Speeding up dynamic programming

The Four Russians
Unit Cost Edit distance

The unit cost edit distance of two strings $X[1..n]$ and $Y[1..m]$ can be computed in time and space $O(nm)$ by simple dynamic programming based on this recursion:

$$D(i, j) = \min \begin{cases} 
D(i-1, j-1) + (X[i] == Y[j] ? 0 : 1) & i > 0, j > 0 \\
D(i-1, j) + 1 & i > 0, j \geq 0 \\
D(i, j-1) + 1 & i \geq 0, j > 0 \\
0 & i = 0, j = 0 
\end{cases}$$
Unit Cost Edit distance

The **unit cost edit distance** of two strings \(X[1..n]\) and \(Y[1..m]\) can be computed in time and space \(O(nm)\) by simple dynamic programming based on this recursion:

\[
D(i, j) = \min \left\{ D(i-1, j-1) + (X[i] == Y[j] ? 0 : 1), \ D(i-1, j) + 1, \ D(i, j-1) + 1, \ 0 \right\}
\]

\[ i > 0, j > 0 \]
\[ i > 0, j \geq 0 \]
\[ i \geq 0, j > 0 \]
\[ i = 0, j = 0 \]

Can we compute the edit distance faster than \(O(nm)\)?

Yes. In time \(O(n^2 / \log n)\) for \(n \geq m\) using the “Four Russians” technique introduced in 1970 for boolean matrix multiplication and adapted by Masek and Paterson in 1980 to unit cost edit distance.
The Four Russians Technique

The basic idea is to precompute parts of the computation involved in filling out the dynamic programming table.

A \textbf{t-block} (here $t=3$) at position $(3,3)$ in the table

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>c</th>
<th>g</th>
<th>t</th>
<th>a</th>
<th>g</th>
<th>c</th>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
The Four Russians Technique

The basic idea is to precompute parts of the computation involved in filling out the dynamic programming table.

A t-block (here t=3) at position (3,3) in the table

Consider a t-block at position (i,j)

The output F is a function of the inputs A, B, C and the substrings X[i+1 .. i+t] and Y[j+1 .. j+t], and it can be computed in time O(t^2) by the basic recursion.

\[ b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) = F \]
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)
The basic idea

Assume the **block-function** \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note:** We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

![Table](image_url)
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note:** We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

```
 0 1 2 3 4 5 6 7 8 9 10 11 12
 a 1 0 1 2 3 4 5 6 7 8 9 10 11
 c 2 1 0 1 2 3 4 5 6 7 8 9 10
 g 3 2 1 0 1 2 3 4 5 6 7 8 9
 t 4 3 2 1 0 1 2 3 4 5 6 7 8
 g 5 4 3 2 1 1 1 2 3 4 5 6 7
 t 6 5 4 3 2 2 2 1 2 3 4 5 6
 c 7 6 5 4 3 2 2 2 3 3 4 5 4
 a 8 7 6 5 4 3 3 3 2 3 4 4 4
 g 9 8 7 6 5 4 4 4 3 3 4 5 4
 t 10 9 8 7 6 5 5 5 4 4 3 4 5
 a 11 10 9 8 7 6 5 6 5 4 4 4 5
 c 12 11 10 9 8 7 6 5 6 5 5 4 5
```
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 \ldots i+t], Y[j+1 \ldots j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note:** We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>g</th>
<th>t</th>
<th>c</th>
<th>g</th>
<th>t</th>
<th>a</th>
<th>c</th>
<th>g</th>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X_{i+1 \ldots i+t}, Y_{j+1 \ldots j+t}) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 \ldots i+t], Y[j+1 \ldots j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \(b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t])\) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \(D[n,m]\)

Note: We (of course) do not allocate the entire D-table, since this would take time \(O(n^2)\) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function $b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t])$ has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return $D[n,m]$ 

Note: We (of course) do not allocate the entire D-table, since this would take time $O(n^2)$ by itself. We allocate a “row of blocks”.

