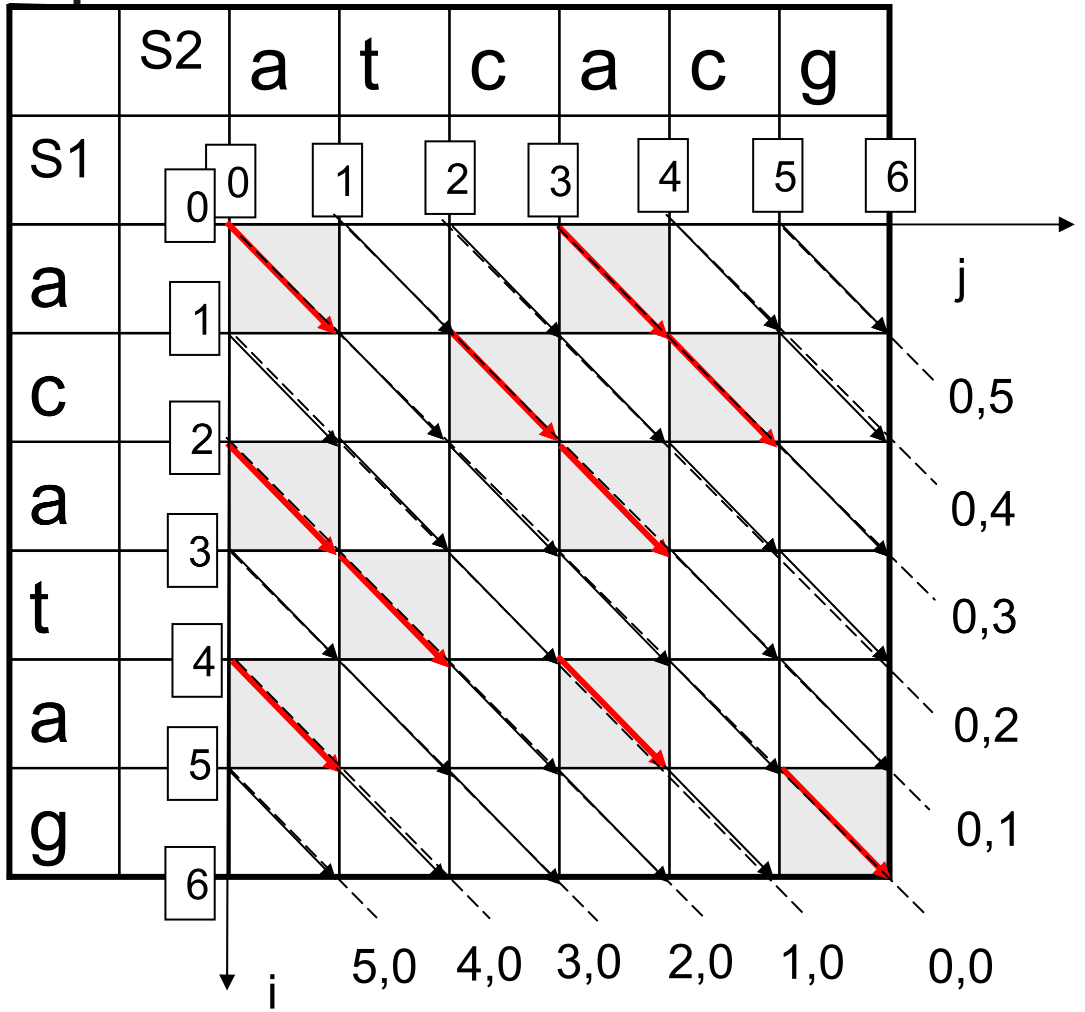


A *d*-path in the edit graph is a path which starts at point (0,0) and has a cost exactly *d*

A d-paths can end only at d diagonals around the main diagonal

This is because we cannot move from the main diagonal to (d+1,0) or (0,d+1) diagonal in less than d+1 insertions (deletions)

The MM algorithm

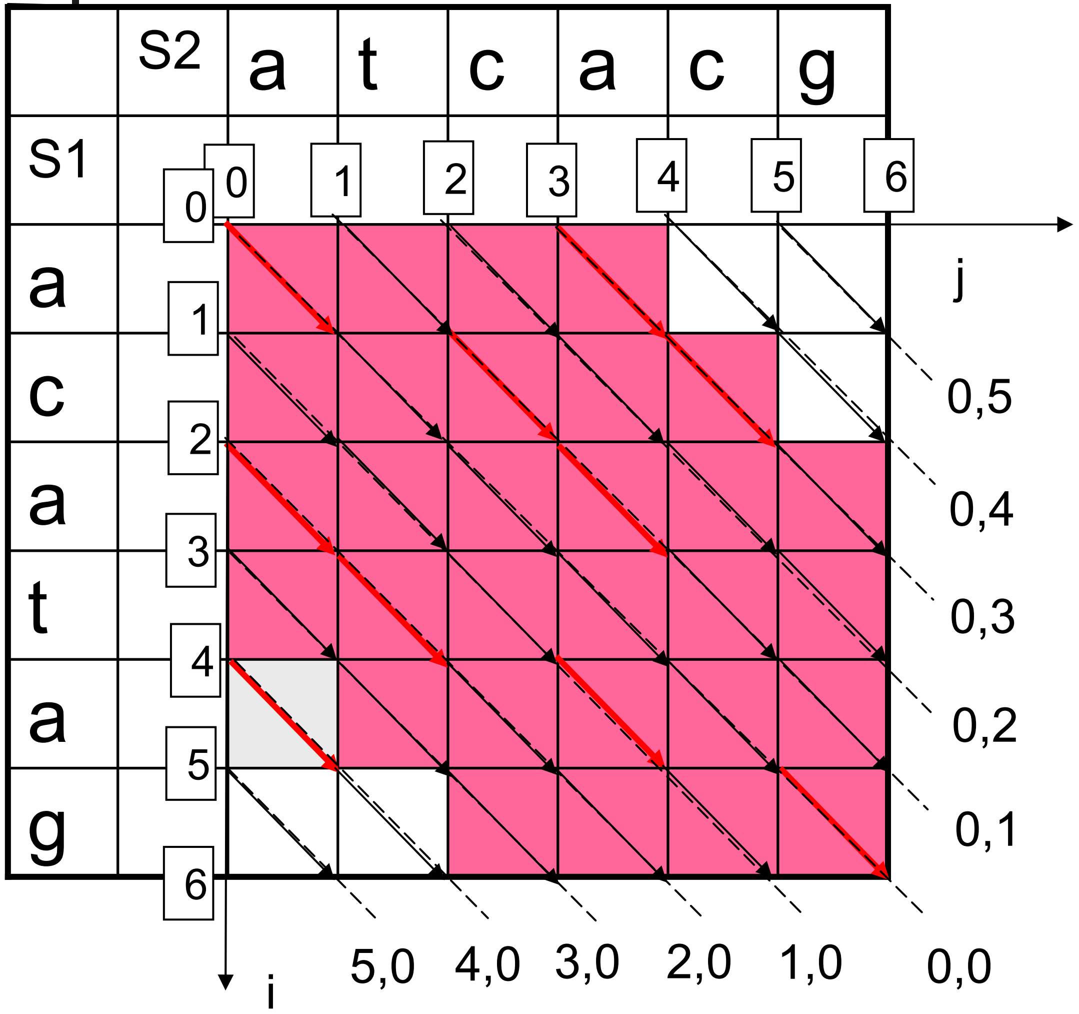


The algorithm performs an initialization and *D* iterations, where

D is an edit distance between S1 and S2

In each iteration *d* the algorithm builds all *d*-paths, extending the *d*-1-paths

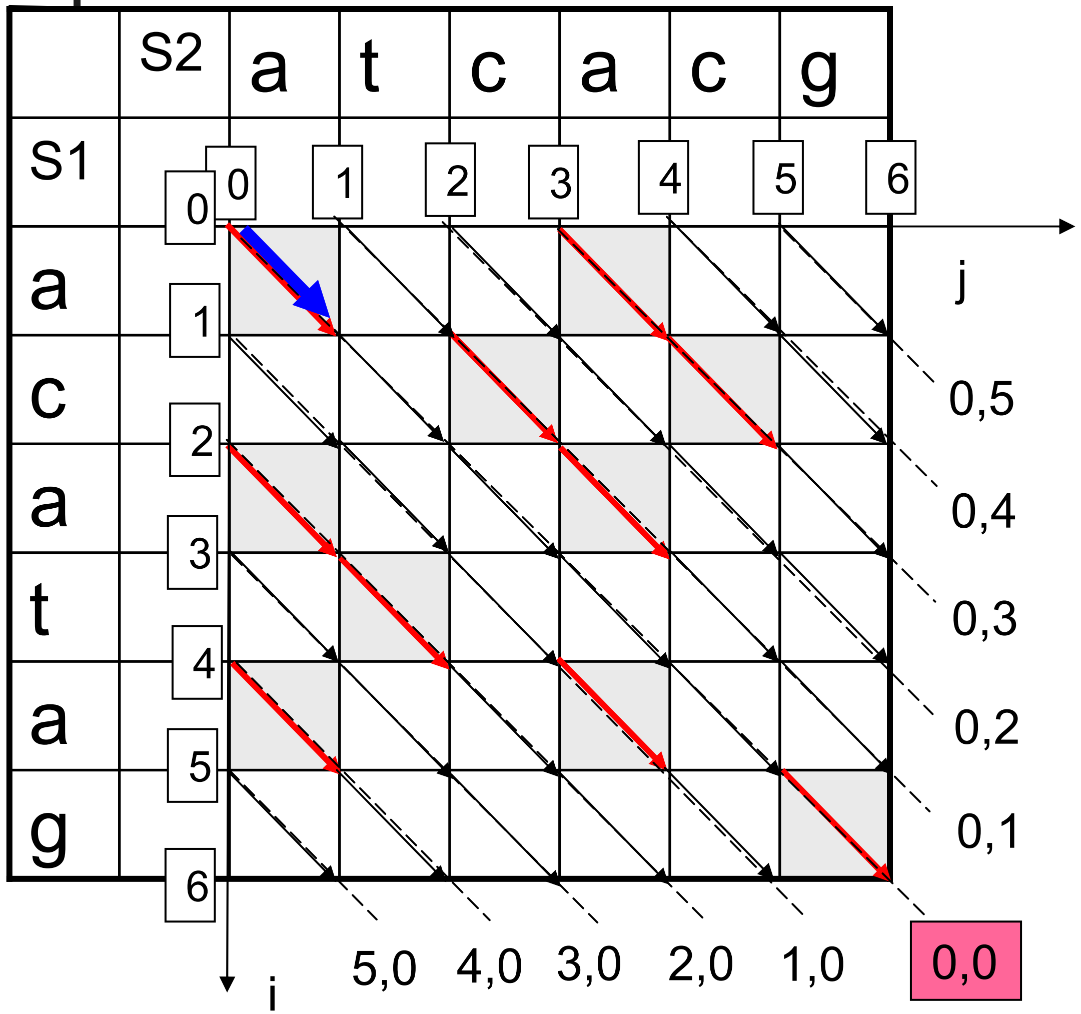
The MM algorithm



The key observation is that id the final edit distance is *D*, we only need to compute the grid values in a strip 2D+1 around the main diagonal

