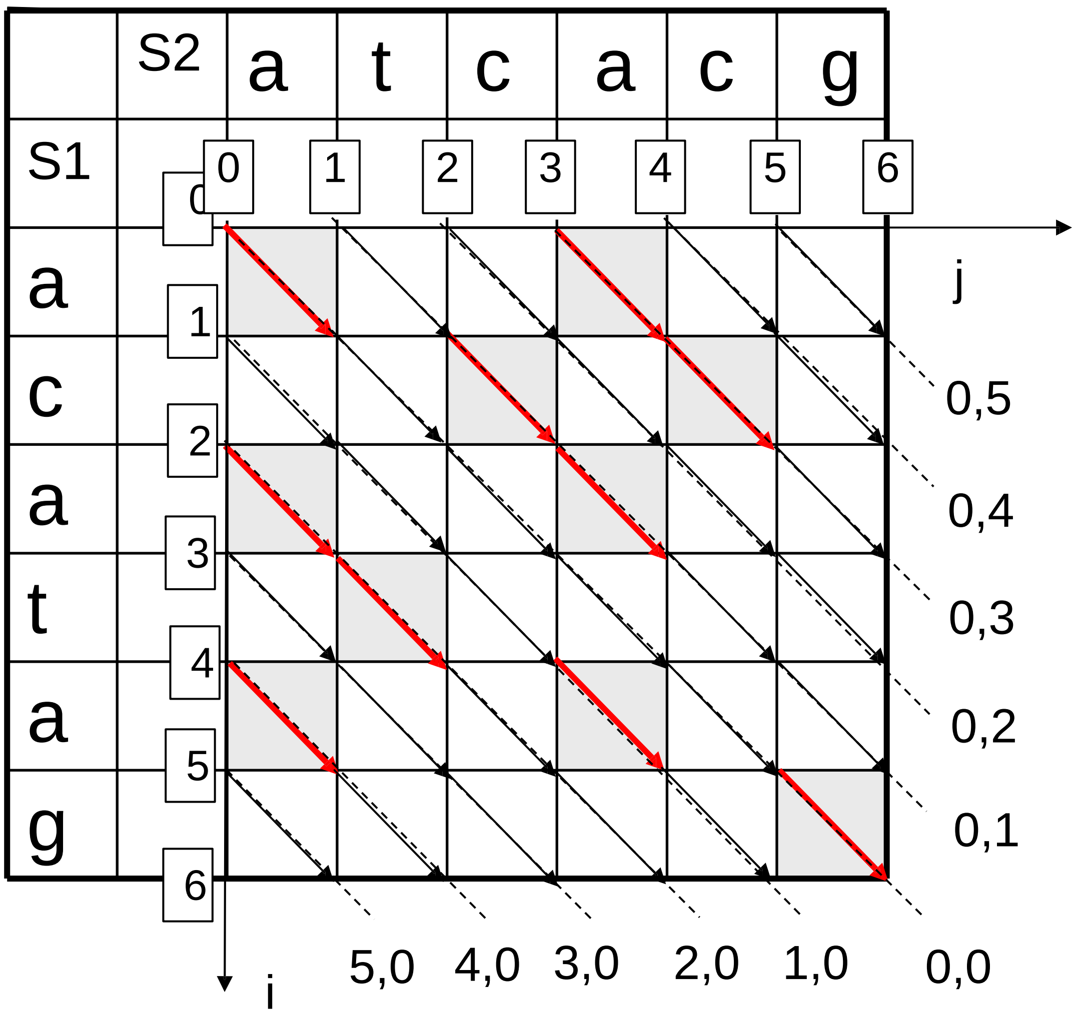
Edit Distance: improving running time

Lecture 05.04

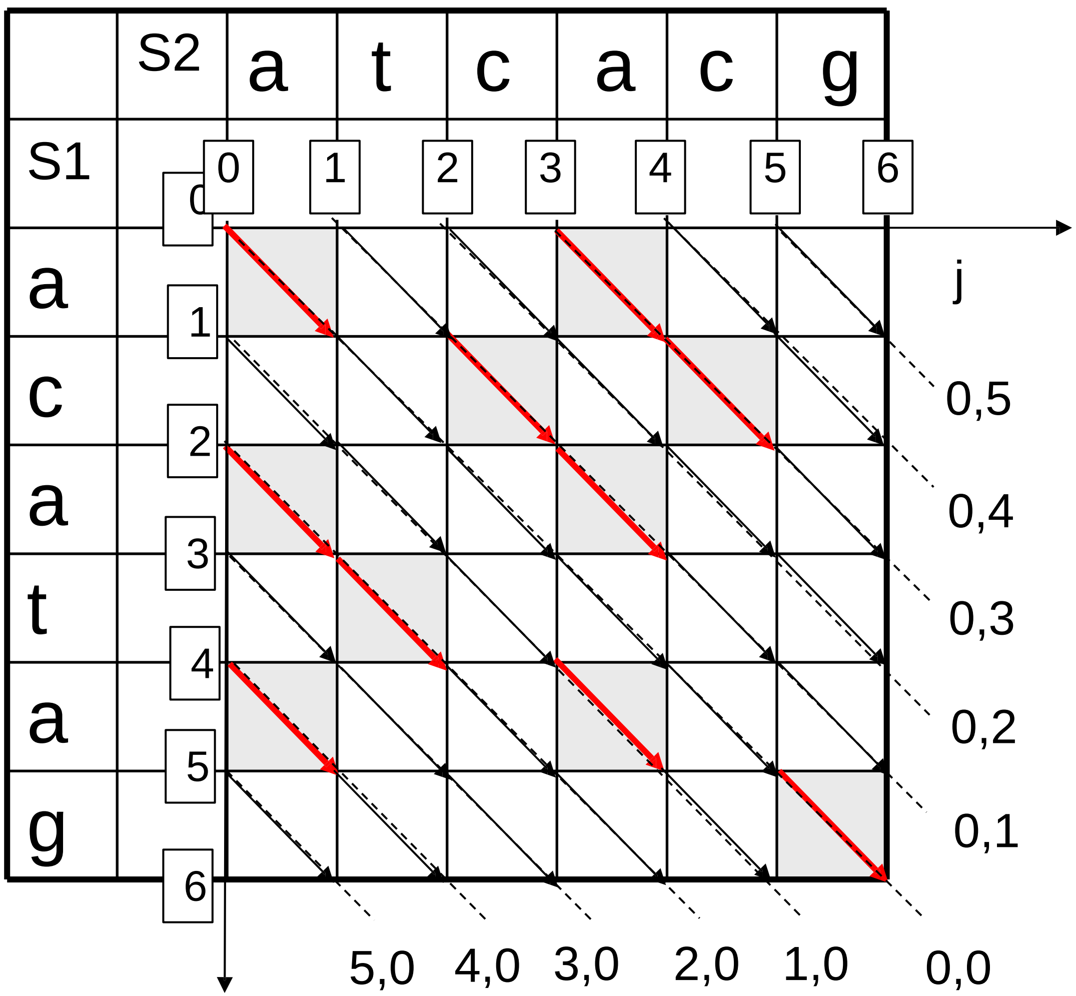
by Marina Barsky

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)

