Approximate pattern matching/indexing

Paweł Gawrychowski

University of Wrocław & Max-Planck-Institut für Informatik

Up until now we consider **exact** pattern matching / text indexing. Now we would like to relax the setting and look at **approximate** pattern matching / text indexing.

Let us start with approximate pattern matching: we want to detect all fragments of the text which are similar to the pattern. So, what could similar mean?

Pattern matching with k mismatches

Given text $t[1..n]$ and pattern $p[1..m]$, is there i such that $t[i..i+m-1]$ and $p[1..m]$ differ on at most k positions?

Up until now we consider **exact** pattern matching / text indexing. Now we would like to relax the setting and look at **approximate** pattern matching / text indexing.

Let us start with approximate pattern matching: we want to detect all fragments of the text which are similar to the pattern. So, what could similar mean?

Pattern matching with k mismatches

Given text $t[1..n]$ and pattern $p[1..m]$, is there i such that $t[i..i+m-1]$ and $p[1..m]$ differ on at most k positions?

Up until now we consider **exact** pattern matching / text indexing. Now we would like to relax the setting and look at **approximate** pattern matching / text indexing.

Let us start with approximate pattern matching: we want to detect all fragments of the text which are similar to the pattern. So, what could similar mean?

Pattern matching with k mismatches

Given text $t[1..n]$ and pattern $p[1..m]$, is there i such that $t[i..i+m-1]$ and $p[1..m]$ differ on at most k positions?

A simple algorithm for pattern matching with k mismatches:

1. construct the suffix array for $p\$t$,
2. for every $i = 1, 2, \ldots, n - m + 1$ use up to $k + 1$ lcp queries to calculate the mismatches between $t[i..i + m - 1]$ and $p[1..m]$.

The total complexity is $\mathcal{O}(nk)$. Can we do better?

A simple algorithm for pattern matching with k mismatches:

1. construct the suffix array for $p\$t$,
2. for every $i = 1, 2, \ldots, n - m + 1$ use up to $k + 1$ lcp queries to calculate the mismatches between $t[i..i + m - 1]$ and $p[1..m]$.

The total complexity is $\mathcal{O}(nk)$. Can we do better?

A simple algorithm for pattern matching with k mismatches:

1. construct the suffix array for $p\$t$,
2. for every $i = 1, 2, \ldots, n - m + 1$ use up to $k + 1$ lcp queries to calculate the mismatches between $t[i..i + m - 1]$ and $p[1..m]$.

The total complexity is $\mathcal{O}(nk)$. Can we do better?

A simple algorithm for pattern matching with k mismatches:

1. construct the suffix array for $p\$t$,
2. for every $i = 1, 2, \ldots, n - m + 1$ use up to $k + 1$ lcp queries to calculate the mismatches between $t[i..i + m - 1]$ and $p[1..m]$.

The total complexity is $\mathcal{O}(nk)$. Can we do better?

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.

Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.

Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.

Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.
Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.
Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Assume that $|\Sigma|$ is small, say $\Sigma = \{a, b\}$.

We want to count the mismatches between every $t[i..i+m-1]$ and $p[1..m]$. It is enough to count **matches** for every element of Σ, that is, js such that $t[i+j-1] = p[j] = \mathtt{a}$.
Define two polynomials:

$$t(x) = \sum_{i:t[i]=\mathtt{a}} x^{i-1}$$

$$p(x) = \sum_{j:p[j]=\mathtt{a}} x^{m-j}$$

Then the coefficient next to x^k in $t(x) \cdot p(x)$ gives us the desired number of js at starting position $i = k - m$.

FFT

$t(x) \cdot p(x)$ can be computed in $\mathcal{O}(n \log n)$ time with Fast Fourier Transform (FFT).

Repeating the reasoning for every $a \in \Sigma$, we obtain $\mathcal{O}(|\Sigma| n \log n)$ time algorithm. Can be easily improved to $\mathcal{O}(|\Sigma| n \log m)$, do you see how?

Abrahamson 1987

$\mathcal{O}(n\sqrt{m \log m})$ time algorithm exists.