And we actually need to compute the values in at most (2*D*+1)·*N* grid cells, such obtaining the O(ND) algorithm

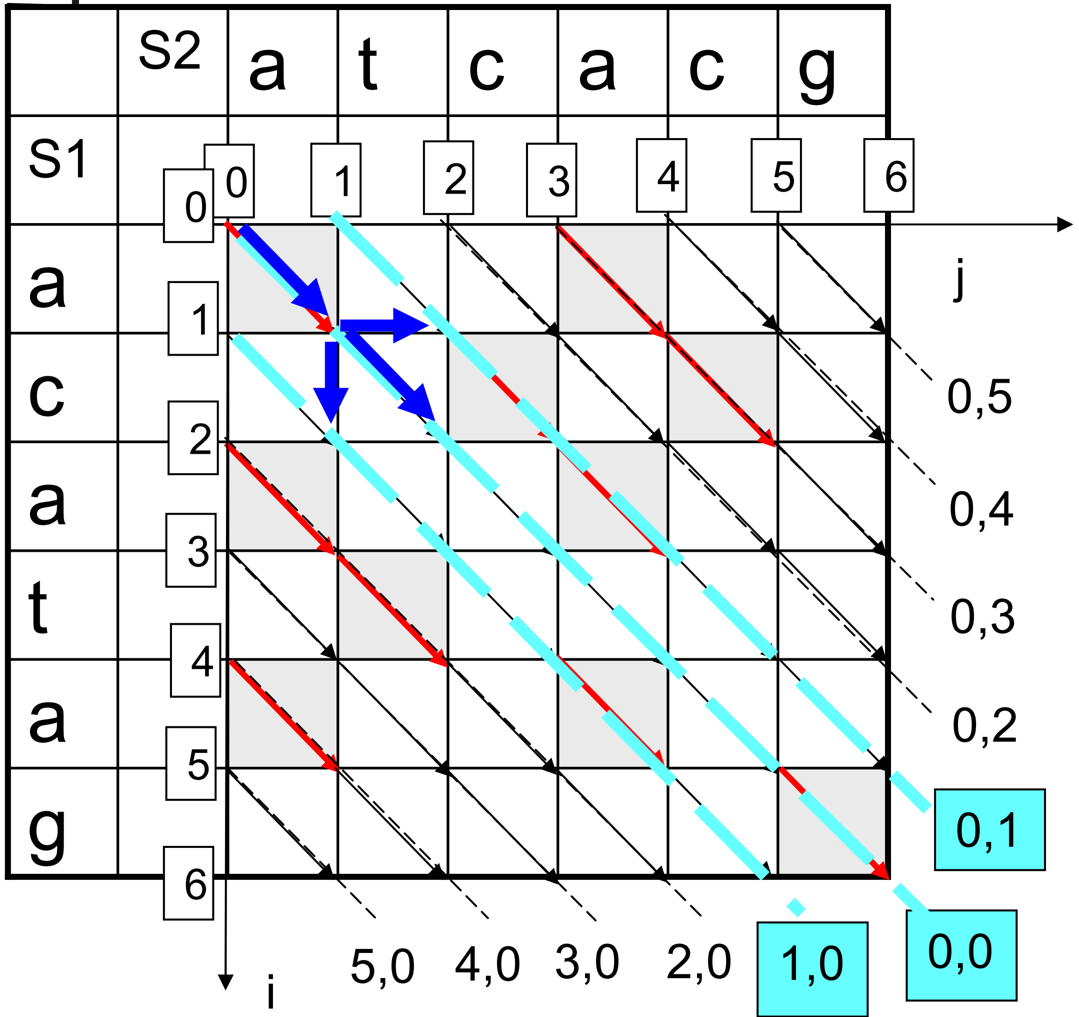
The MM algorithm. Iteration 0



In the initialization phase, we build the path of cost 0.

There is only one possible path of a total cost 0, which starts at a source point (0,0) and runs along the main diagonal through the sequence of character matches

The MM algorithm. Iteration 1

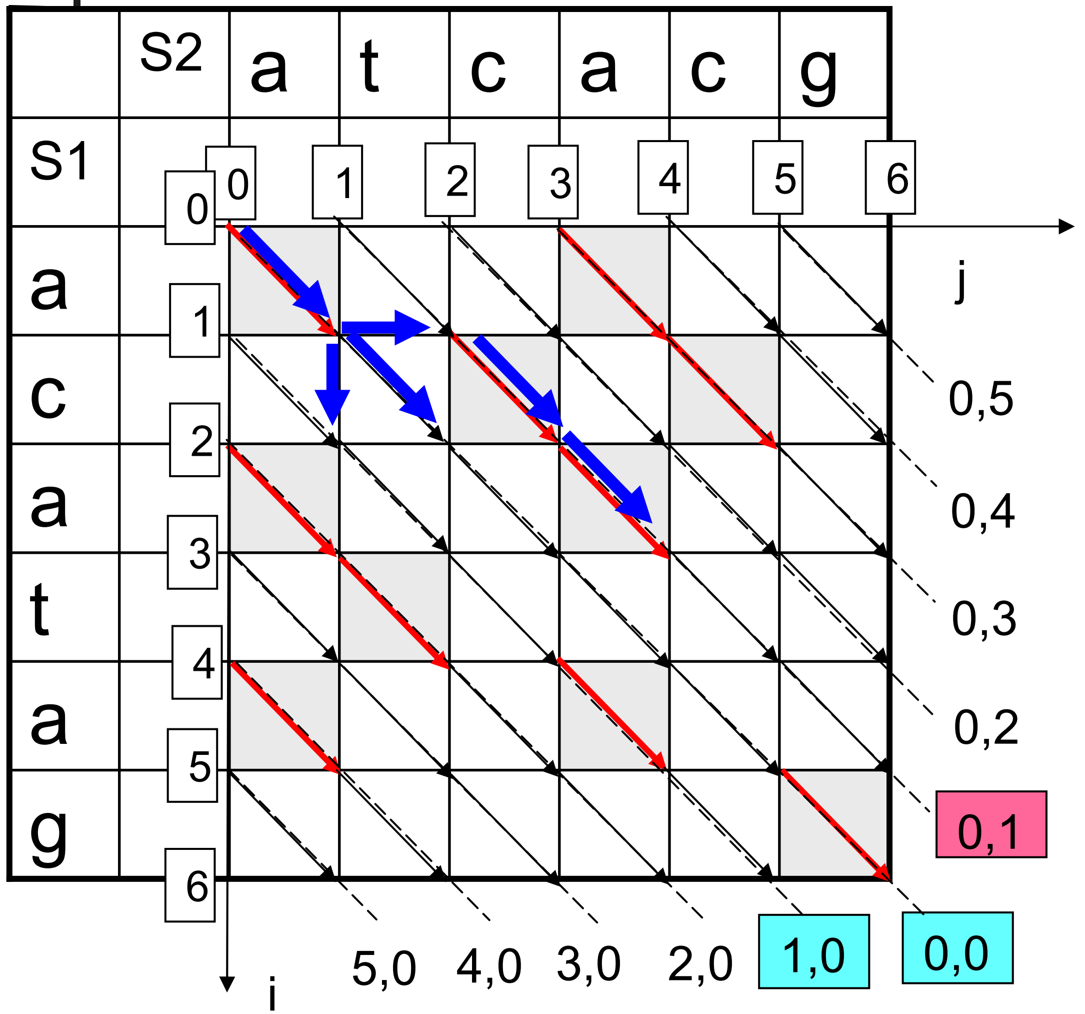


We produce all possible paths with a total cost 1.

There can be only 3 possible paths with the cost 1 and they end at:
the main diagonal (0,0)
and its 2 neighbor diagonals

In order to find these paths, we extend the 0-cost path with 1 edit operation, reaching each of the two neighbor diagonals with a jump of cost 1 and adding a mismatch to the end of a 0-path on the main diagonal

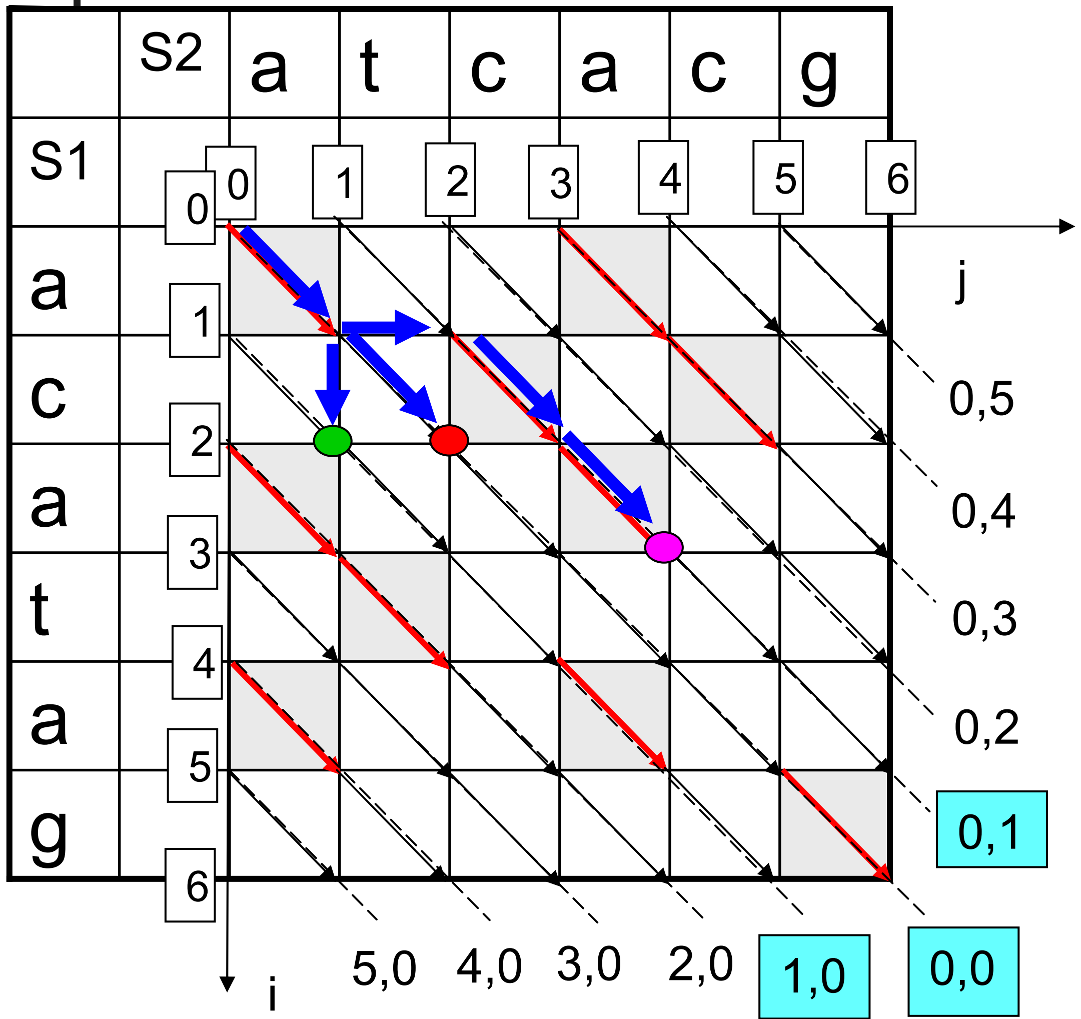
The MM algorithm. Iteration 1



We produce all possible paths with a total cost 1.