![Diagram showing a matrix with rows and columns labeled with letters and numbers, indicating the process of initializing and filling the D-table.](image)
The basic idea

Assume the block-function $b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t])$ has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return $D[n,m]$

Note: We (of course) do not allocate the entire D-table, since this would take time $O(n^2)$ by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1..i+t], Y[j+1..j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note:** We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

```plaintext
\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
2 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
3 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
4 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
5 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
6 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
7 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
8 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
9 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
10 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
11 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
12 & 10 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 \ldots i+t], Y[j+1 \ldots j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note**: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 \ldots i+t], Y[j+1 \ldots j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

**Note:** We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function $b(A, B, C, X[i+1 \ldots i+t], Y[j+1 \ldots j+t])$ has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return $D[n,m]$

Note: We (of course) do not allocate the entire D-table, since this would take time $O(n^2)$ by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Note: We (of course) do not allocate the entire D-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Running time?
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Running time?

Step 1 takes time \( O(n) \), step 3 takes time \( O(1) \).

Step 2 takes time \( O(n^2 / t^2) \) x “time it takes to get the output of the block-function”. Since the input and output is of size \( O(t) \), step 2 takes time \( O(n^2 / t) \).
The basic idea

Assume the block-function \( b(A, B, C, X[i+1 .. i+t], Y[j+1 .. j+t]) \) has been precomputed for all possible inputs.

1) Initialize first row and column in the D-table.

2) Fill the table row-by-row using the block-function.

3) Return \( D[n,m] \)

Running time?

Step 1 takes time \( O(n) \), step 3 takes time \( O(1) \).

Step 2 takes time \( O(n^2 / t^2) \times \text{“time it takes to get the output of the block-function”} \). Since the input and output is of size \( O(t) \), step 2 takes time \( O(n^2 / t) \).

Total running time: \( O(n^2 / t) + \text{“time it takes to precompute b”} \)
Precomputing the block-function

Computing $b(A, B, C, x, y)$, $x=\text{X}[i+1 \ldots i+t]$ and $y=\text{Y}[j+1 \ldots j+t]$), takes time $O(t^2)$ for each input. What is the number of inputs?

Observe that $0 \leq D[i,j] \leq n$, i.e.

1) Number of A's: $n+1$
2) Number of B's: $(n+1)^t$
3) Number of C's: $(n+1)^t$

If the input strings $X$ and $Y$ are from $\Sigma^*$, then

4) Number of x's: $|\Sigma|^t$
5) Number of y's: $|\Sigma|^t$

Total number of input: $(n+1)^{2t+1}|\Sigma|^{2t}$
Reducing the number of inputs

The number of inputs to the block-function can be reduced by two observations.

Observation 1

Neighboring cells in the dynamic programming table differ by at most 1, i.e. inputs B and C can be encoded as offsets (-1, 0, +1) from A.

Fx A=3, B=(4,5,6) and C=(2,1,2) becomes A=3, B'= (+1, +1, +1) and C'= (-1, -1, +1).
Proof of “observation 1”