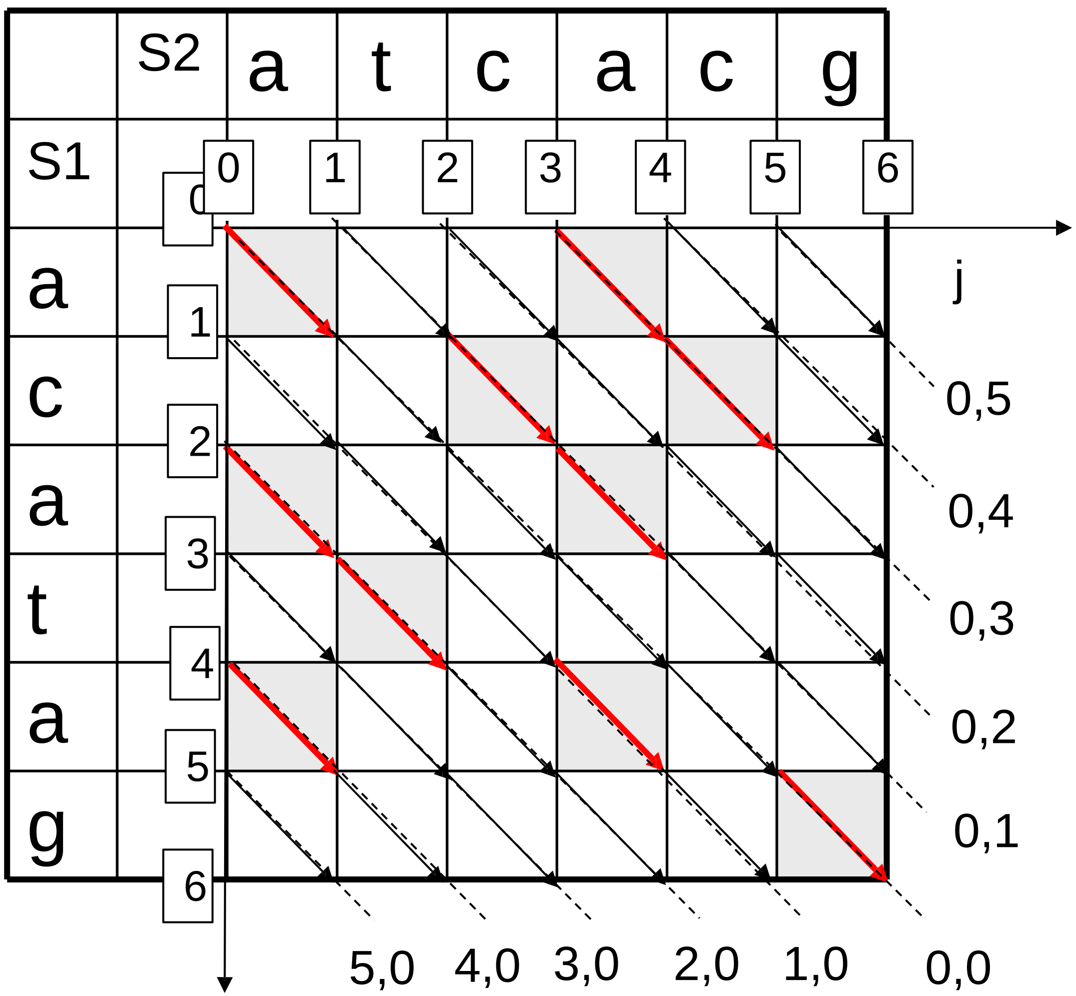
and diagonal (0,1)

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

diagonal (0,1)

and diagonal (0,3)

The MM algorithm: observation

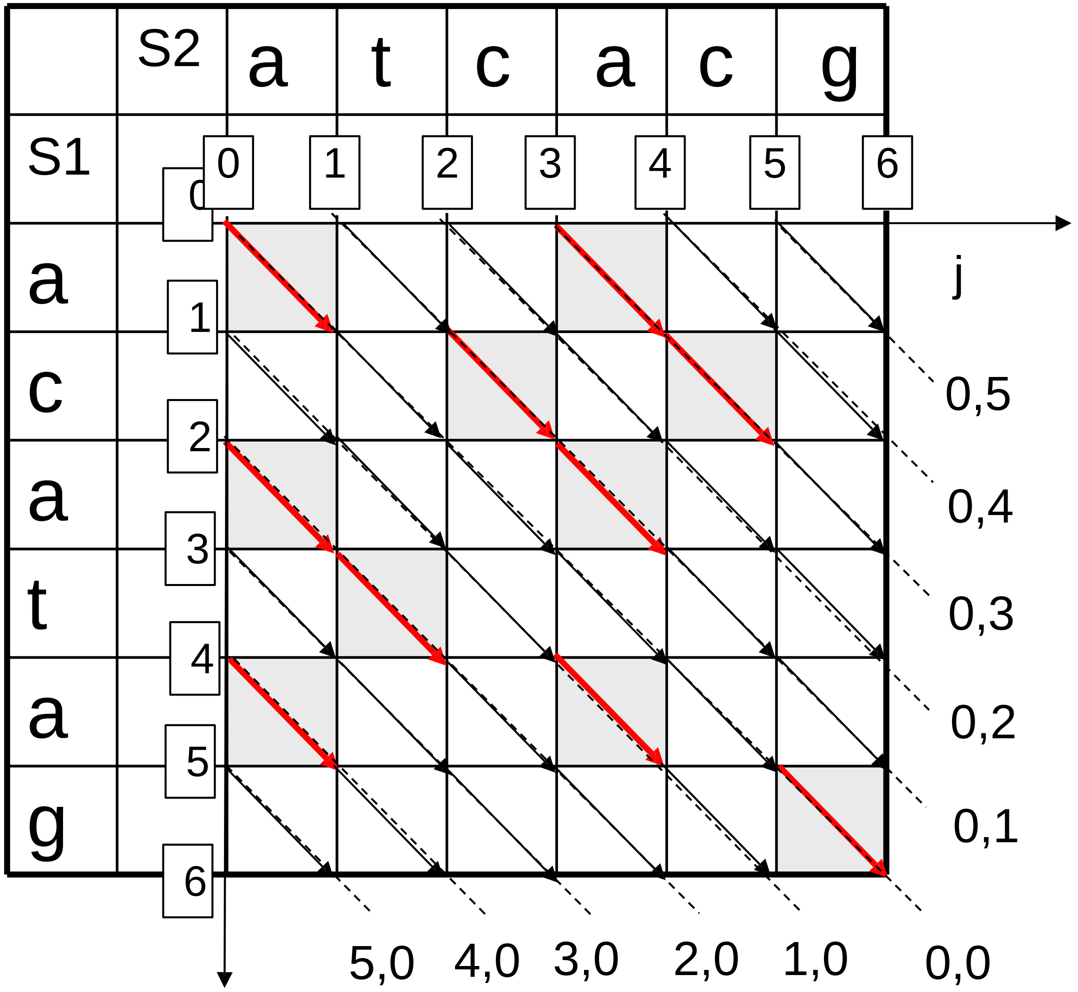


A ***d*-path** in the edit graph is a path which starts at point (0,0) and has a cost exactly *d*

Observation: 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)