Amir, Lewenstein, and Porat 2004

$\mathcal{O}(n\sqrt{k \log k})$ time algorithm.

Repeating the reasoning for every $a \in \Sigma$, we obtain $\mathcal{O}(|\Sigma| n \log n)$ time algorithm. Can be easily improved to $\mathcal{O}(|\Sigma| n \log m)$, do you see how?

Abrahamson 1987

$\mathcal{O}(n\sqrt{m \log m})$ time algorithm exists.

Amir, Lewenstein, and Porat 2004

$\mathcal{O}(n\sqrt{k \log k})$ time algorithm.

Repeating the reasoning for every $a \in \Sigma$, we obtain $\mathcal{O}(|\Sigma| n \log n)$ time algorithm. Can be easily improved to $\mathcal{O}(|\Sigma| n \log m)$, do you see how?

Abrahamson 1987

$\mathcal{O}(n\sqrt{m \log m})$ time algorithm exists.

Amir, Lewenstein, and Porat 2004

$\mathcal{O}(n\sqrt{k \log k})$ time algorithm.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s,t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s,t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s,t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s,t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s, t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

But mismatches (substitutions) are not the only possible errors. Can we define a more general model?

We define the edit distance between a pair of strings s and t. $d(s, t)$ is the smallest number of changes we need to get t from s, where each operation is one of the following:

- removing a letter,
- inserting a letter,
- changing a letter.

This known as the **Levenshtein** distance.

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Other variants are possible: we can assign a possibly different cost to each operation, and then ask about the cheapest way to transform one string into another.

LCS

One commonly used variant is the longest common subsequence, where the cost of removing and inserting is 1, and the cost of changing is ∞. Then the cost corresponds to....?

So how to compute $d(s, t)$ efficiently, assuming we are concerned about the LCS distance? It turns out that there is a simple recursive formula.

$$d(s[1..n], t[1..m]) =$$

$$= \begin{cases} d(s[1..n-1], t[1..m-1]) + 1 & \text{if } s[n] = t[m], \\ \max(d(s[1..n], t[1..m-1]), d(s[1..n-1], t[1..m])) & \text{otherwise.} \end{cases}$$

We compute a big table $T[i, j] = d(s[1..i], t[1..j])$ with two for loops. Then the complexity becomes just $\mathcal{O}(nm)$.

Would a similar formula work for the Levensthein distance?

LCS

One commonly used variant is the longest common subsequence, where the cost of removing and inserting is 1, and the cost of changing is ∞. Then the cost corresponds to....?

So how to compute $d(s, t)$ efficiently, assuming we are concerned about the LCS distance? It turns out that there is a simple recursive formula.

$$d(s[1..n], t[1..m]) =$$
$$= \begin{cases} d(s[1..n-1], t[1..m-1]) + 1 & \text{if } s[n] = t[m], \\ \max(d(s[1..n], t[1..m-1]), d(s[1..n-1], t[1..m])) & \text{otherwise.} \end{cases}$$

We compute a big table $T[i, j] = d(s[1..i], t[1..j])$ with two for loops. Then the complexity becomes just $\mathcal{O}(nm)$.

Would a similar formula work for the Levensthein distance?

LCS

One commonly used variant is the longest common subsequence, where the cost of removing and inserting is 1, and the cost of changing is ∞. Then the cost corresponds to....?

So how to compute $d(s,t)$ efficiently, assuming we are concerned about the LCS distance? It turns out that there is a simple recursive formula.

$$d(s[1..n], t[1..m]) =$$
$$= \begin{cases} d(s[1..n-1], t[1..m-1]) + 1 & \text{if } s[n] = t[m], \\ \max(d(s[1..n], t[1..m-1]), d(s[1..n-1], t[1..m])) & \text{otherwise.} \end{cases}$$

We compute a big table $T[i,j] = d(s[1..i], t[1..j])$ with two for loops. Then the complexity becomes just $\mathcal{O}(nm)$.

Would a similar formula work for the Levensthein distance?