Then we extend the end of each such path with a series of consecutive matches running as far as possible down the corresponding diagonal, such obtaining all possible paths of a total cost 1

The MM algorithm. Iteration 1

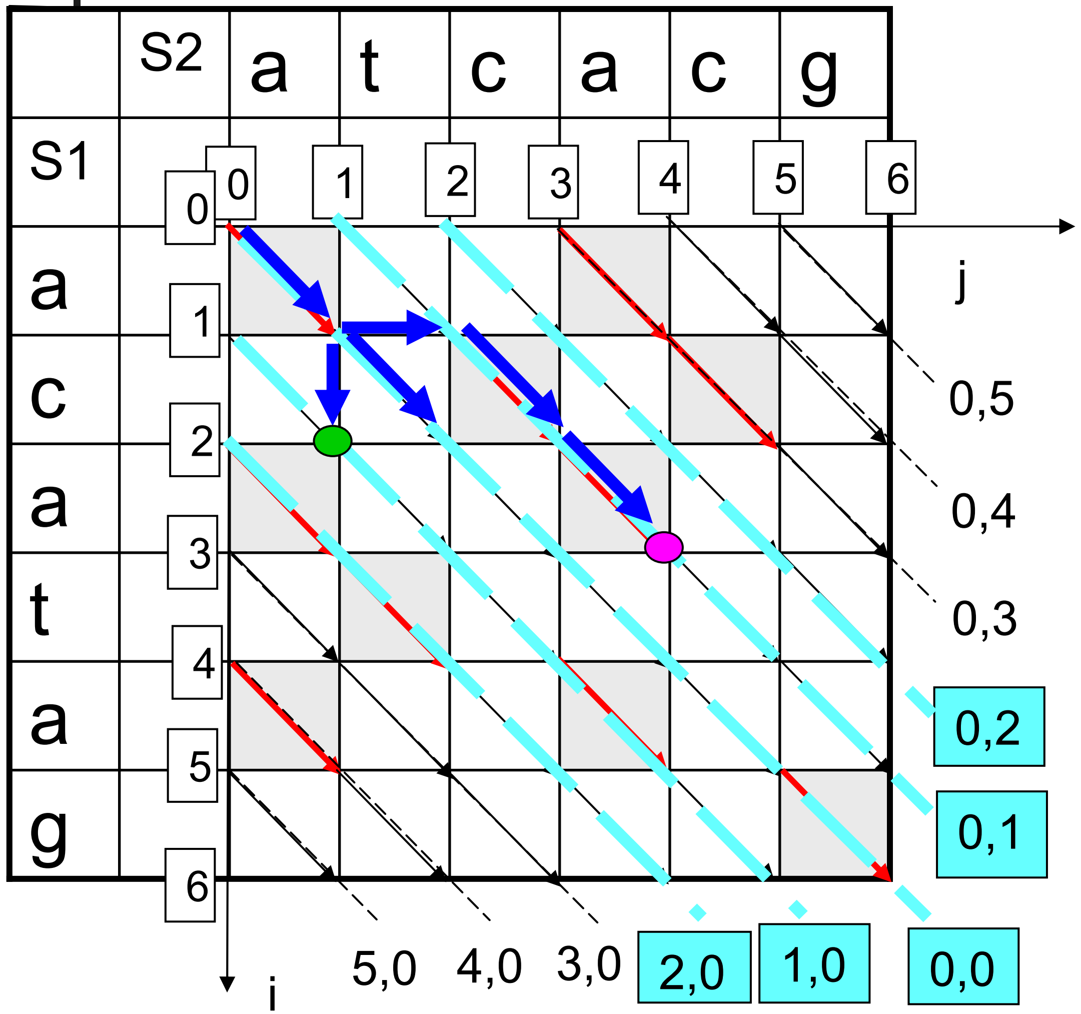


We produce all possible paths with a total cost 1.

The ends of all paths of a total cost 1:

🟢 🔴 🟣

The MM algorithm. Iteration 2.

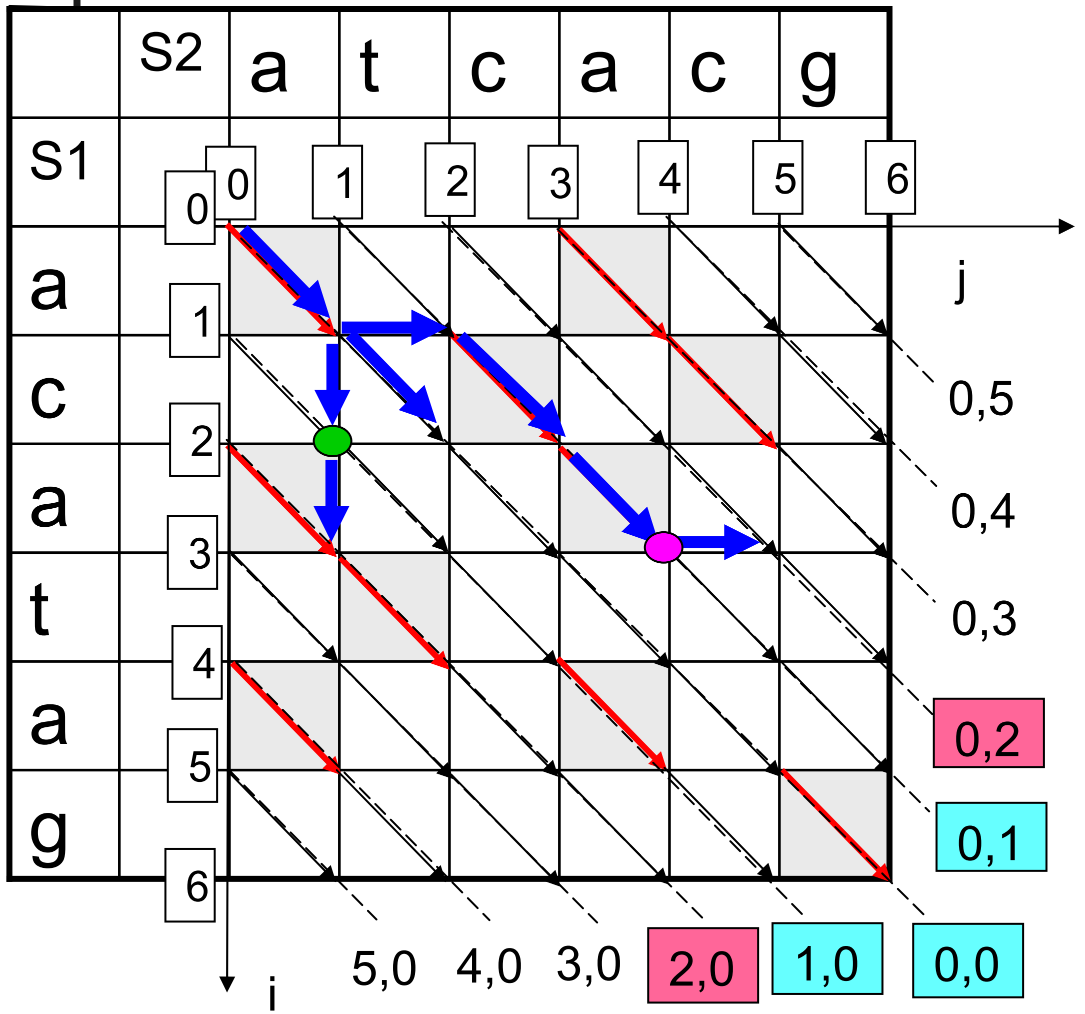


We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

Since the paths which end at all other diagonals, for example (0,3), involve at least 3 edit operations of moving from the main diagonal to the corresponding diagonal.

The MM algorithm. Iteration 2.