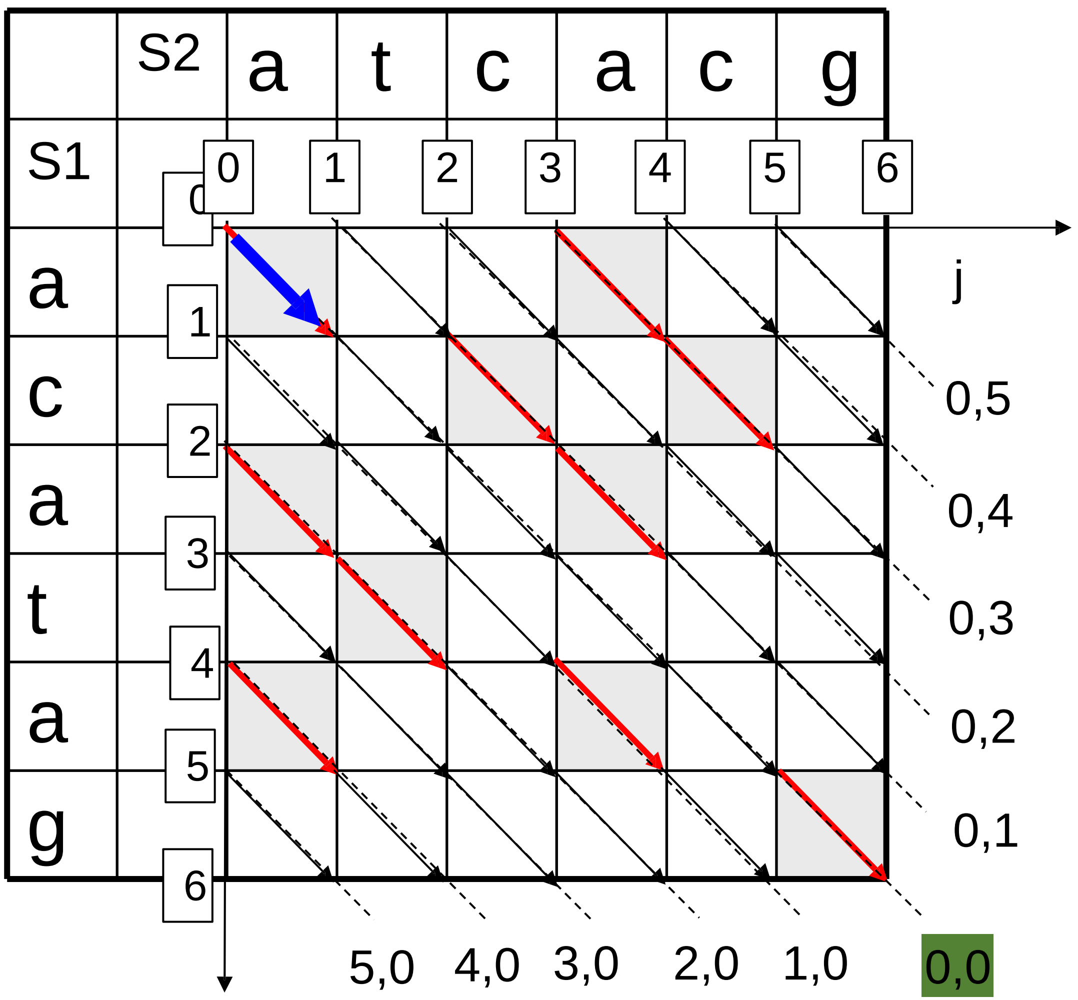
The MM algorithm



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

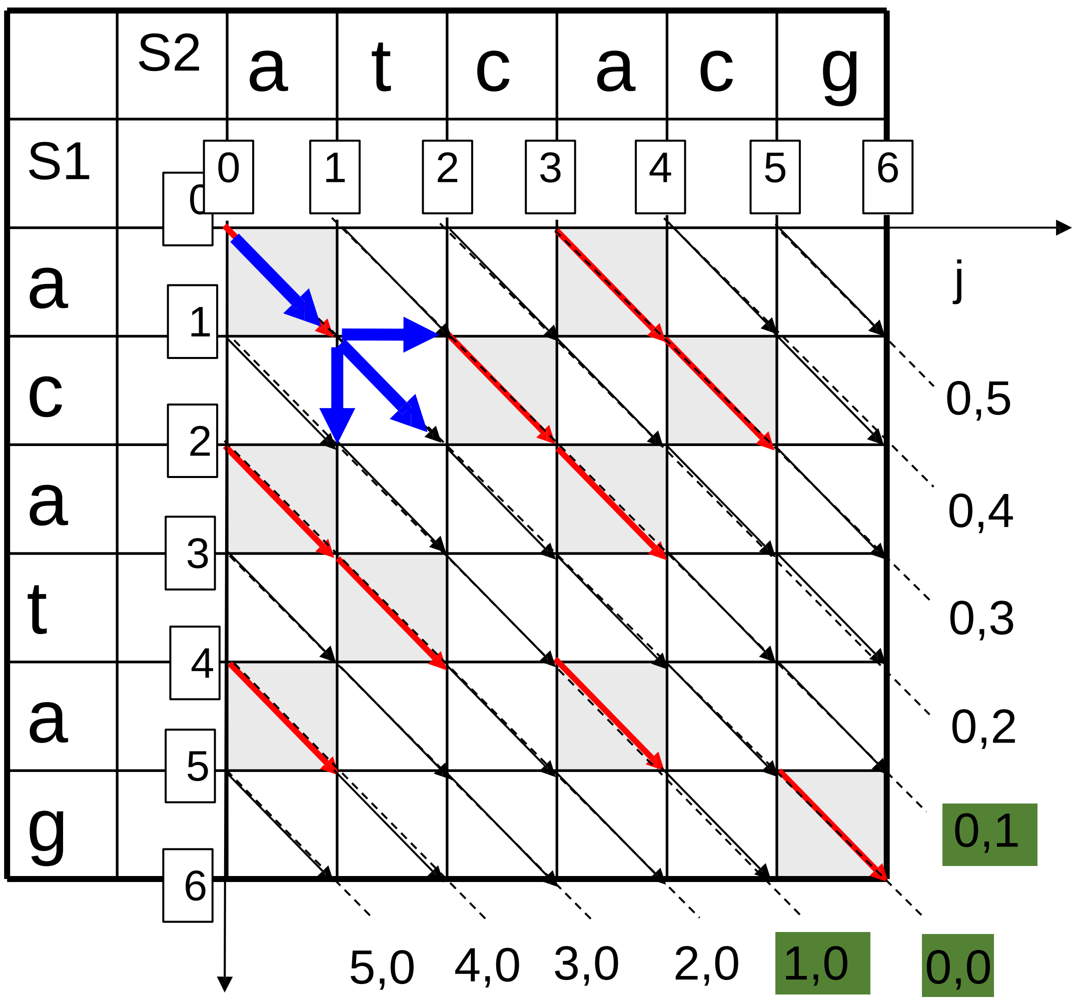
The MM algorithm. Iteration 0



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

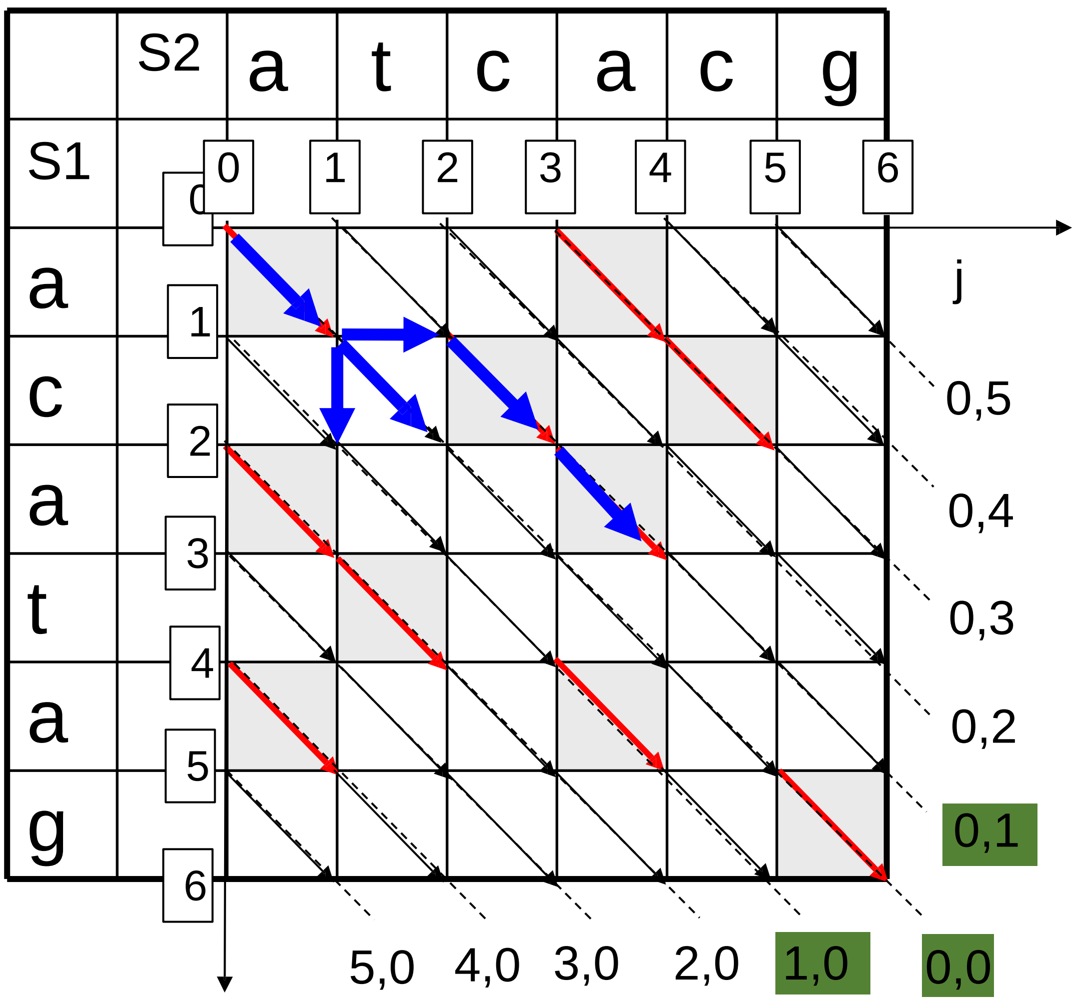


We produce all possible paths with a total cost 1.