We will show that $D(i, j-1)-1 \leq D(i, j) \leq D(i, j-1)+1$, i.e. that neighbors in row $i$ differ by at most 1. The same can be done for columns.

Part 1:
Upper bound $D(i,j) \leq D(i, j-1)+1$ follows immediately from the basic recursion.

Part 2:
Lower bound $D(i, j-1) -1 \leq D(i,j)$ follows from looking at the optimal path from $(0,0) \rightarrow (i,j)$ with cost $D(i,j)$. There are two cases:

Case 1:
Path ends with a horizontal edge, i.e. $(0,0) \rightarrow (i,j-1) \rightarrow (i,j)$, then:

$$D(i,j) = D(i, j-1)+1 \implies D(i,j-1)-1 \leq D(i,j).$$

Case 2:
Path ends with a diagonal or vertical edge, i.e. $(0,0) \rightarrow (i-k,j-1) \rightarrow (i-k+1,j) \rightarrow (i,j)$ for some $k \geq 1$ (if $k=1$ then path ends in a diagonal edge, if $k > 1$, then path ends in $k-1$ vertical edges).

Since the cost of the edge $(i-k,j-1) \rightarrow (i-k+1,j)$ is 0 or 1, we have $D(i,j) \geq D(i,k,j-1) + k - 1$, and since $D(i,j-1) \leq D(i-k,j-1)+k$, we have $D(i,j) \geq D(i,j-1) - 1$. 
Proof of “observation 1”

We will show that $D(i, j-1)-1 \leq D(i, j) \leq D(i, j-1)+1$, i.e. that neighbors in row $i$ differ by at most 1. The same can be done for columns.

**Part 1:**
Upper bound $D(i,j) \leq D(i, j-1)+1$ follows immediately from the basic recursion.

**Part 2:**
Lower bound $D(i, j-1) -1 \leq D(i,j)$ follows from looking at the optimal path from $(0,0) \rightarrow (i,j)$ with cost $D(i,j)$. There are two cases:

**Case 1:**
Path ends with a horizontal edge, i.e. $(0,0) \rightarrow (i,j-1) \rightarrow (i,j)$, then:

$D(i,j) = D(i, j-1)+1 \Rightarrow D(i,j-1)-1 \leq D(i,j)$.

**Case 2:**
Path ends with a diagonal or vertical edge, i.e. $(0,0) \rightarrow (i-k,j-1) \rightarrow (i-k+1,j) \rightarrow (i,j)$ for some $k \geq 1$ (if $k=1$ then path ends in a diagonal edge, if $k > 1$, then path ends in $k-1$ vertical edges).

Since the cost of the edge $(i-k,j-1) \rightarrow (i-k+1,j)$ is 0 or 1, we have $D(i,j) \geq D(i-k,j-1) + k - 1$, and since $D(i,j-1) \leq D(i-k,j-1)+k$, we have $D(i,j) \geq D(i,j-1) - 1$. 
Reducing the number of inputs

The number of inputs to the block-function can be reduced by two observations.

**Observation 1**

Neighboring cells in the dynamic programming table differ by at most 1, i.e. inputs B and C can be encoded as offsets (-1, 0, +1) from A.

\[ \text{Fx } A=3, B=(4,5,6) \text{ and } C=(2,1,2) \text{ becomes } A=3, B'= (+1, +1, +1) \text{ and } C'= (-1, -1, +1). \]

**Observation 2**

Let \( b' \) be the offset-encoded block-function, i.e. inputs B, C, and output F, are encoded as offsets, then \( b'(A,B',C',x,y) = b'(0,B',C',x,y) \), i.e. no need to precompute for all values of A.
Reducing the number of inputs

The number of inputs to the block-function can be reduced by two observations.

**Observation 1**

Neighboring cells in the dynamic programming table differ by at most 1, i.e. inputs B and C can be encoded as offsets (-1, 0, +1) from A. 

Fx A=3, B=(4,5,6) and C=(2,1,2) becomes A=3, B'=(+1, +1, +1) and C'=(-1, -1, +1).

**Observation 2**

Let b' be the offset-encoded block-function, i.e. inputs B, C, and output F, are encoded as offsets, then b'(A,B',C',x,y) = b'(0,B',C',x,y), i.e. no need to precompute for all values of A.
Reducing the number of inputs

The number of inputs to the block-function can be reduced by two observations.

**Observation 1**

Neighboring cells in the dynamic programming table differ by at most 1, i.e. inputs B and C can be encoded as offsets (-1, 0, +1) from A.

Fx A=3, B=(4, 5, 6) and C=(2, 1, 2) becomes A=3, B'=(+1, +1, +1) and C'=(-1, -1, +1).

**Observation 2**

Let \( b'(B',C',x,y) \) be the offset-encoded block-function, i.e. inputs B, C, and output F, are encoded as offsets, then \( b'(A,B',C',x,y) = b'(0,B',C',x,y) \), i.e. no need to precompute for all values of A.