LCS

One commonly used variant is the longest common subsequence, where the cost of removing and inserting is 1, and the cost of changing is ∞. Then the cost corresponds to....?

So how to compute $d(s,t)$ efficiently, assuming we are concerned about the LCS distance? It turns out that there is a simple recursive formula.

$$d(s[1..n], t[1..m]) =$$
$$= \begin{cases} d(s[1..n-1], t[1..m-1]) + 1 & \text{if } s[n] = t[m], \\ \max(d(s[1..n], t[1..m-1]), d(s[1..n-1], t[1..m])) & \text{otherwise.} \end{cases}$$

We compute a big table $T[i,j] = d(s[1..i], t[1..j])$ with two for loops. Then the complexity becomes just $\mathcal{O}(nm)$.

Would a similar formula work for the Levenshtein distance?

LCS

One commonly used variant is the longest common subsequence, where the cost of removing and inserting is 1, and the cost of changing is ∞. Then the cost corresponds to....?

So how to compute $d(s, t)$ efficiently, assuming we are concerned about the LCS distance? It turns out that there is a simple recursive formula.

$$d(s[1..n], t[1..m]) =$$
$$= \begin{cases} d(s[1..n-1], t[1..m-1]) + 1 & \text{if } s[n] = t[m], \\ \max(d(s[1..n], t[1..m-1]), d(s[1..n-1], t[1..m])) & \text{otherwise.} \end{cases}$$

We compute a big table $T[i, j] = d(s[1..i], t[1..j])$ with two for loops. Then the complexity becomes just $\mathcal{O}(nm)$.

Would a similar formula work for the Levensthein distance?

That is all very nice, but quadratic complexity is not very useful. The edit distance is exactly what is internally used in the `diff` utility to compare files (although there a single line become a single character), and this utility works quickly even for fairly large files. How?

Allocating a table of size $n \cdot m$ might or might not be possible for large values of n and m.

That is all very nice, but quadratic complexity is not very useful. The edit distance is exactly what is internally used in the `diff` utility to compare files (although there a single line become a single character), and this utility works quickly even for fairly large files. How?

Allocating a table of size $n \cdot m$ might or might not be possible for large values of n and m.

Computing the edit distance:

$$
\begin{aligned}
d(s[1..n], t[1..m]) = \min(& \\
& d(s[1..n-1], t[1..m]) + 1, \\
& d(s[1..n], t[1..m-1]) + 1, \\
& d(s[1..n-1], t[1..m-1]) + [s[n] \neq t[m]] \\
&)
\end{aligned}
$$

We can use a simple trick: to compute the i-th row of the table we don't need the first, second, ..., $i-2$-th one. Hence we only store the current row and the previous one during our computation. Then the space complexity becomes $\mathcal{O}(m)$, and the time complexity stays the same.

Computing the edit distance:

$$
\begin{aligned}
d(s[1..n], t[1..m]) = \min(& \\
& d(s[1..n-1], t[1..m]) + 1, \\
& d(s[1..n], t[1..m-1]) + 1, \\
& d(s[1..n-1], t[1..m-1]) + [s[n] \neq t[m]] \\
&)
\end{aligned}
$$

We can use a simple trick: to compute the i-th row of the table we don't need the first, second, ..., $i-2$-th one. Hence we only store the current row and the previous one during our computation. Then the space complexity becomes $\mathcal{O}(m)$, and the time complexity stays the same.

That is all nice when you are only interested in the distance itself. But what if you would like to actually compute the sequence of operations? Or, in the case of LCS, output the longest common subsequence?