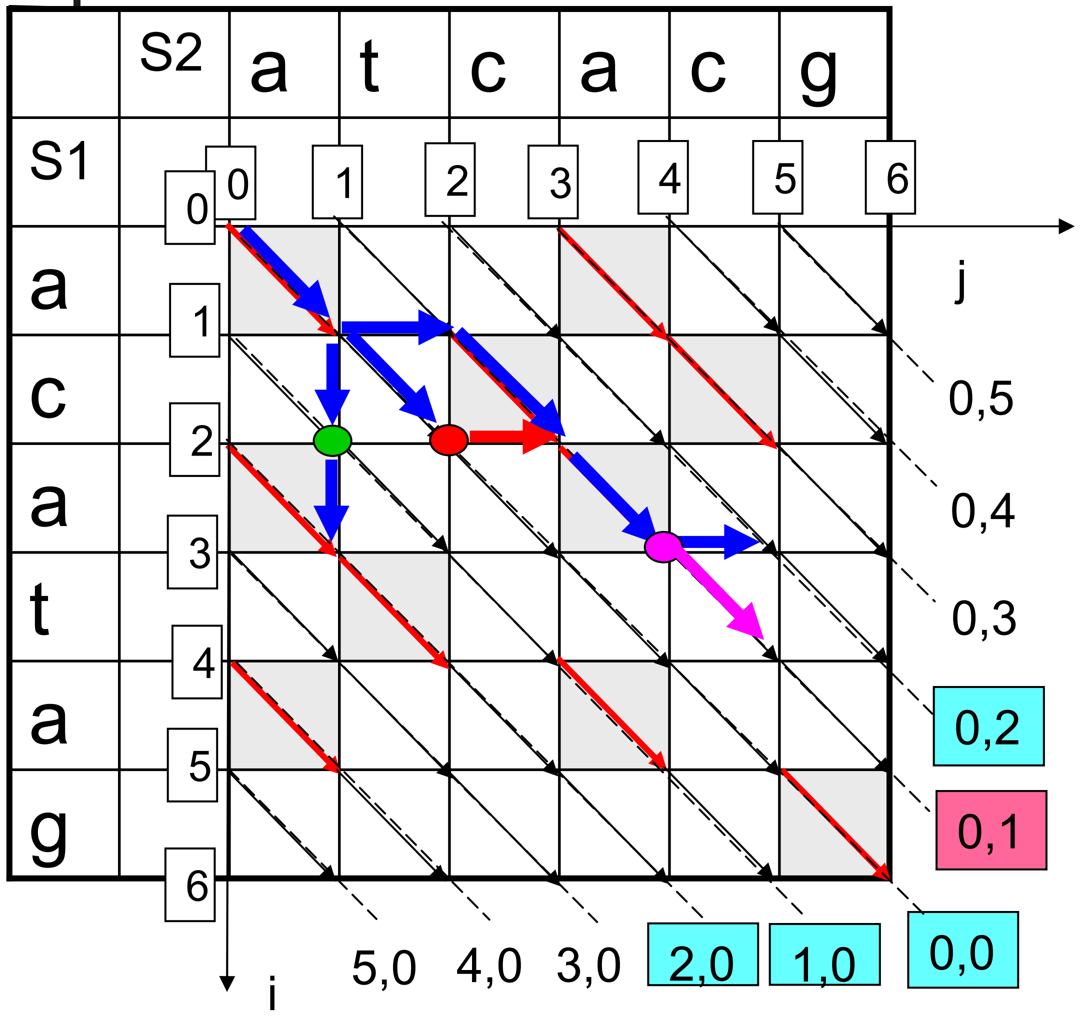


We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

First, we find the paths of the total cost 2 which end at diagonal (0,2) – by adding a jump from the end of the best path with the cost 1 from diagonal (0,1)
and at diagonal (2,0) – extending the path ended at diagonal (1,0)

The MM algorithm. Iteration 2. Dynamic programming

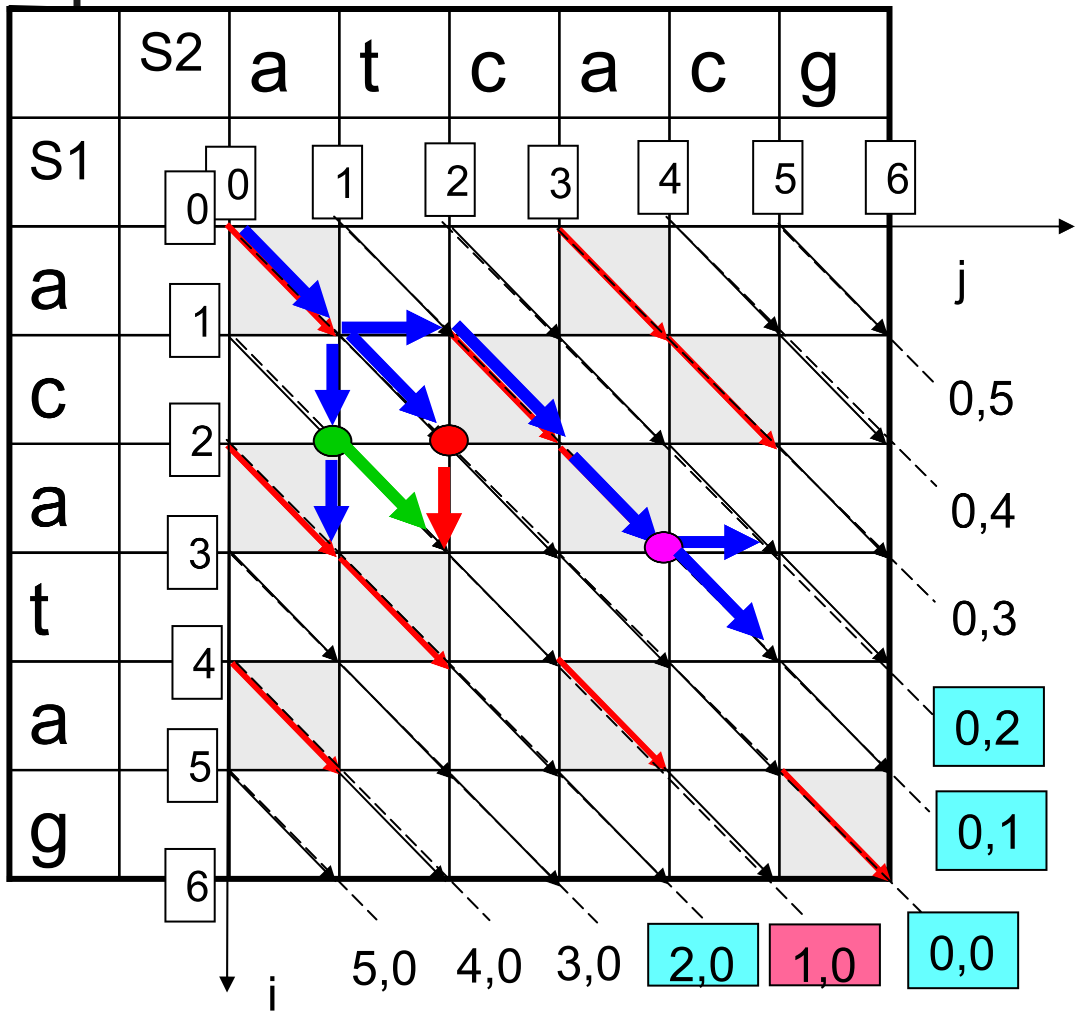


We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

For diagonal (0,1) there are 2 possible ways of obtaining paths of cost 2: by adding 1 mismatch from ●
or by adding 1 horizontal jump from ●
We choose from between 2 the extension of a previous path which runs further along the diagonal: ●

The MM algorithm. Iteration 2. Dynamic programming

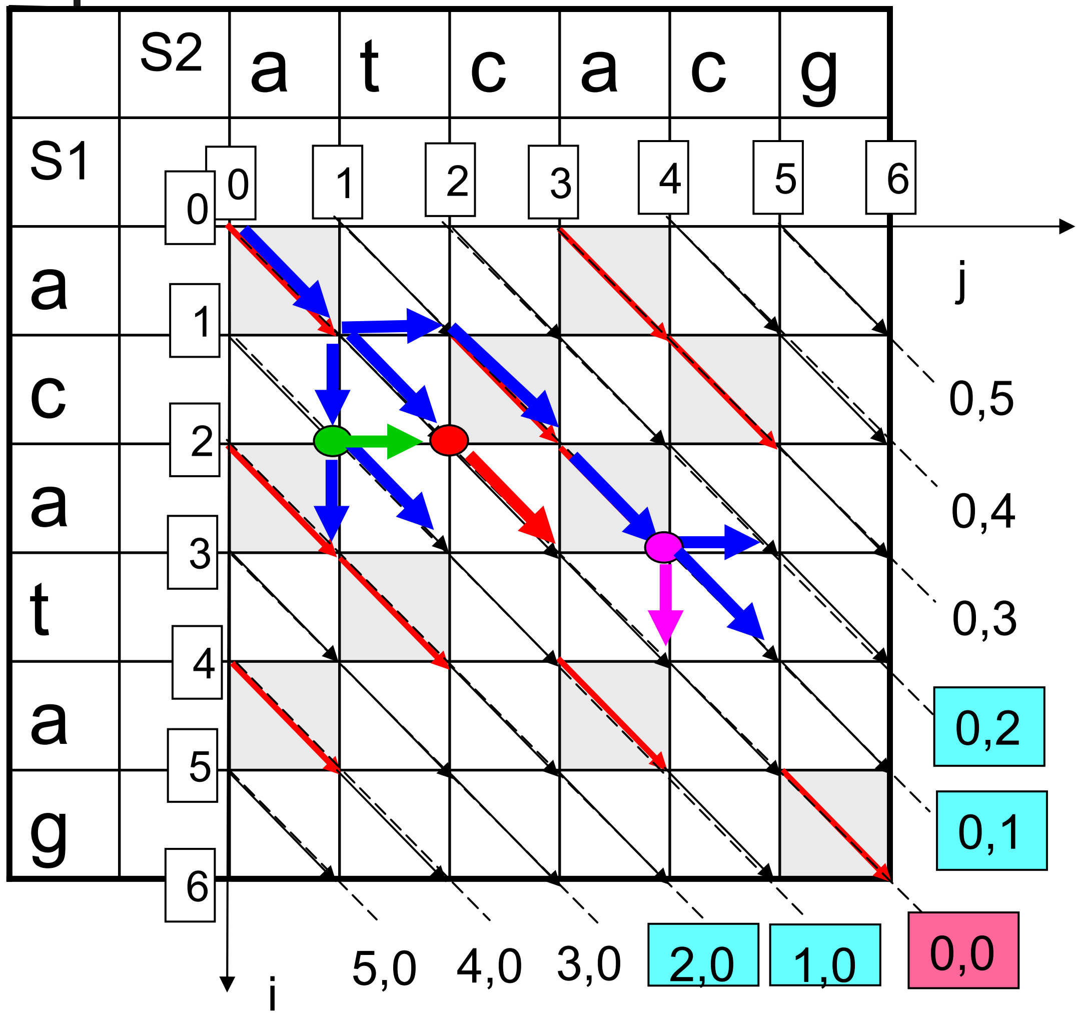


We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

The same logic is applied for diagonal (1,0)
In this example both extensions 🟢 🔴
are of equal quality, so we chose one of them: 🟢