There can be only 3 possible paths with the cost 1 and they end at:
the main diagonal (0,0)
Or one of 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

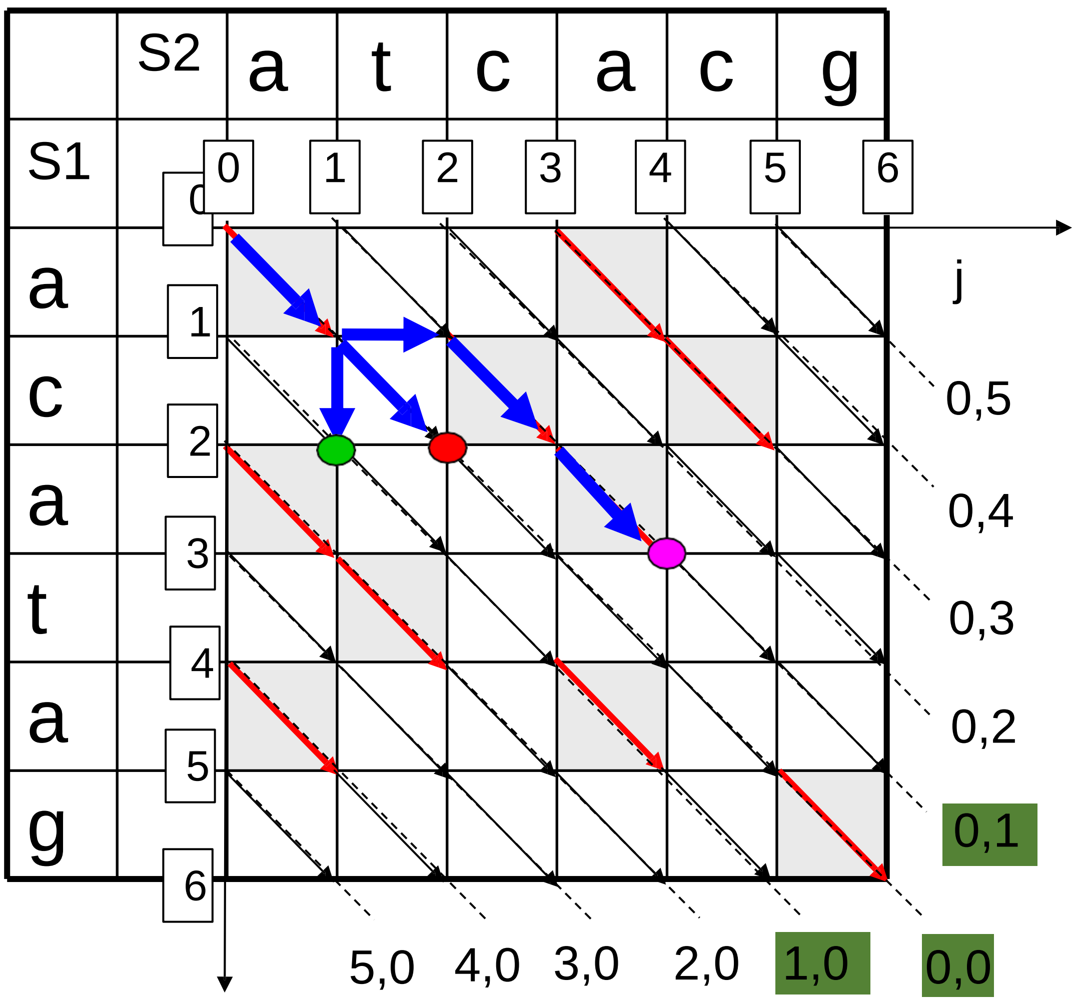
The MM algorithm. Iteration 1



We produced 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.

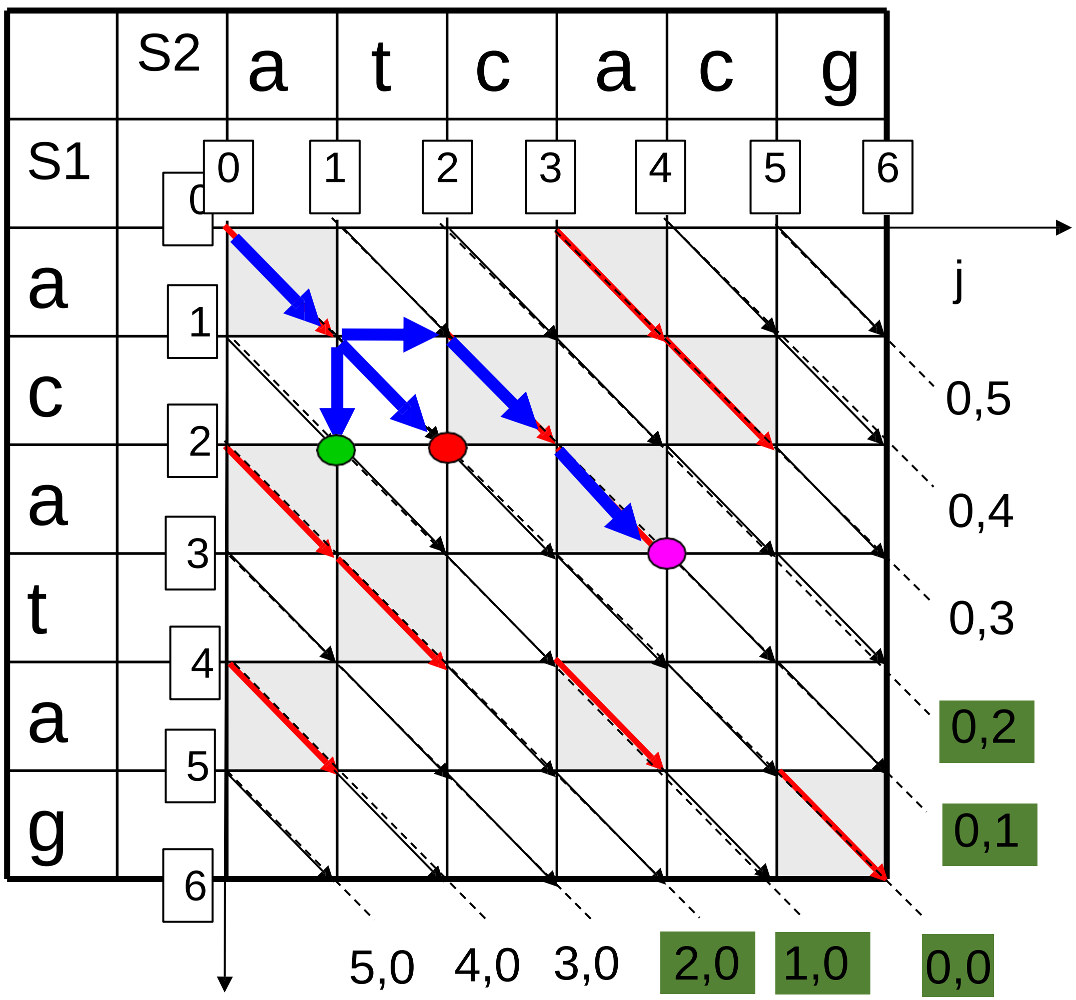
The MM algorithm. Iteration 1



We produced all possible paths with a total cost 1.

The ends of all paths of a total cost 1:

The MM algorithm. Iteration 2.