Hirschberg 1975

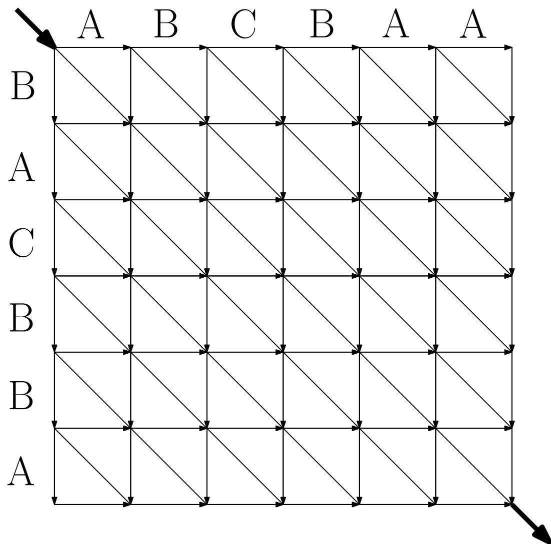
The longest common subsequence can be reconstructed in $\mathcal{O}(nm)$ time and $\mathcal{O}(n+m)$ space.

That is all nice when you are only interested in the distance itself. But what if you would like to actually compute the sequence of operations? Or, in the case of LCS, output the longest common subsequence?

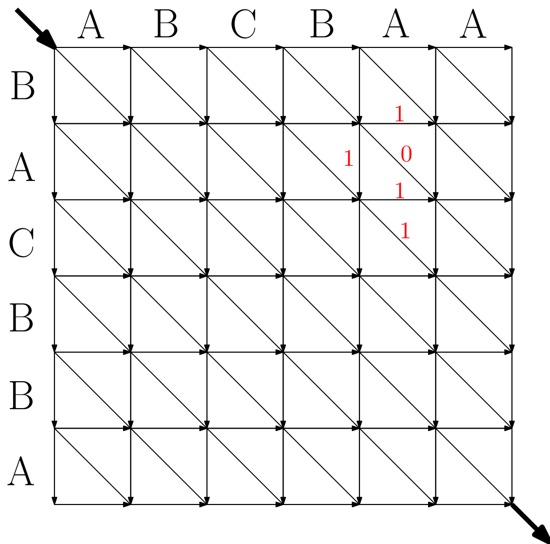
Hirschberg 1975

The longest common subsequence can be reconstructed in $\mathcal{O}(nm)$ time and $\mathcal{O}(n+m)$ space.

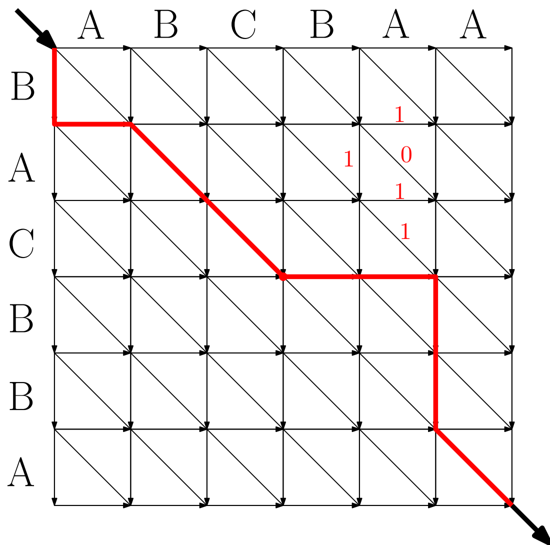
We start with visualizing the DP solution in a slightly different manner.



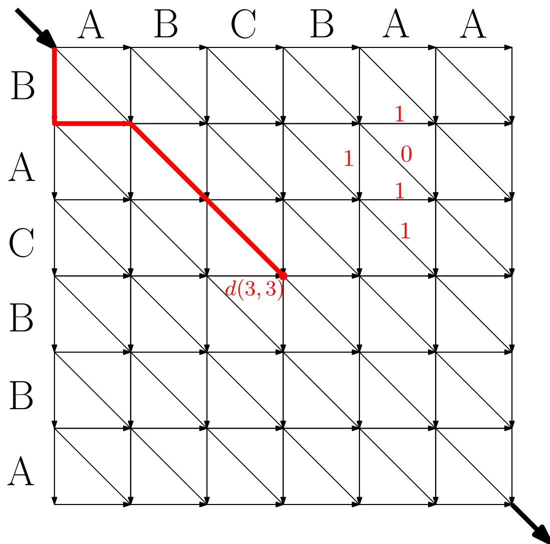
We start with visualizing the DP solution in a slightly different manner.