The MM algorithm. Iteration 2. Dynamic programming



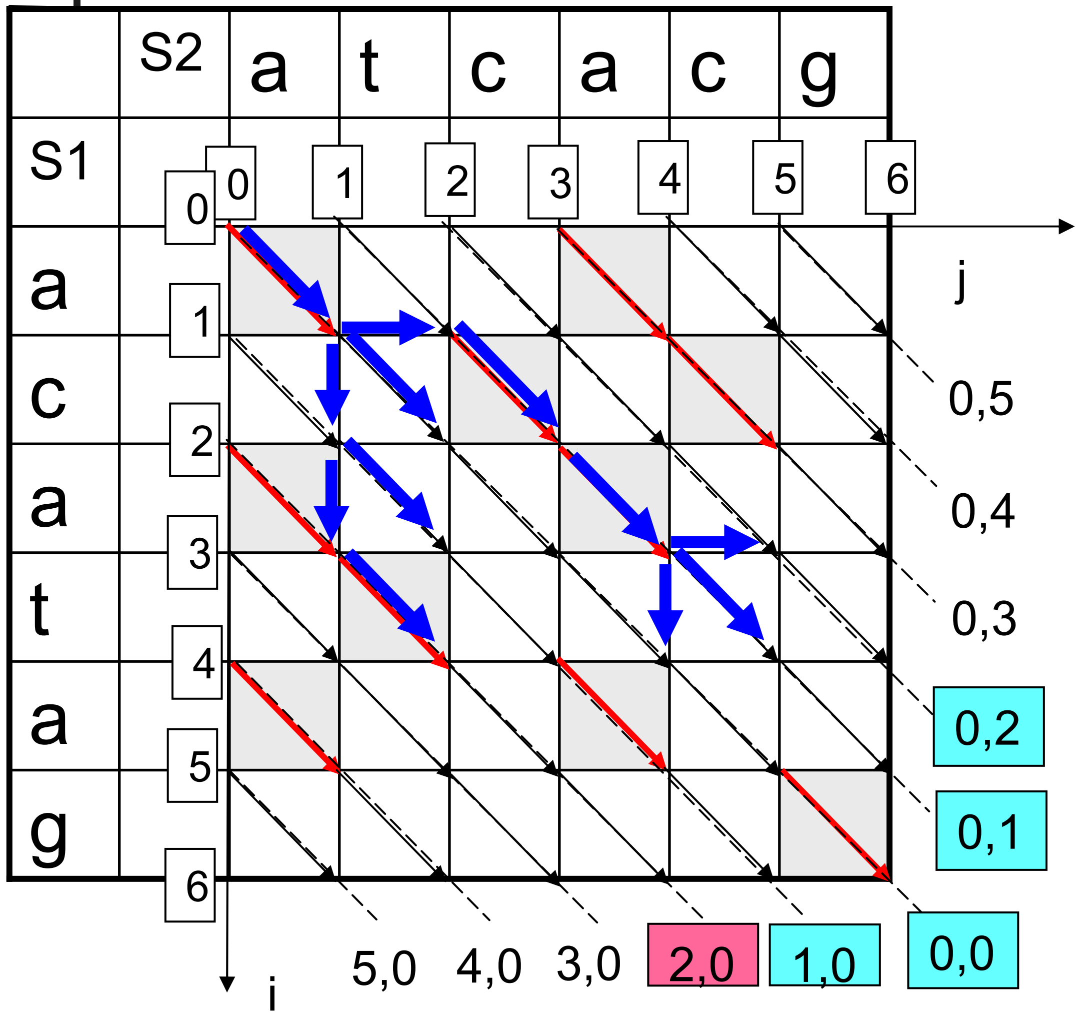
We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

For diagonal (0,0) there are 3 possible extensions:
● ● ●

We choose the furthest reaching along this diagonal: ●

The MM algorithm. Iteration 2. Dynamic programming

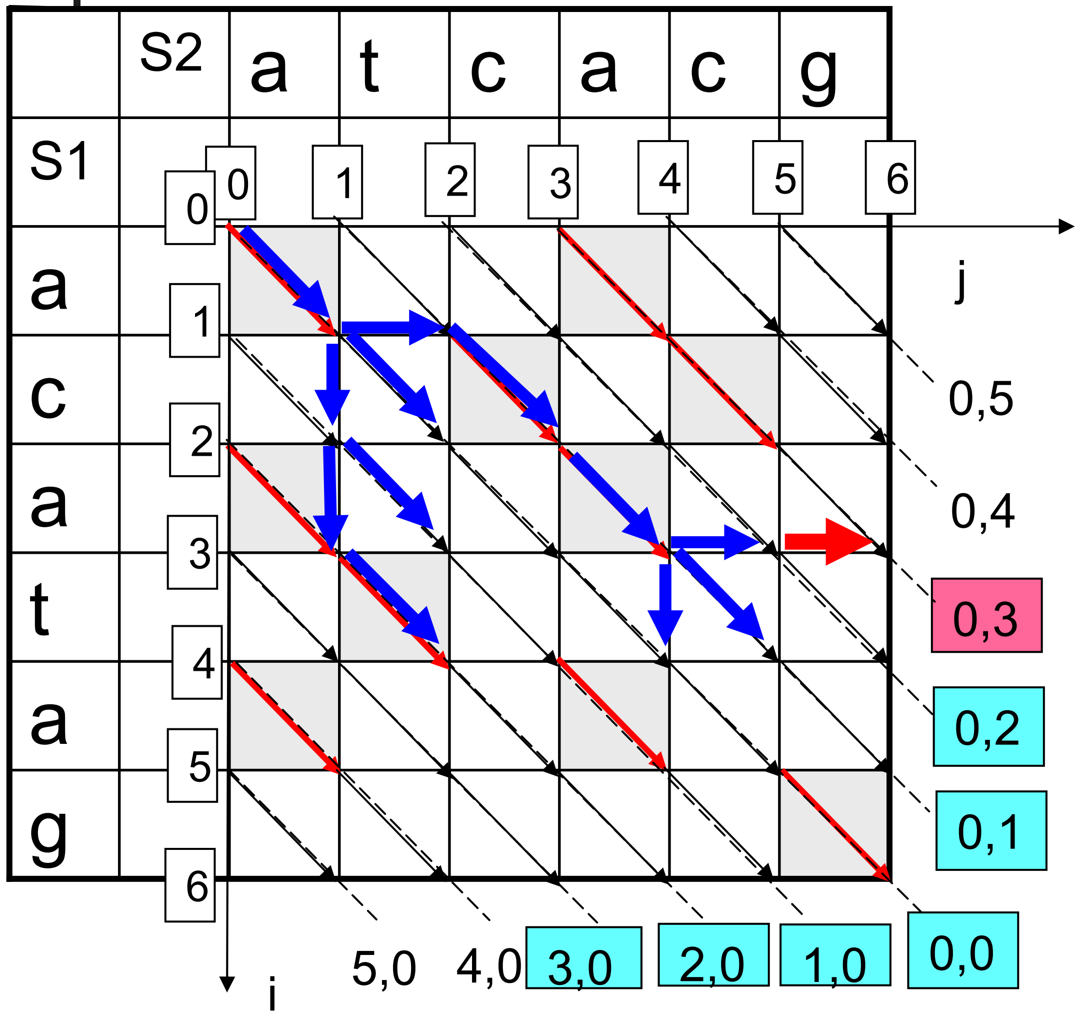


We produce all possible paths with a total cost 2.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (1,0) (2,0)

When the best path extensions are made for each diagonal, we extend the path for each diagonal with a series of matches, such obtaining all the paths with a total cost 2