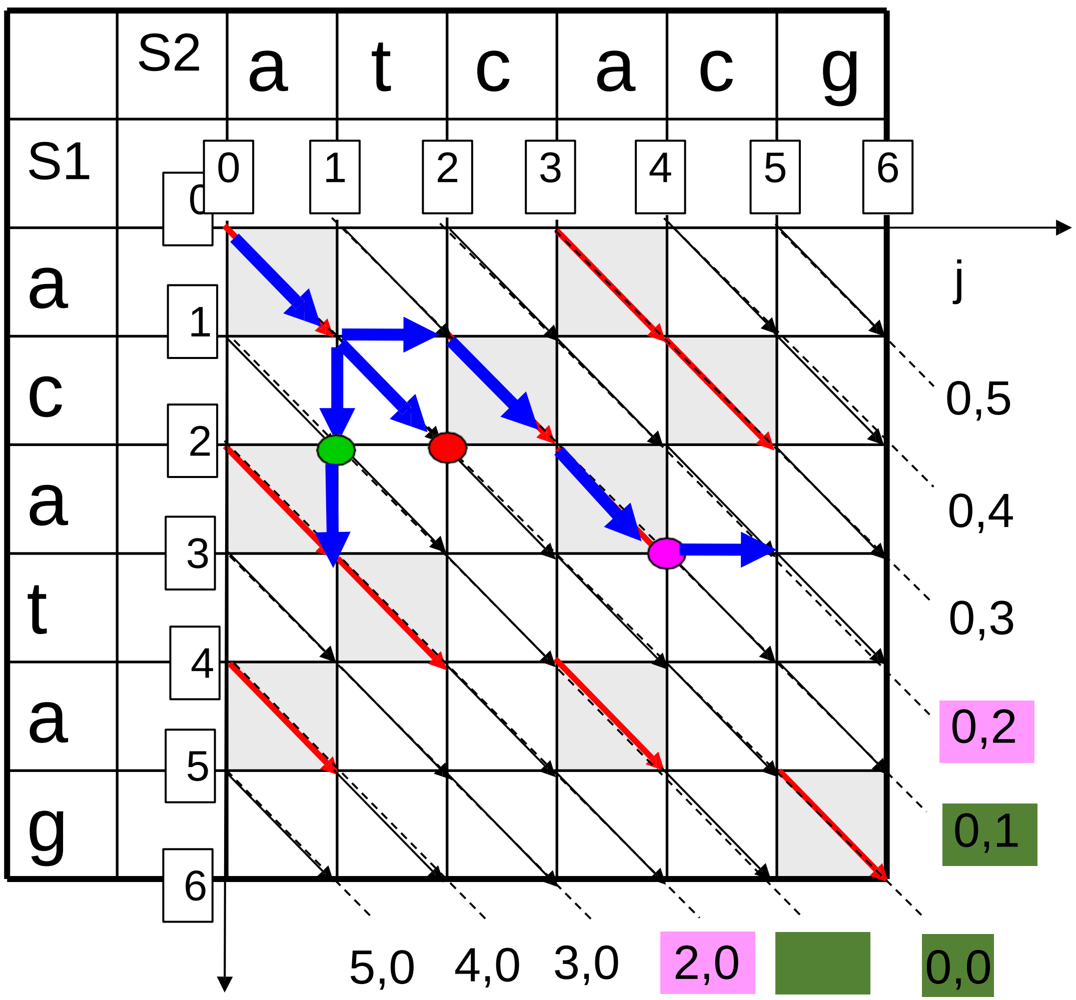


We produce all possible paths with a total cost 2.

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.

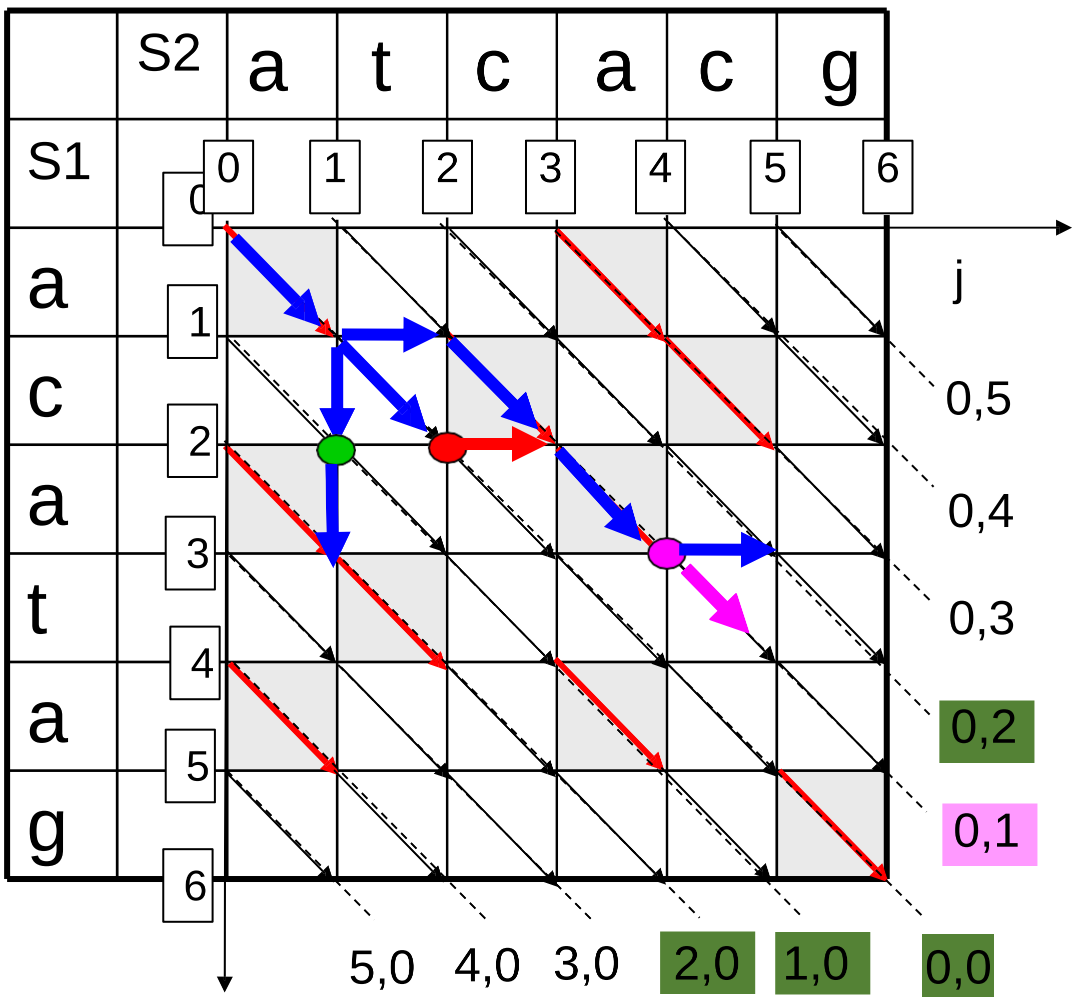


We produce all possible paths with a total cost 2.

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.



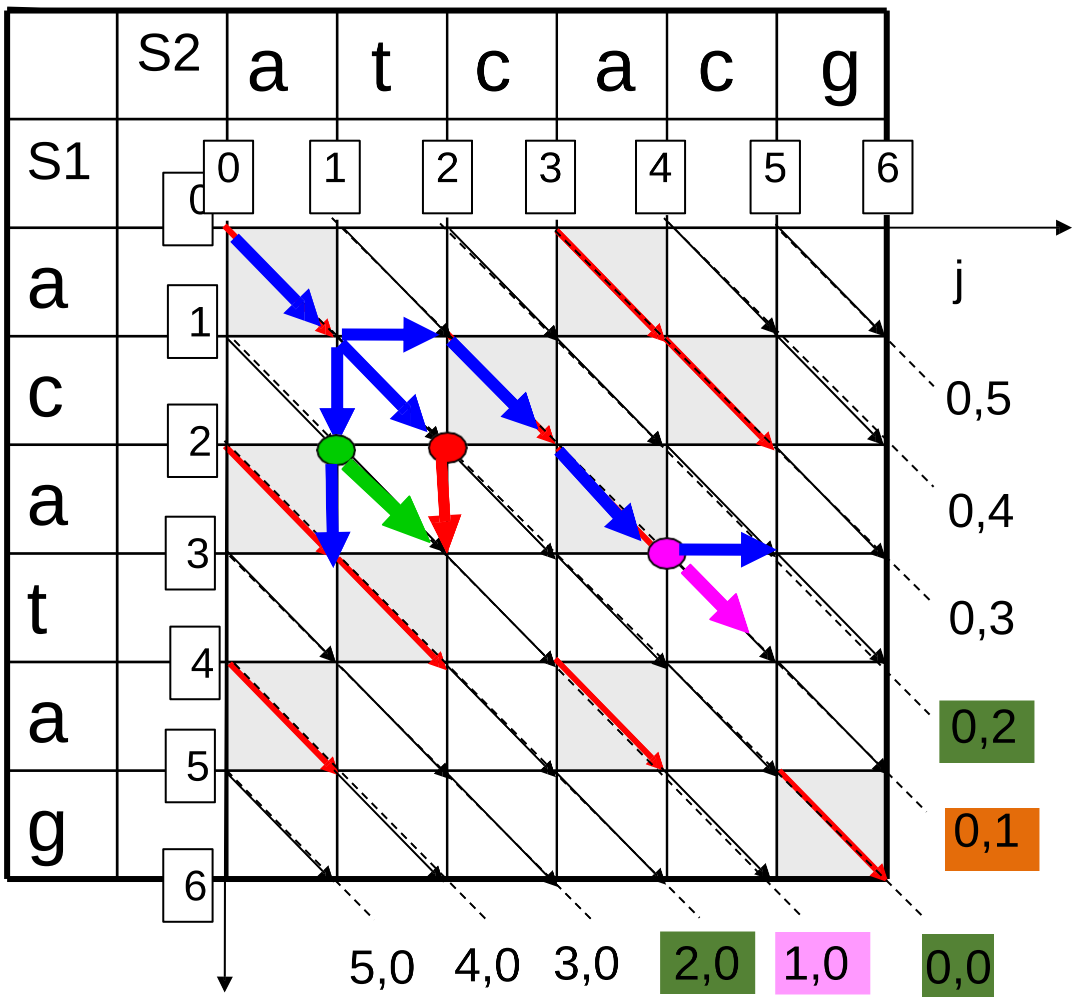
We produce all possible paths with a total cost 2.