We start with visualizing the DP solution in a slightly different manner.



We start with visualizing the DP solution in a slightly different manner.



$d(i, j)$ is the weight of the cheapest path from $(0, 0)$ to (i, j).

Main trick

Can we compute $d'(i, j)$, the weight of the cheapest path from (i, j) to (n, m), using a similar method?

Lemma

For any $x \in [0, n]$, the edit distance is $\min_{y \in [0,m]} d(x, y) + d'(x, y)$.

$d(i, j)$ is the weight of the cheapest path from $(0, 0)$ to (i, j).

Main trick

Can we compute $d'(i, j)$, the weight of the cheapest path from (i, j) to (n, m), using a similar method?

Lemma

For any $x \in [0, n]$, the edit distance is $\min_{y \in [0,m]} d(x, y) + d'(x, y)$.

$d(i, j)$ is the weight of the cheapest path from $(0, 0)$ to (i, j).

Main trick

Can we compute $d'(i, j)$, the weight of the cheapest path from (i, j) to (n, m), using a similar method?

Lemma

For any $x \in [0, n]$, the edit distance is $\min_{y \in [0,m]} d(x, y) + d'(x, y)$.

A recursive method for reconstructing the cheapest path

- choose any $x \in [0, n]$
- compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$
- select y minimizing the sum
- recursively reconstruct the path in two smaller grids corresponding to $d(s[1..x], t[1..y])$ and $d(s[x+1..n], t[y+1..m])$.

A recursive method for reconstructing the cheapest path

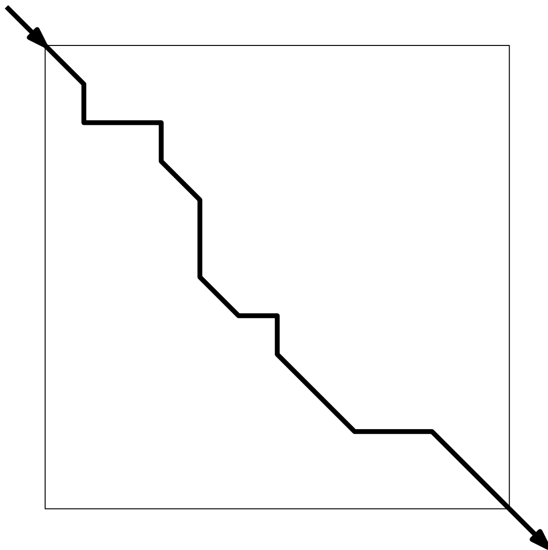
- choose any $x \in [0, n]$
- compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$
- select y minimizing the sum
- recursively reconstruct the path in two smaller grids corresponding to $d(s[1..x], t[1..y])$ and $d(s[x+1..n], t[y+1..m])$.

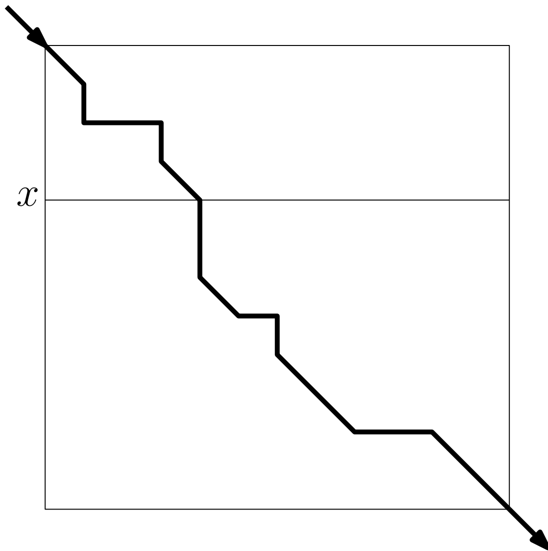
A recursive method for reconstructing the cheapest path