The MM algorithm. Iteration 3. Dynamic programming

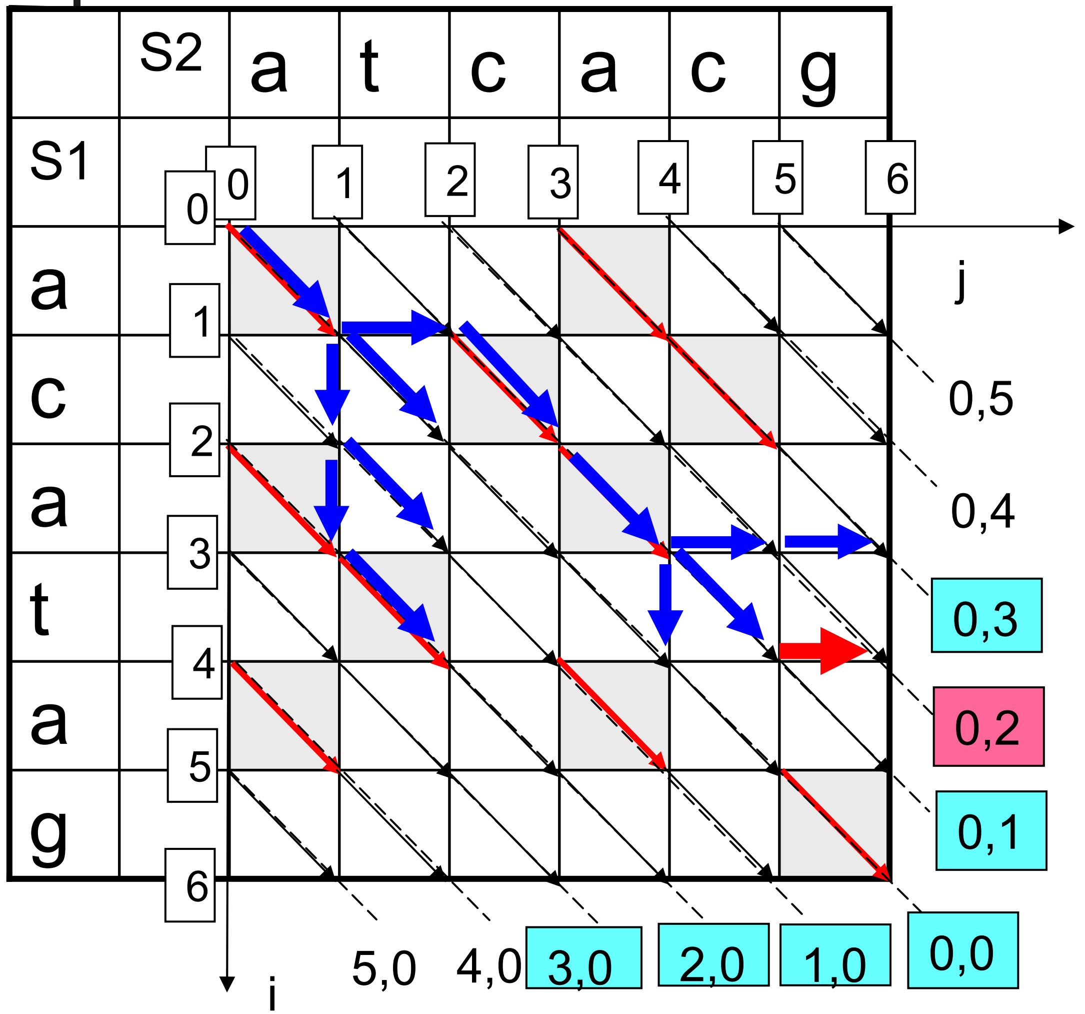


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

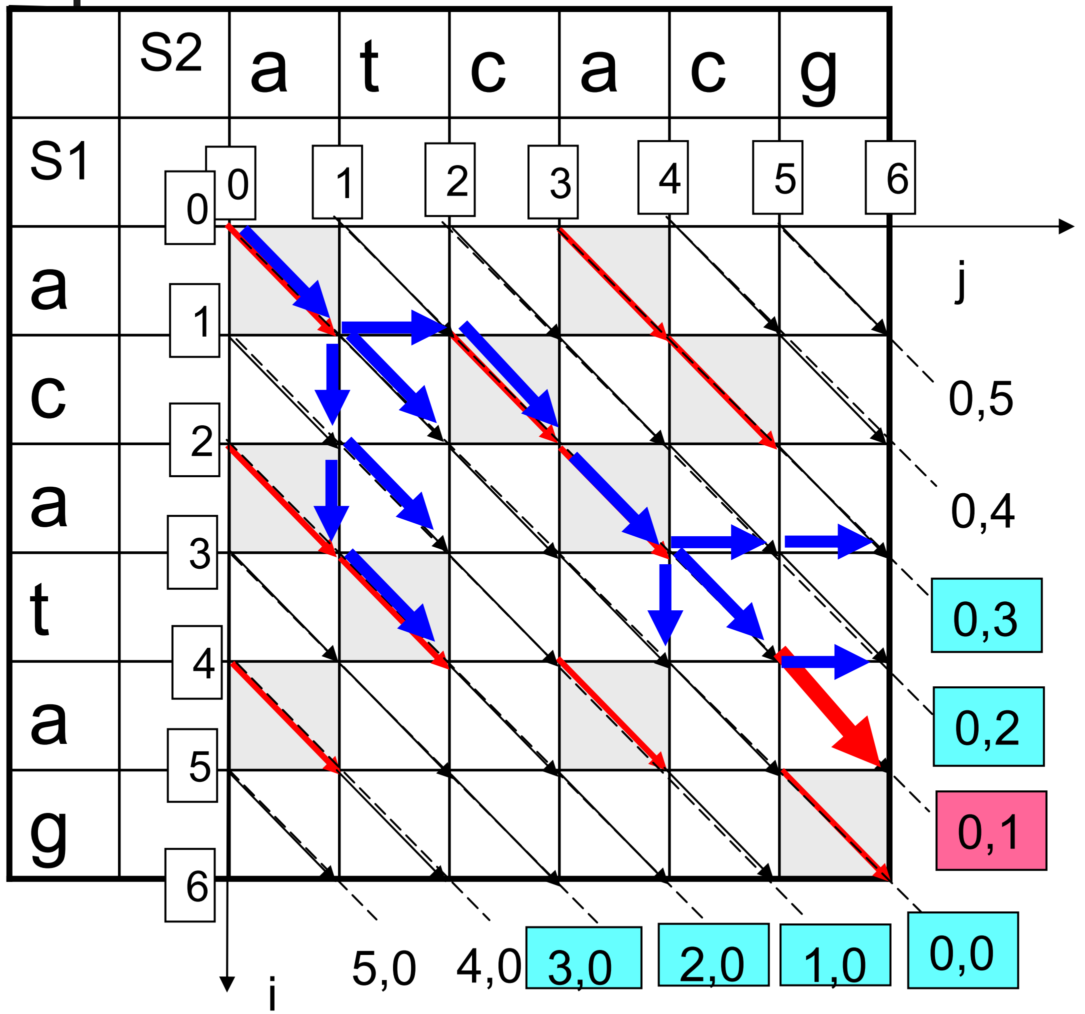


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

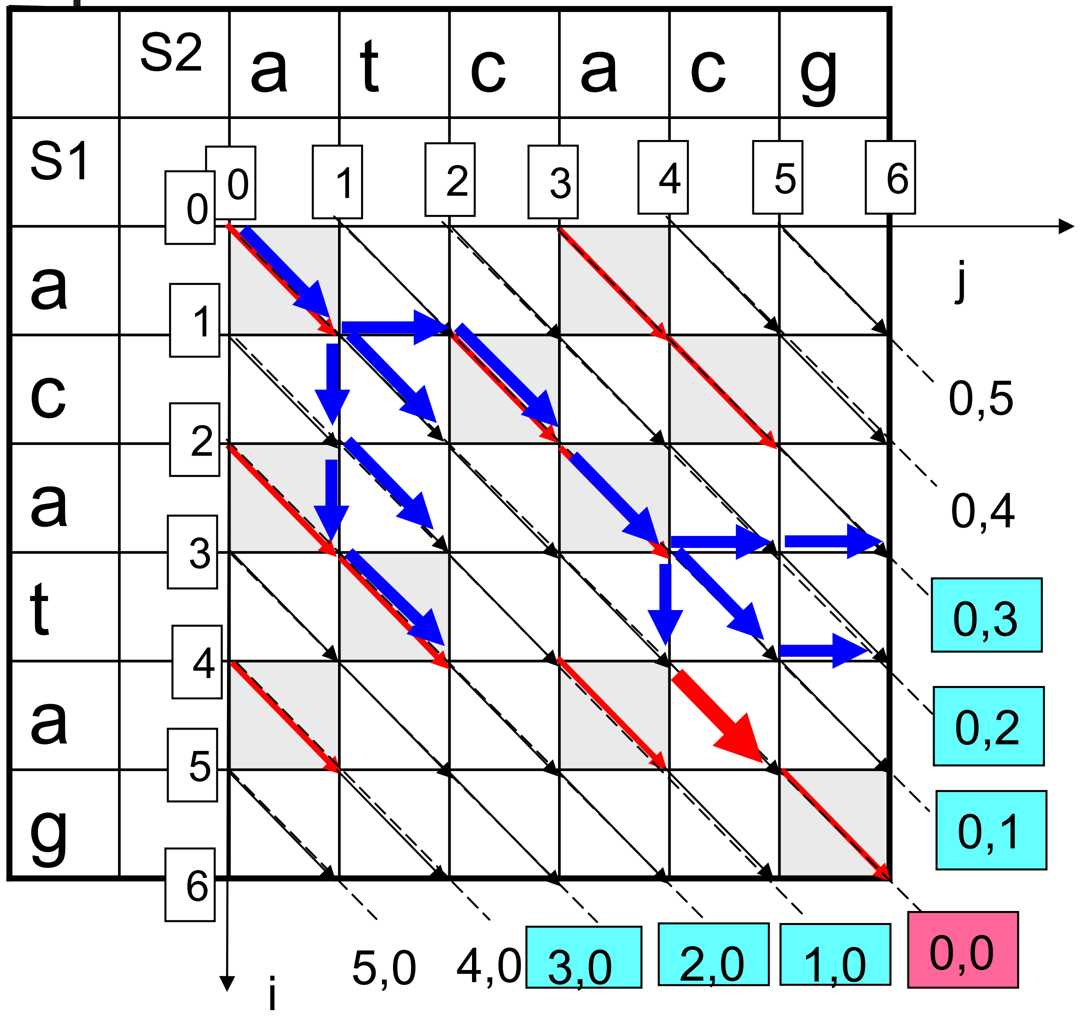


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