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 the extension of a previous path which runs further along this diagonal:

The MM algorithm. Iteration 2. Dynamic programming

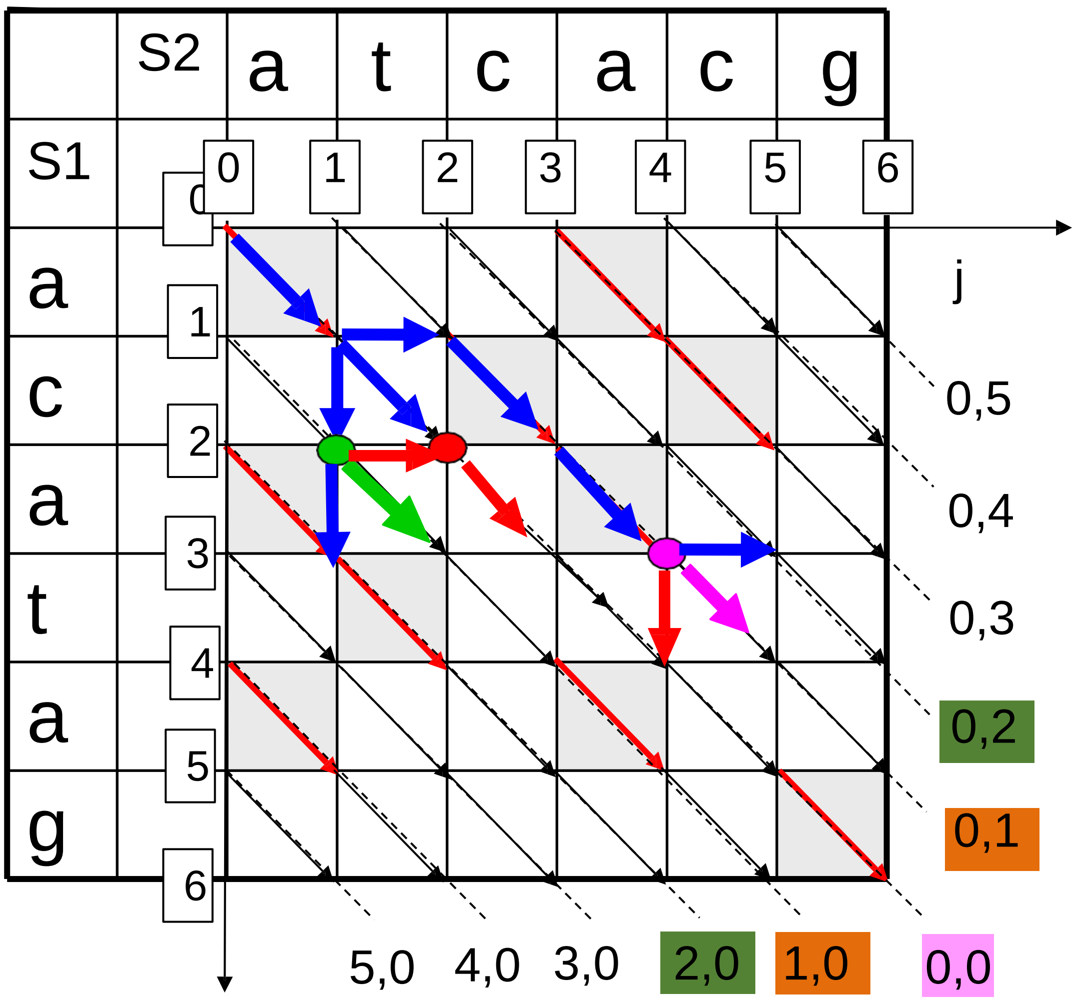


We produce all possible paths with a total cost 2.

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

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



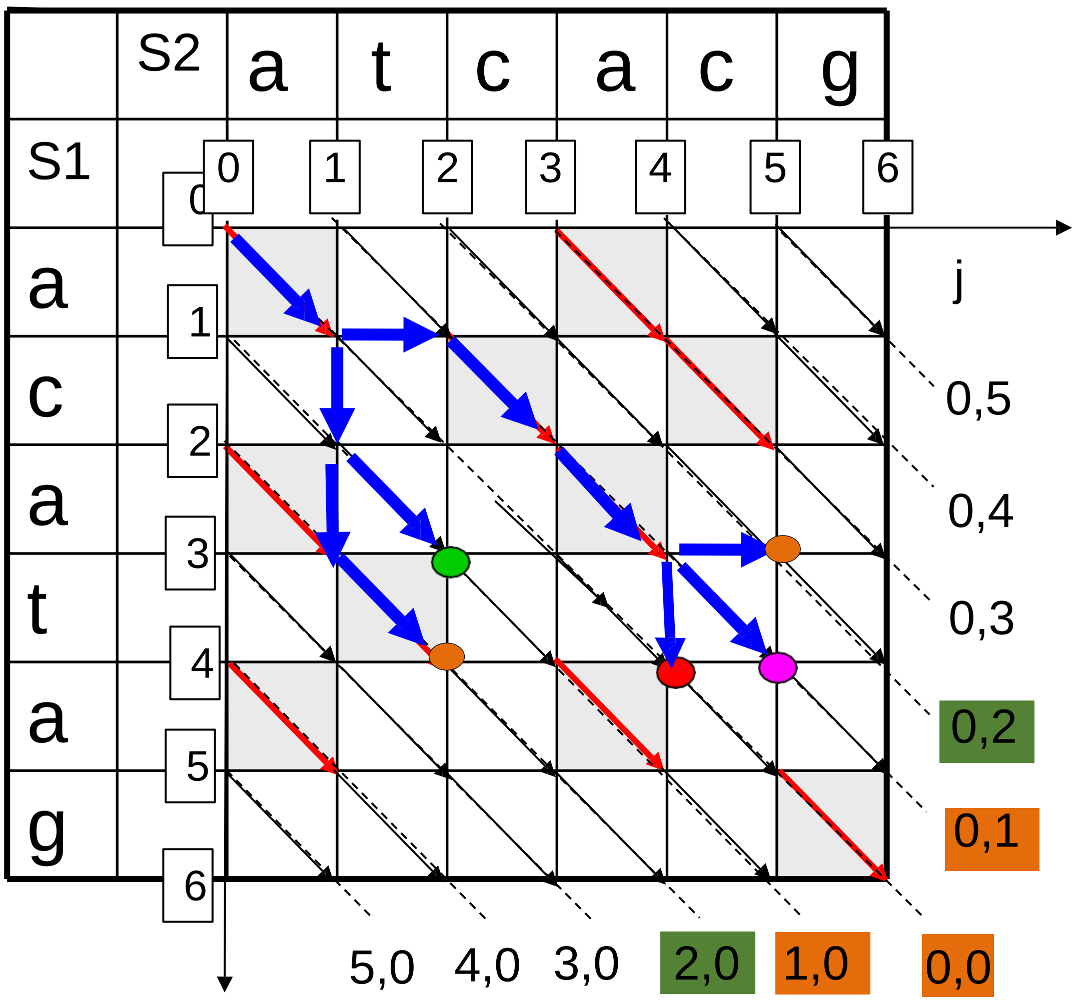
We produce all possible paths with a total cost 2.