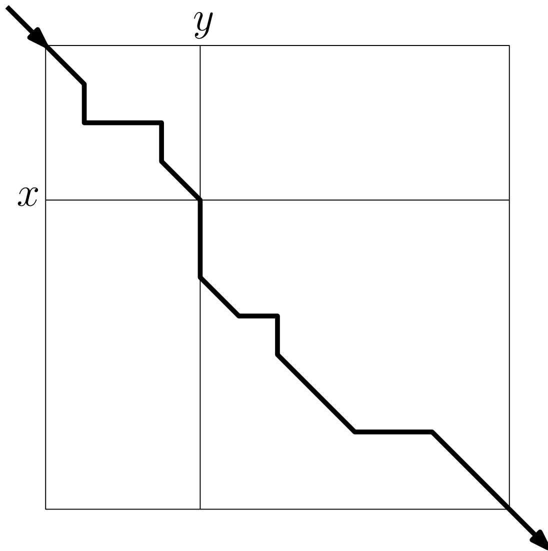
- choose any $x \in [0, n]$
- compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$
- select y minimizing the sum
- recursively reconstruct the path in two smaller grids corresponding to $d(s[1..x], t[1..y])$ and $d(s[x+1..n], t[y+1..m])$.

A recursive method for reconstructing the cheapest path

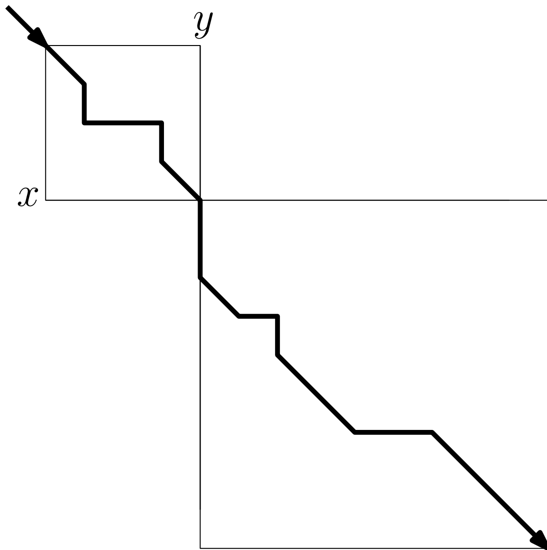
- choose any $x \in [0, n]$
- compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$
- select y minimizing the sum
- recursively reconstruct the path in two smaller grids corresponding to $d(s[1..x], t[1..y])$ and $d(s[x + 1..n], t[y + 1..m])$.




x



y
x



y
x

How to choose x?

$$T(n,m) = T(x,y) + T(n-x, m-y) + \mathcal{O}(nm)$$

Now choose $x = \frac{n}{2}$.

$$T(n,m) = T(\frac{n}{2}, y) + T(\frac{n}{2}, m-y) + \mathcal{O}(nm)$$

If you think that nm is the "size" of the problem, you can see that the combined size of the two smaller subproblems is half as big as the original size. It follows that $T(n,m) = \mathcal{O}(nm)$.

How to choose x?

$$T(n, m) = T(x, y) + T(n - x, m - y) + \mathcal{O}(nm)$$

Now choose $x = \frac{n}{2}$.

$$T(n, m) = T(\frac{n}{2}, y) + T(\frac{n}{2}, m - y) + \mathcal{O}(nm)$$

If you think that *nm* is the “size” of the problem, you can see that the combined size of the two smaller subproblems is half as big as the original size. It follows that $T(n, m) = \mathcal{O}(nm)$.

How to choose x?

$$T(n,m) = T(x,y) + T(n-x, m-y) + \mathcal{O}(nm)$$

Now choose $x = \frac{n}{2}$.

$$T(n,m) = T(\frac{n}{2}, y) + T(\frac{n}{2}, m-y) + \mathcal{O}(nm)$$

If you think that nm is the “size” of the problem, you can see that the combined size of the two smaller subproblems is half as big as the original size. It follows that $T(n,m) = \mathcal{O}(nm)$.