The offset-encoded block-function \( b'(B',C',x,y) \) can be computed in time \( O(t^2) \) by filling out the first row/column cf. the offsets in the inputs B' and C', running the basic algorithm, and converting the last row/column to offsets in the output F'.

Number of inputs becomes: \( (\text{Number of B's}) \times (\text{Number of C's}) \times (\text{Number of x and y's}) = 3^t \Sigma^t \times 2^t \Sigma^t = (3 \Sigma)^{2t} \), i.e. independent of \( n \).
The algorithm

Assume that the offset coded block-function \( b'(B', C', x, y) \) has been precomputed for all possible inputs.

1) Allocate offset-table \( T[0..n][0..m] \), and initialize first row and column with +1 offsets.

2) Fill the offset-table row-by-row using the offset-encoded block-function.

3) Return \( n + \text{“sum of last row of offsets”} = n + T[n,0] + T[n,1] + .. + T[n,m] \)
The algorithm

Assume that the offset coded block-function $b'(B', C', x, y)$ has been precomputed for all possible inputs.

1) Allocate offset-table $T[0..n][0..m]$, and initialize first row and column with +1 offsets.

2) Fill the offset-table row-by-row using the offset-encoded block-function.

3) Return $n + \text{"sum of last row of offsets"} = n + T[n,0] + T[n,1] + .. + T[n,m]$

Note: We (of course) do not allocate the entire $T$-table, since this would take time $O(n^2)$ by itself. We allocate a "row of blocks".
### The algorithm

Assume that the offset coded block-function \( b'(B', C', x, y) \) has been precomputed for all possible inputs.

1. Allocate offset-table \( T[0..n][0..m] \), and initialize first row and column with +1 offsets.

2. Fill the offset-table row-by-row using the offset-encoded block-function.

3. Return \( n + \text{“sum of last row of offsets”} = n + T[n,0] + T[n,1] + \ldots + T[n,m] \)

**Note:** We (of course) do not allocate the entire T-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+1</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>+1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>+1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>+1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>+1</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>+1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+1</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+1</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>+1</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>+1</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>+1</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
</tr>
</tbody>
</table>

Note: We do not allocate the entire T-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

The offset-coded block-function is used to find the offsets in a table. The table is initialized with all +1 offsets. Then, the table is filled row-by-row using the offset-coded block-function. Finally, the sum of the last row of offsets is returned.
The algorithm

Assume that the offset coded block-function \( b'(B', C', x, y) \) has been precomputed for all possible inputs.

1) Allocate offset-table \( T[0..n][0..m] \), and initialize first row and column with +1 offsets.

2) Fill the offset-table row-by-row using the offset-encoded block-function.

3) Return \( n + \text{“sum of last row of offsets”} = n + T[n,0] + T[n,1] + .. + T[n,m] \)

Note: We (of course) do not allocate the entire T-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.
The algorithm

Assume that the offset coded block-function \( b'(B', C', x, y) \) has been precomputed for all possible inputs.

1) Allocate offset-table \( T[0..n][0..m] \), and initialize first row and column with +1 offsets.

2) Fill the offset-table row-by-row using the offset-encoded block-function.