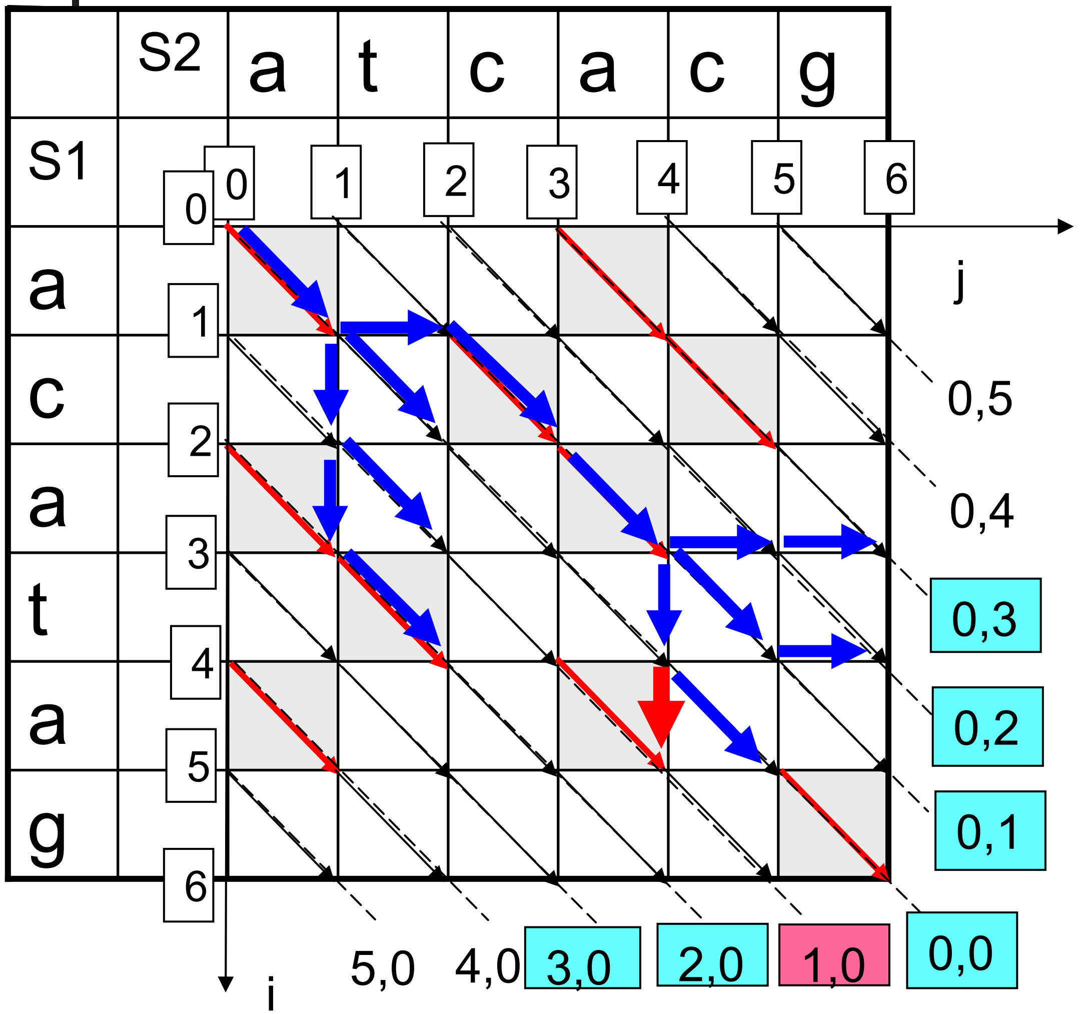


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

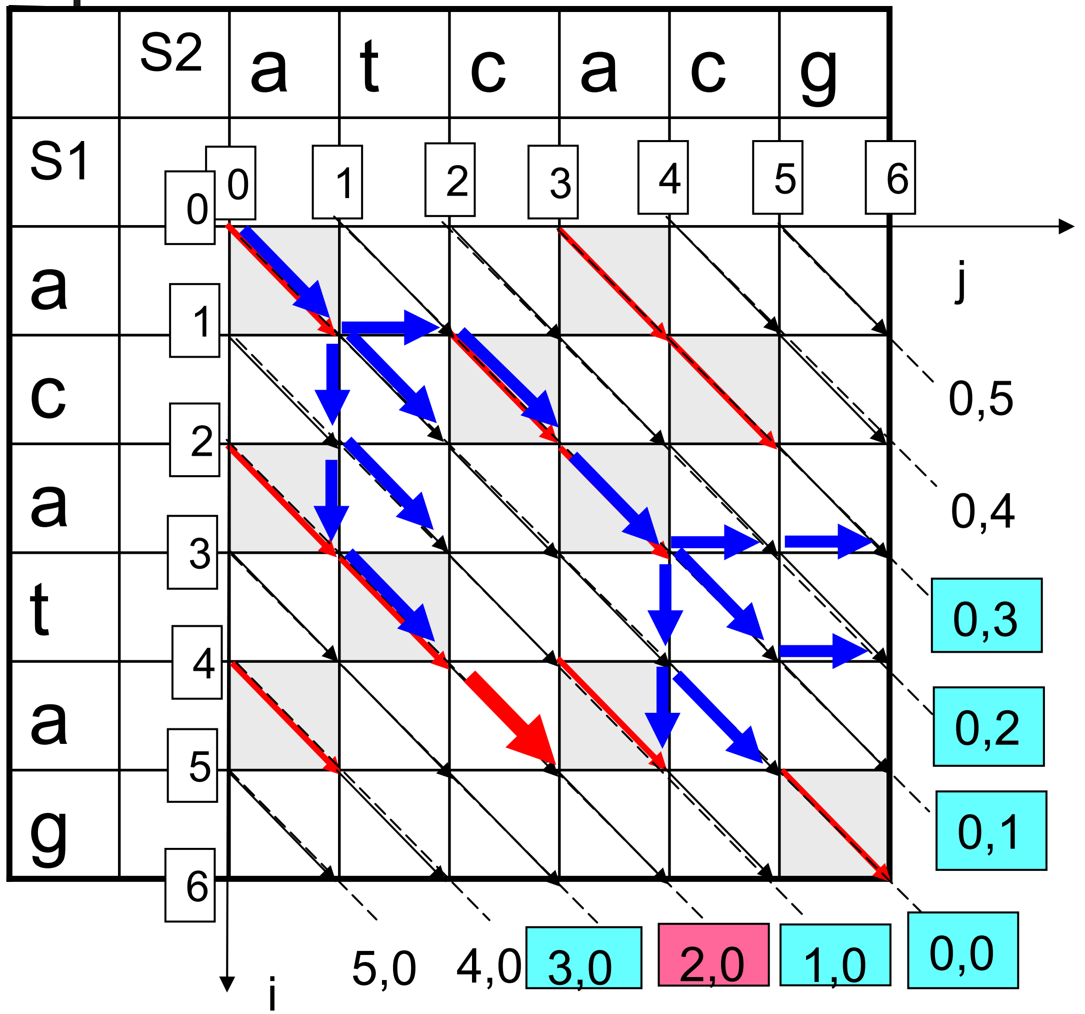


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

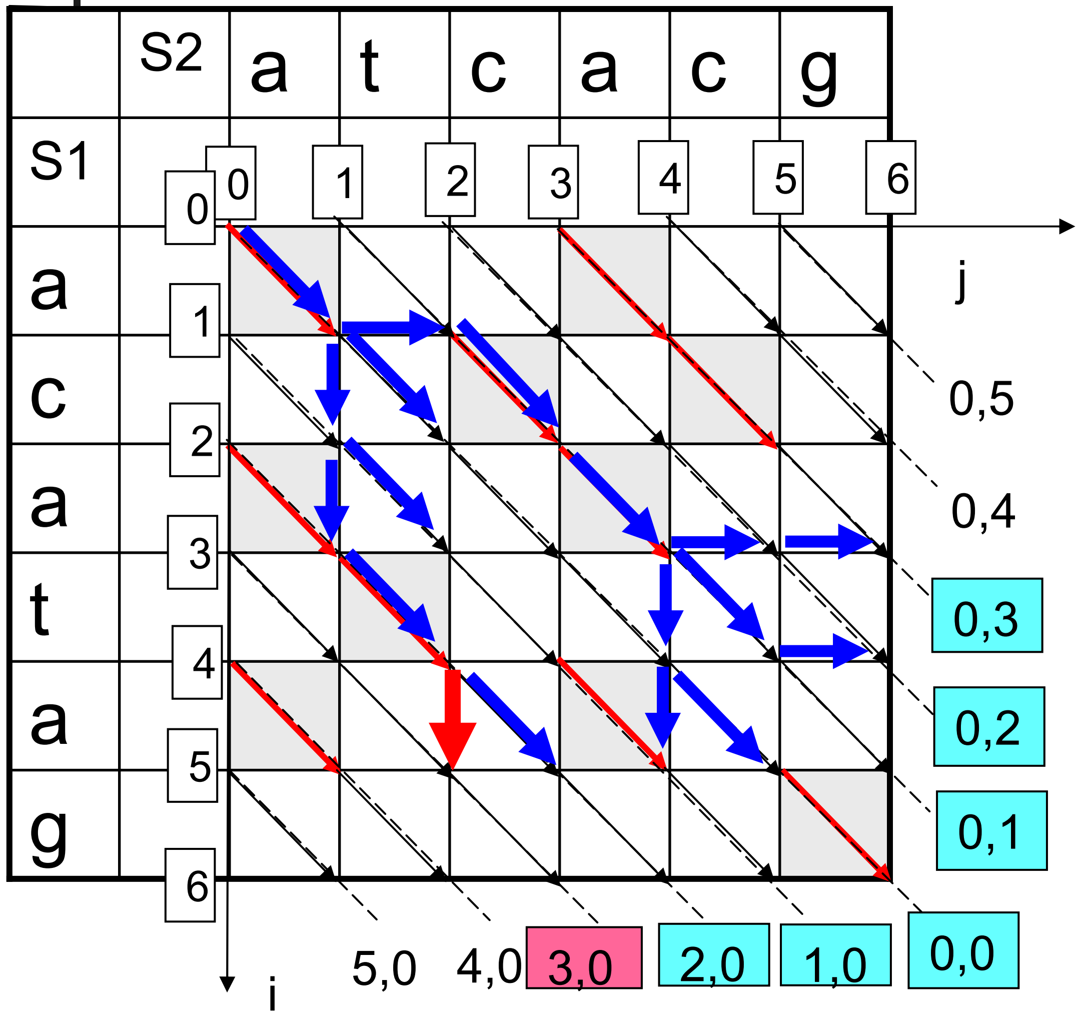


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

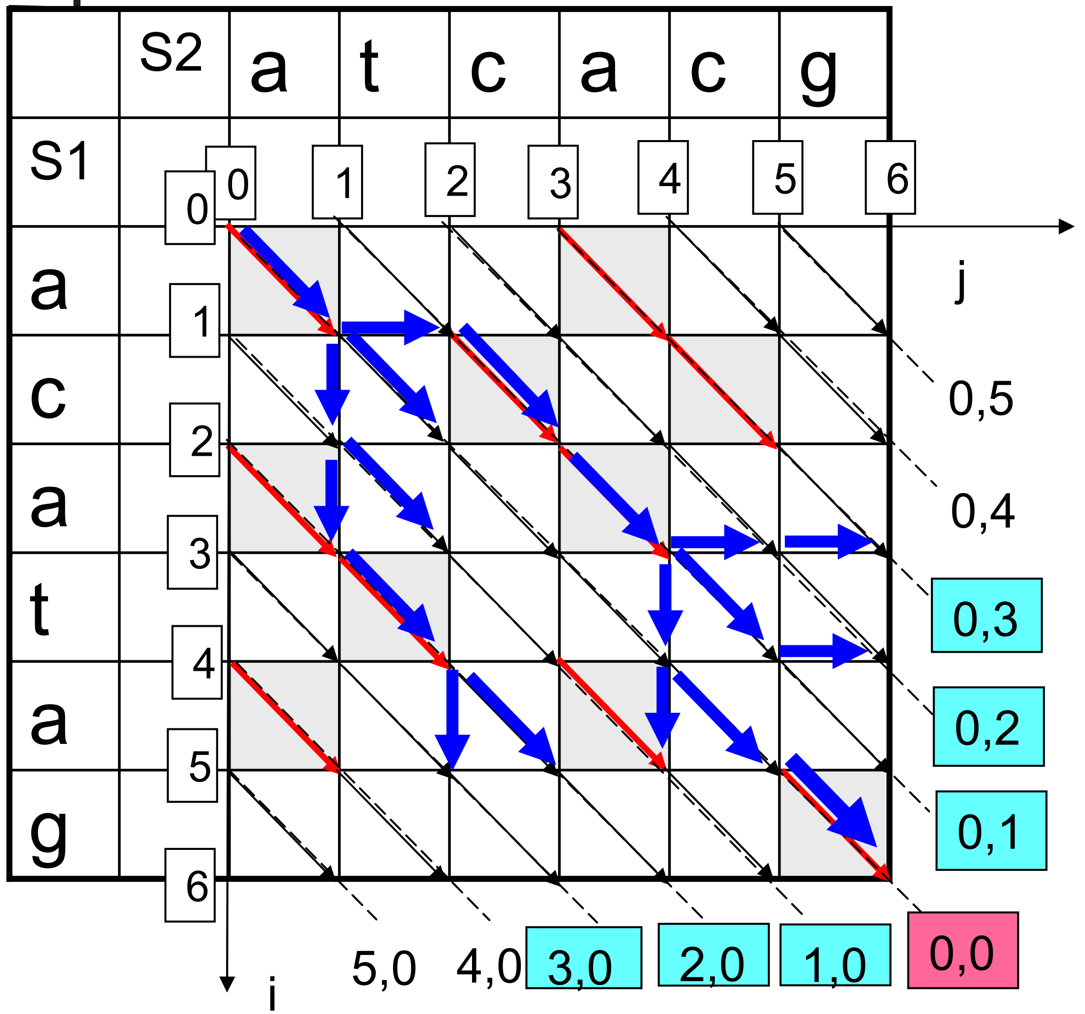


We produce all possible paths with a total cost 3.