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

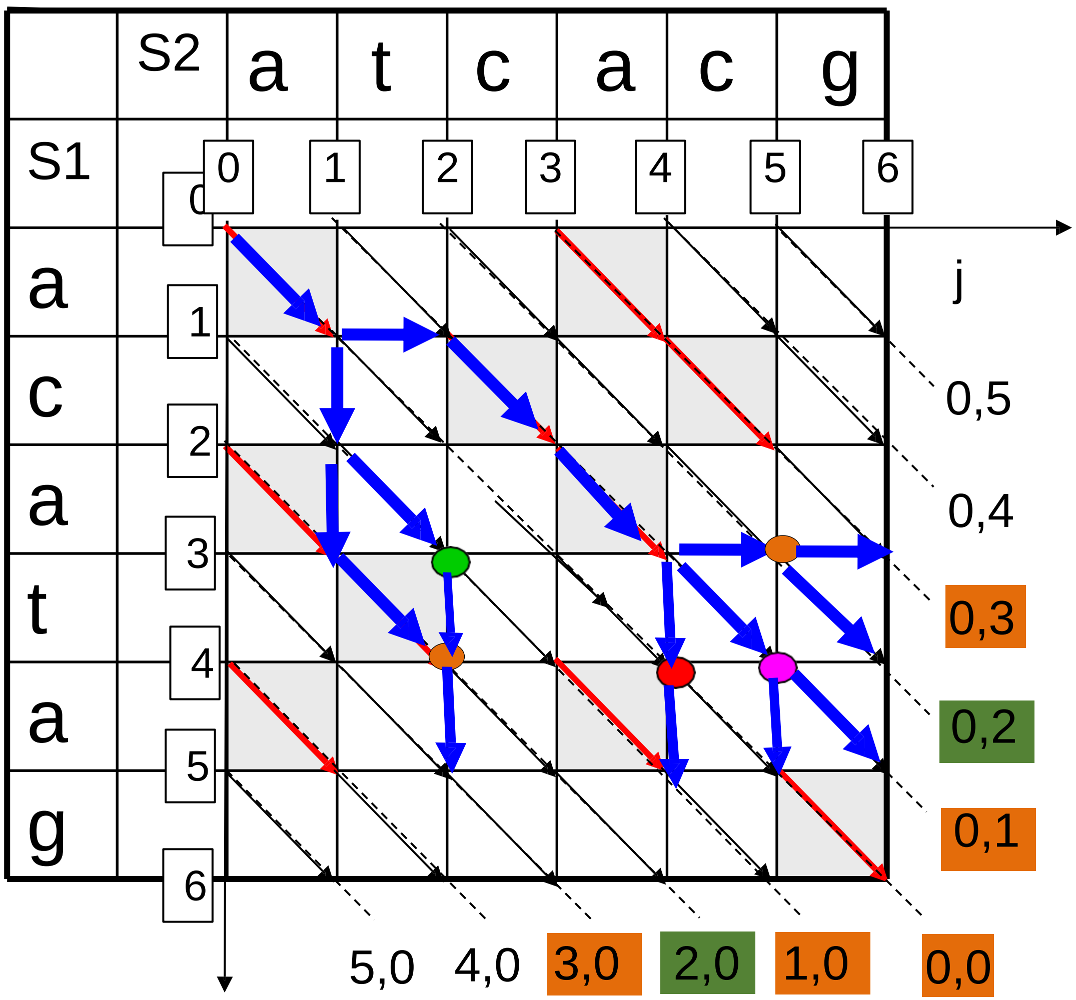


We produce all possible paths with a total cost 2.

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

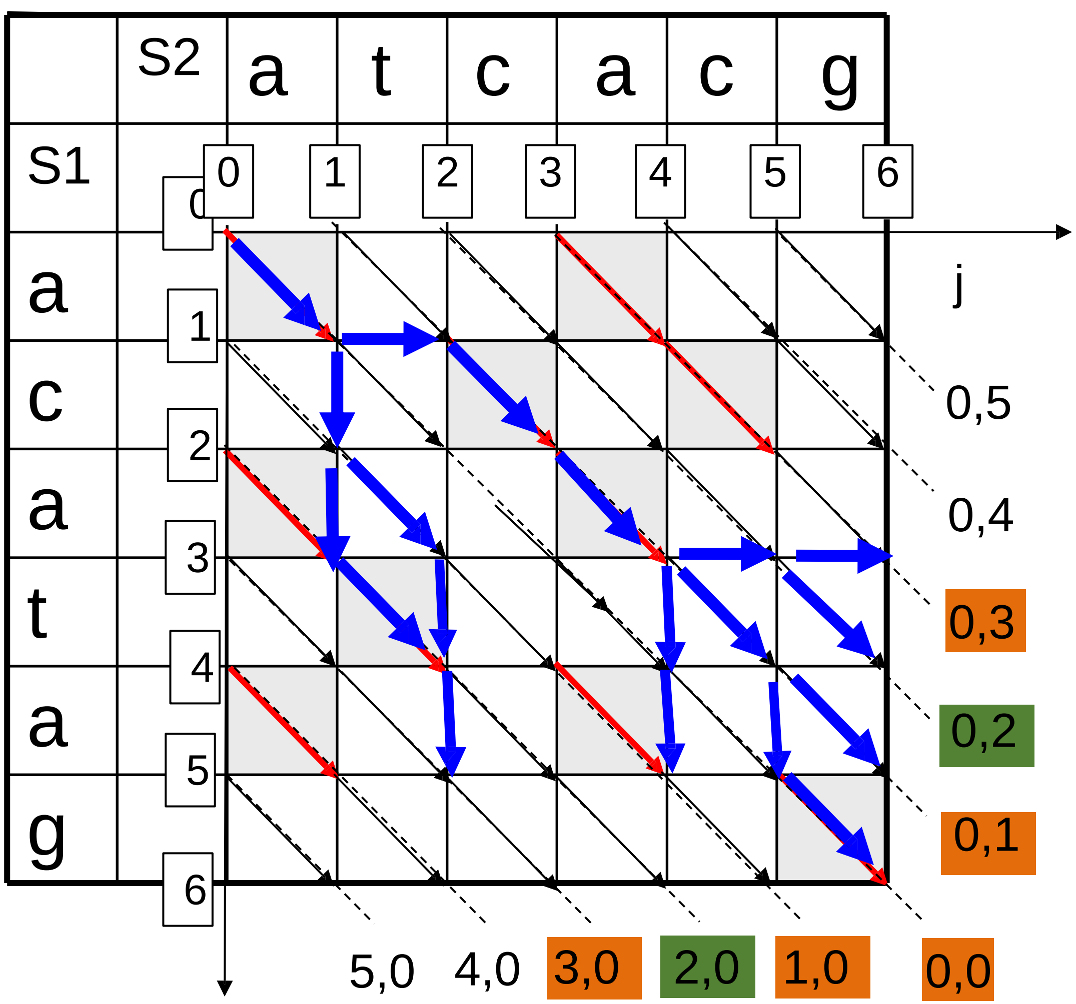


We produce all possible paths with a total cost 3.

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



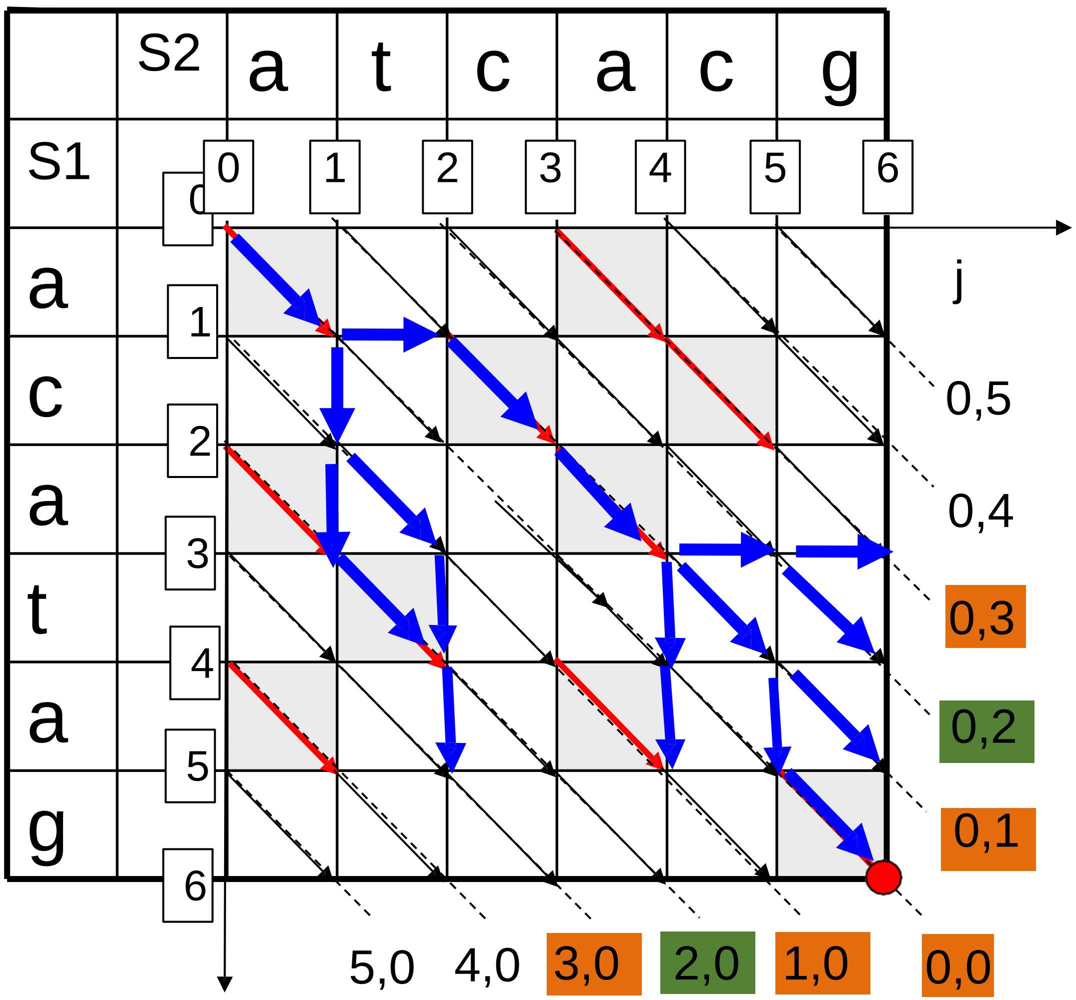
We produce all possible paths with a total cost 3.