3) Return \( n + \text{“sum of last row of offsets”} = n + T[n,0] + T[n,1] + .. + T[n,m] \)

**Note:** We (of course) do not allocate the entire \( T \)-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

Total running time: \( O(n^2 / t) + \text{“time it takes to precompute b’”} \)
### How to pick the block-size \( t \) to improve total running time

Recall that the number of inputs to \( b' \) is \((3|\Sigma|)^t\).

Pick \( t = (\log_{3|\Sigma|} n) / 2 \), then the time it takes to compute the block-function for all inputs becomes:

\[
(3|\Sigma|)^{\log_{3|\Sigma|} n} \cdot t^2 = n \left(\frac{\log_{3|\Sigma|} n}{2}\right)^2 = O\left(n \log^2 n\right)
\]

and the total running time becomes

\[
O\left(\frac{n^2}{t} + n \log^2 n\right) = O\left(\frac{n^2}{\log n}\right)
\]

i.e. a speed-up!

1. Allocate offset-table \( T[0..n][0..m] \), and initialize first row and column with +1 offsets.
2. Fill the offset-table row-by-row using the offset-encoded block-function.
3. Return \( n + \) “sum of last row of offsets” = \( n + T[n,0] + T[n,1] + \ldots + T[n,m] \)

**Note:** We (of course) do not allocate the entire \( T \)-table, since this would take time \( O(n^2) \) by itself. We allocate a “row of blocks”.

**Total running time:** \( O(n^2 / t) + \) “time it takes to precompute \( b' \) ”
Does it work in practice?

How to pick block size $t = (\log_{3|\Sigma|} n) / 2$ in practice.

Fx $n=1000$, $|\Sigma| = 4$, then $(\log_{12} 1000) / 2 = (\log 1000 / \log 12) / 2 = 1.39$, i.e. we should pick $t=2$. Since $(\log_{12} n) / 2 < 2$ iff $n < 20736$, we should pick $t=2$ for all $n < 20736$. 
Does it work in practice?

How to pick block size \( t = \frac{\log_{3|\Sigma|} n}{2} \) in practice.

For \( n=1000, |\Sigma| = 4 \), then \( \frac{\log_{12} 1000}{2} = \frac{\log 1000}{\log 12}/2 = 1.39 \), i.e. we should pick \( t=2 \). Since \( \frac{\log_{12} n}{2} < 2 \) iff \( n < 20736 \), we should pick \( t=2 \) for all \( n < 20736 \).

Example (\( t=2 \) and \( |\Sigma| = 4 \))

We can encode the inputs \( B', C', x \) and \( y \) in 16 bits (2 bits for each of the 4 offset in \( B' \) and \( C' \), and 2 bits for each of the 4 symbol in \( x \) and \( y \)). The output \( F' \) can be encoded in 8 bits (2 bits for each of the 4 offset).

The offset block-function \( b' \) can thus be stored as an array \( b'[0..2^{16}-1] \) of bytes, i.e. in 64 Kb in total.

Looking up an entry in this array is fast in practice, so we might observe a speedup (compared to the basic algorithm) by a factor somewhere between \( t \) and \( t^2 \), i.e. 2 and 4.
Does it work in practice?

How to pick block size $t = \frac{\log_{3|\Sigma|} n}{2}$ in practice.

For $n=1000$, $|\Sigma| = 4$, then $(\log_{12} 1000) / 2 = (\log 1000 / \log 12) / 2 = 1.39$, i.e. we should pick $t=2$. Since $(\log_{12} n) / 2 < 2$ iff $n < 20736$, we should pick $t=2$ for all $n < 20736$.

Example ($t=2$ and $|\Sigma| = 4$)

We can encode the inputs $B'$, $C'$, $x$ and $y$ in 24 bits (2 bits for each of the 6 offset in $B'$ and $C'$, and 2 bits for each of the 6 symbol in $x$ and $y$). The output $F'$ can be encoded in 12 bits (2 bits for each of the 6 offset).

The offset block-function $b'$ can thus be stored as an array $b'[0..2^{24}-1]$, where each entry stores 2 bytes (but only uses 12 bits of these), i.e. in 32 Mb in total.

Looking up an entry in this array is fast in practice (but slower that for $t=2$ since the table might not be in cache due to its size), so we might observe a speedup (compared to the basic algorithm) by a factor somewhere between $t$ and $t^2$, i.e. 3 and 9.
Does it work in practice?

How to pick block size \( t = \frac{\log_{|\Sigma|} n}{2} \) in practice.