These paths can end only at diagonals:
(0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

We apply the same dynamic programming approach as in iteration 2 for each such diagonal in turn

The MM algorithm. Iteration 3. Dynamic programming

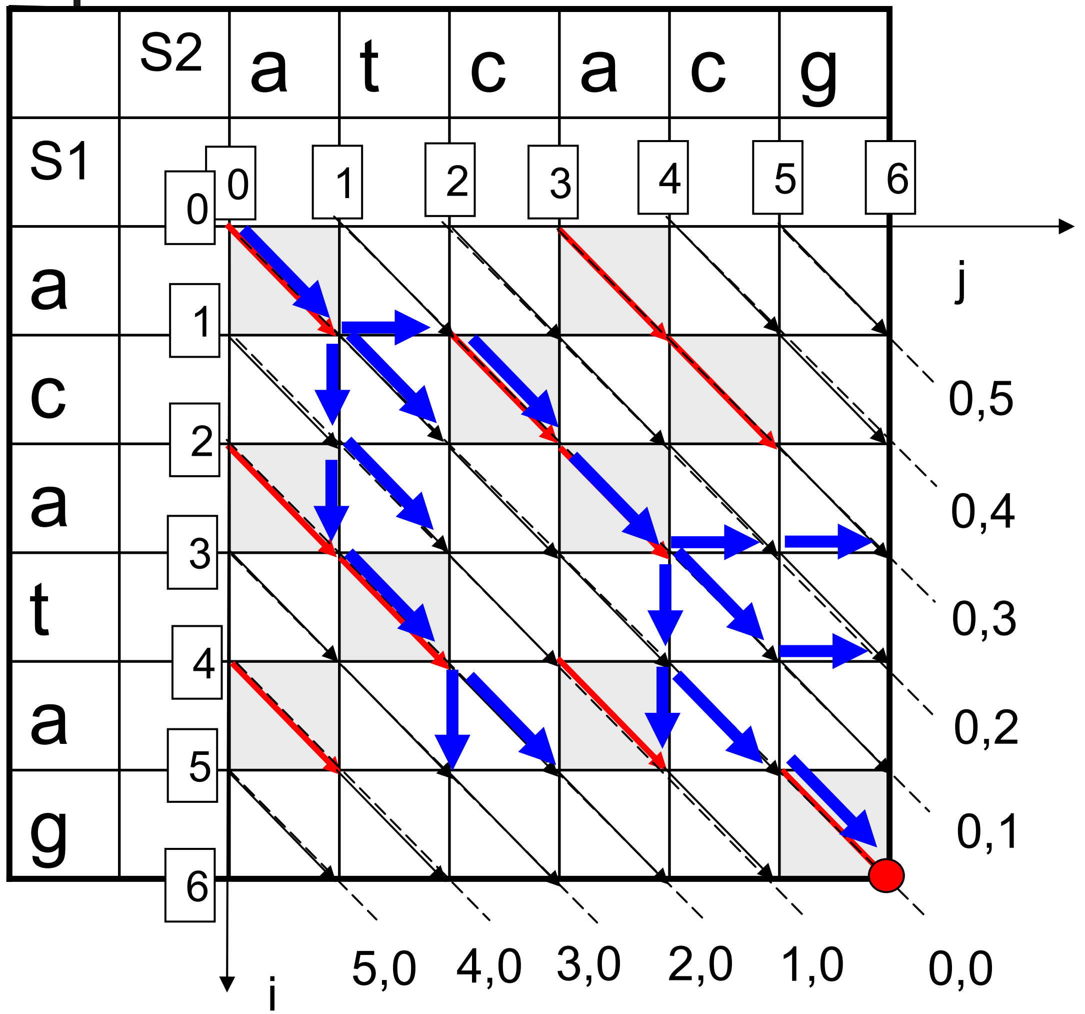


We produce all possible paths with a total cost 3.

These paths can end only at diagonals: (0,0) (0,1) (0,2) (0,3) (1,0) (2,0) (3,0)

Next, we extend each path with a series of matches along the corresponding diagonal

The MM algorithm. The destination

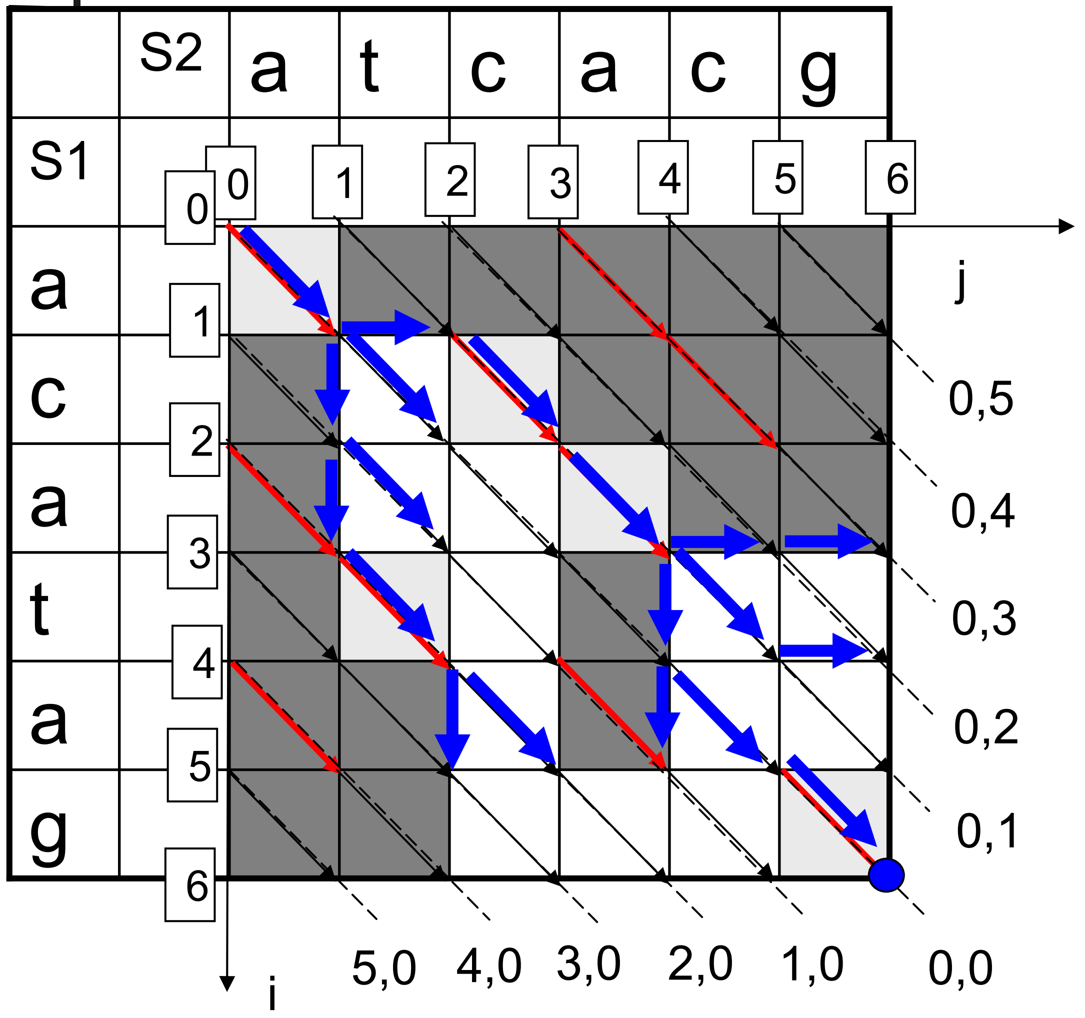


We produce all possible paths with a total cost 3.

At this point, one of paths with a total cost 3 has reached the destination – point (6,6)

The algorithm terminates, and *D*=3

The MM algorithm. The complexity



Note that we did not compare some possible pairs of symbols in S1 and S2 (dark grey)

We have worked with no more than $2D+1$ diagonals. The length of each diagonal is at most N (if $N>=M$)

So, the total complexity is $O(D \bullet N)$

Thus, the algorithm performs well for similar strings (with a small edit distance)

The MM algorithm – pseudocode I

```
algorithm MM_Edit_Distance ($S_1$, $S_2$)
    destinationReached:=false
    $d$:=0
    initializeDiagonalArrays()
    snake(0,0)
    while destinationReached=false do
        $d$: =$d$+1
        buildExtensions ($d$)
    return $d$
```

```
algorithm initializeDiagonalArrays()
//allocate arrays of end points for the paths for
    each diagonal
prevFrontier[N+M+1]
currentFrontier[N+M+1]

for $i$ from 1 to $N$:
    prevFrontier($i$,0):=(-1,-1)
for $i$ from 1 to $M$:
    prevFrontier(0,$i$):=(-1,-1)
prevFrontier(0,0):=(0,0)
```

The MM algorithm – pseudocode II

```
algorithm MM_Edit_Distance (S1, S2)
    destinationReached := false
    d := 0
    initializeDiagonalArrays()
    snake(0,0)
    while destinationReached = false do
        d := d+1
        buildExtensions (d)
    return d
```

```
algorithm buildExtensions (l)
    for i from l down to 1:
        currentFrontier(i,0) = bestExtension (i, 0)
        currentFrontier(0,i) = bestExtension (0,i)

    /* main diagonal at last */
    currentFrontier(0,0) = bestExtension (0,0)

    for i from l down to 1:
        prevFrontier(i,0) = currentFrontier (i,0)
        prevFrontier(0,i) = currentFrontier (0,i)
    prevFrontier(0,0) = currentFrontier (0,0)
```

The MM algorithm – pseudocode III

```
algorithm bestExtension (diagonal name (i,j))
    if i=0 and j=0: //the main diagonal
        pointFromAbove: = max ((0,0), (prevFrontier(0,1).X+1, prevFrontier (0,1).Y))
        pointFromBelow: = max ((0,0), (prevFrontier (1,0).X, prevFrontier (1,0).Y+1))
        pointFromItself: = max((0,0),( prevFrontier (0,0).X+1, prevFrontier (0,0).Y+1))
    else
        if i=0: //the diagonals above the main diagonal
            pointFromAbove:= max ((0,j), (prevFrontier (0,j+1).X+1, prevFrontier (0,j+1).Y))
            pointFromBelow:= max ((0,j), (prevFrontier (0,j-1).X, prevFrontier (0,j+1).Y+1))
           pointFromItself:= max((0,j),( prevFrontier (0,j).X+1, prevFrontier (0,j).Y+1))

        if j=0: //the diagonals below the main diagonal
            pointFromAbove:=max ((i,0), (prevFrontier (i-1,0).X+1, prevFrontier (i-1,0).Y))
            pointFromBelow:= max ((i,0), (prevFrontier (i+1,0).X, prevFrontier (i+1,0).Y+1))
            pointFromItself: = max((i,0),( prevFrontier (i,0).X+1, prevFrontier (i,0).Y+1))

    currEnd: = max (pointFromAbove, pointFromBelow, pointFromItself)
    currEnd: = snake (currEnd.X, currEnd.Y)
    if currEnd=(N,M):
        destinationReached: =true
    return currEnd
```

The MM algorithm – pseudocode IV

```
algorithm MM_Edit_Distance (S1, S2)
    destinationReached:=false
    d:=0
    initializeDiagonalArrays()
    snake(0,0)
    while destinationReached=false do
        d: =d+1
        buildExtensions (d)
    return d
```

```
algorithm snake ((x,y))
    while $x<N$ and $y<N$ and $S_1[x]=S_2[y]$ do:
        $x:=x+1$
        $y:=y+1$
    return (x,y)
```

Better Edit Distance computation. An open problem

- There are algorithms which perform better for the case of large edit distance.
- The complexity of all these algorithms is still quadratic in the worst case
- The best result (four-Russians speed-up) is $O(N^2/\log N)$
- Can it be done better?