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

And we extend each best path with the sequence of matches

The MM algorithm. Iteration 3. Reached destination

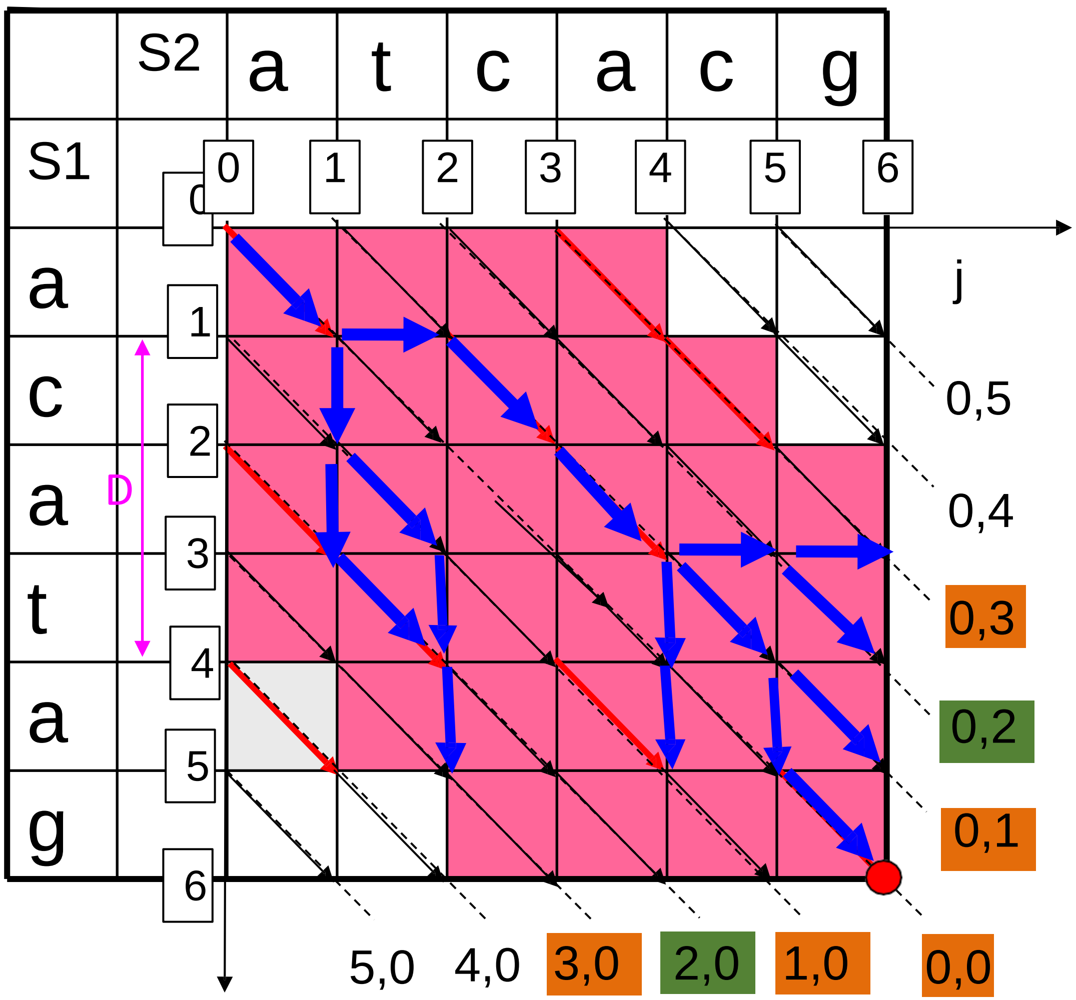


We produce all possible paths with a total cost 3.

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

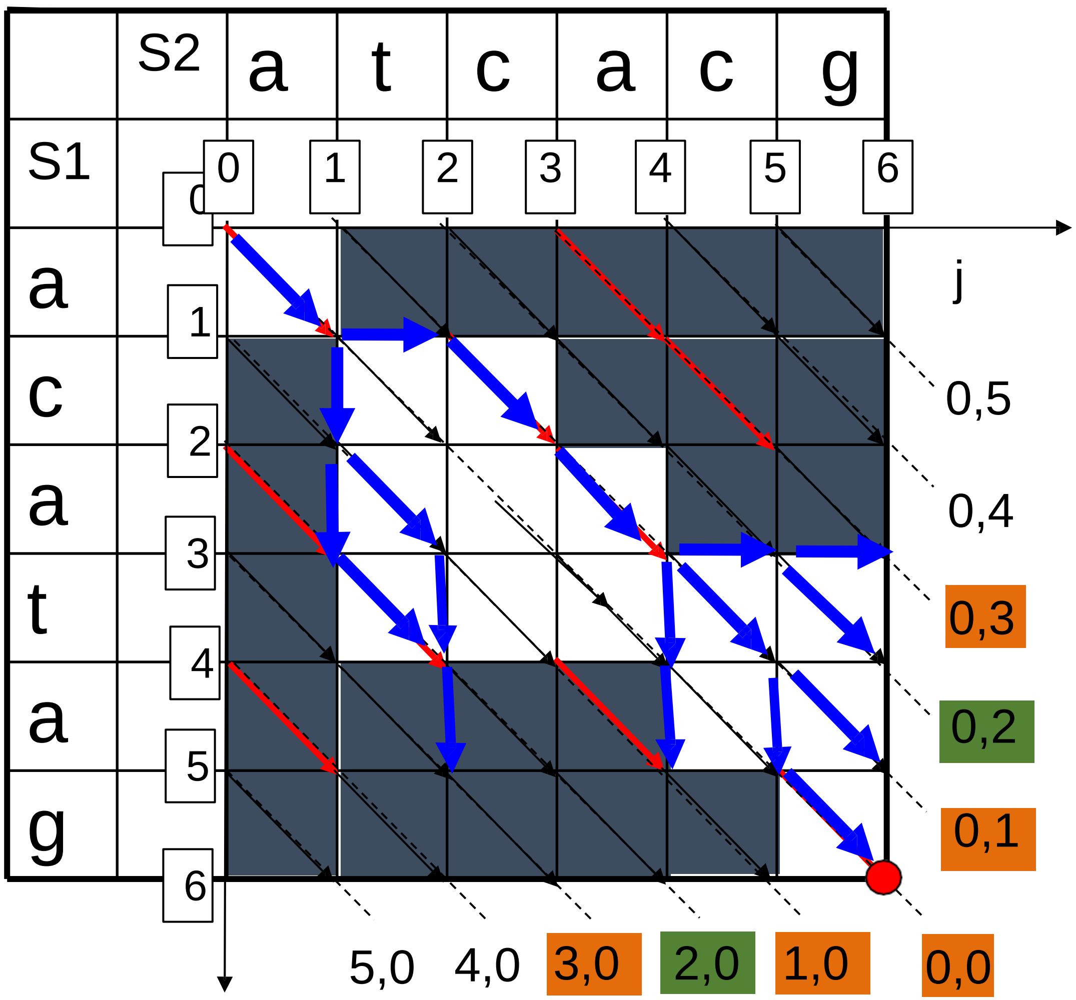
The algorithm terminates, and *D*=3.

The MM algorithm. Total work



If the final edit distance is *D*, we only compute the grid values in a strip 2D+1 around the main diagonal.

The MM algorithm. Total work



Note that we did not compute values of some cells at all (shown in grey)

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

The total running time is *O(ND)*

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

The MM algorithm – pseudocode 1/4

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 2/4

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

```
algorithm buildExtensions (I)
    for i from I 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 I 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 3/4

```
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 4/4

```
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 S1[x]=S2[y] do:
        x:=x+1
        y:=y+1
    return (x,y)
```

Faster Edit Distance: open problem

- There are also 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 – using Fast Fourier Transform) is O(N^2/log N)

Can it be done better?