For \( n=1000, |\Sigma| = 4 \), then \( \frac{\log_{12} 1000}{2} = \frac{\log 1000}{\log 12} / 2 = 1.39 \), i.e. we should pick \( t=2 \). Since \( \frac{\log_{12} n}{2} < 2 \) iff \( n < 20736 \), we should pick \( t=2 \) for all \( n < 20736 \).

Example (\( t=2 \) and \( |\Sigma| = 4 \))

We can encode the inputs \( B', C', x \) and \( y \) in 16 bits (2 bits for each of the 4 offset in \( B' \) and \( C' \), and 2 bits for each of the 4 symbols in \( x \) and \( y \)). The output \( F' \) can be encoded in 8 bits (2 bits for each of the 4 offsets). The offset block-function \( b' \) can thus be stored as an array \( b'_{0..2^{16}-1} \) of bytes, i.e. in 64 Kb in total.

Looking up an entry in this array is fast in practice (but slower than for \( t=2 \) since the table might not be in cache due to its size), so we might observe a speedup (compared to the basic algorithm) by a factor somewhere between \( t \) and \( t^2 \), i.e. 2 and 4.

Example (\( t=3 \) and \( |\Sigma| = 4 \))

We can encode the inputs \( B', C', x \) and \( y \) in 24 bits (2 bits for each of the 6 offset in \( B' \) and \( C' \), and 2 bits for each of the 6 symbols in \( x \) and \( y \)). The output \( F' \) can be encoded in 12 bits (2 bits for each of the 6 offsets). The offset block-function \( b' \) can thus be stored as an array \( b'_{0..2^{24}-1} \), where each entry stores 2 bytes (but only uses 12 bits of these), i.e. in 32 Mb in total.

Looking up an entry in this array is fast in practice (but slower than for \( t=2 \) since the table might not be in cache due to its size), so we might observe a speedup (compared to the basic algorithm) by a factor somewhere between \( t \) and \( t^2 \), i.e. 3 and 9.

Idea for future projects (fx thesis projects)

- Perform experiments to decide if the technique works in practice as sketched here.
- Adapt technique to other score functions than unit cost edit distance (influences how to encode \( B' \) and \( C' \)).
- Adapt to other dynamic programming based solutions (has been adapted to matching of regex and prediction of RNA structure).

Looking up an entry in this array is fast in practice (but slower than for \( t=2 \) since the table might not be in cache due to its size), so we might observe a speedup (compared to the basic algorithm) by a factor somewhere between \( t \) and \( t^2 \), i.e. 3 and 9.
Some results in practice (1)
Some results in practice (2)
What about?

Getting an optimal alignment by backtracking?
What about?

Getting an optimal alignment by backtracking?

Backtrack through the blocks, starting in the lower right block. Before backtracking through a block, we explicit recompute it in time $O(t^2)$. Since we backtrack through at most $2n/t$ blocks, the total time becomes $O(2nt)$. 

![Diagram showing backtracking through blocks with values and a red path]
What about?

Getting an optimal alignment by backtracking?

Backtrack through the blocks, starting in the lower right block. Before backtracking through a block, we explicit recompute it in time $O(t^2)$. Since we backtrack through at most $2n/t$ blocks, the total time becomes $O(2nt)$.
What about?

Other score functions that unit cost edit distance?
What about?

Other score functions that unit cost edit distance?

When using 'unit cost edit distance', we used that neighboring cells in the dynamic programming tables differ by at most 1, i.e. inputs B and C can be encoded as offsets (-1, 0, +1) from A.

If we have a general score function based on a metric substitution matrix and a linear gap cost, then we can likewise show that neighboring cells in the dynamic programming table differ by at most a constant K (depending on the maximum and minimum entries in the substitution matrix, i.e independent of n), so the inputs B and C can still be encoded as offset {-K, -(K-1), ..., 0, K-1, K}. Everything would still work.