How to choose x?

$$T(n,m) = T(x,y) + T(n-x, m-y) + \mathcal{O}(nm)$$

Now choose $x = \frac{n}{2}$.

$$T(n,m) = T(\frac{n}{2}, y) + T(\frac{n}{2}, m-y) + \mathcal{O}(nm)$$

If you think that *nm* is the "size" of the problem, you can see that the combined size of the two smaller subproblems is half as big as the original size. It follows that $T(n,m) = \mathcal{O}(nm)$.

How to choose x?

$$T(n, m) = T(x, y) + T(n - x, m - y) + \mathcal{O}(nm)$$

Now choose $x = \frac{n}{2}$.

$$T(n, m) = T(\frac{n}{2}, y) + T(\frac{n}{2}, m - y) + \mathcal{O}(nm)$$

If you think that nm is the "size" of the problem, you can see that the combined size of the two smaller subproblems is half as big as the original size. It follows that $T(n, m) = \mathcal{O}(nm)$.

So, the time consumption is $\mathcal{O}(nm)$. What about space? We compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$ using the row-by-row space saving method, then the space complexity is just $\mathcal{O}(n + m)$.

The same trick works for other scoring functions, and in fact for many dynamic programming solutions.

So, the time consumption is $\mathcal{O}(nm)$. What about space? We compute $d(x, y)$ and $d'(x, y)$ for all $y \in [0, m]$ using the row-by-row space saving method, then the space complexity is just $\mathcal{O}(n + m)$.

The same trick works for other scoring functions, and in fact for many dynamic programming solutions.

Even though we decrease the space consumption, computing the edit distance between two long strings is still infeasible. We will try to develop a method which works efficiently when the strings are not very different, i.e., when $d(s, t)$ is not large.

Myers 1986

Edit distance can be computed in time $\mathcal{O}(nD)$, where $D = d(s, t)$.

Why?

Think how you are using `diff`. If D is large, you probably don't care about the edit distance anyway.

Even though we decrease the space consumption, computing the edit distance between two long strings is still infeasible. We will try to develop a method which works efficiently when the strings are not very different, i.e., when $d(s, t)$ is not large.

Myers 1986

Edit distance can be computed in time $\mathcal{O}(nD)$, where $D = d(s, t)$.

Why?

Think how you are using `diff`. If D is large, you probably don't care about the edit distance anyway.

Even though we decrease the space consumption, computing the edit distance between two long strings is still infeasible. We will try to develop a method which works efficiently when the strings are not very different, i.e., when $d(s, t)$ is not large.

Myers 1986

Edit distance can be computed in time $\mathcal{O}(nD)$, where $D = d(s, t)$.

Why?

Think how you are using `diff`. If D is large, you probably don't care about the edit distance anyway.

Consider a diagonal, i.e., all points (i, j) such that $i - j = \delta$. What can we say about the corresponding $d(i, j)$?

Lemma

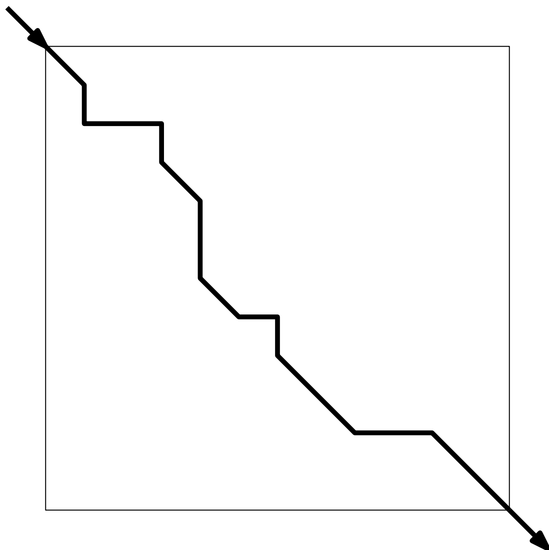
$$d(i, j) \geq |i - j|$$

Consider a diagonal, i.e., all points (i, j) such that $i - j = \delta$. What can we say about the corresponding $d(i, j)$?

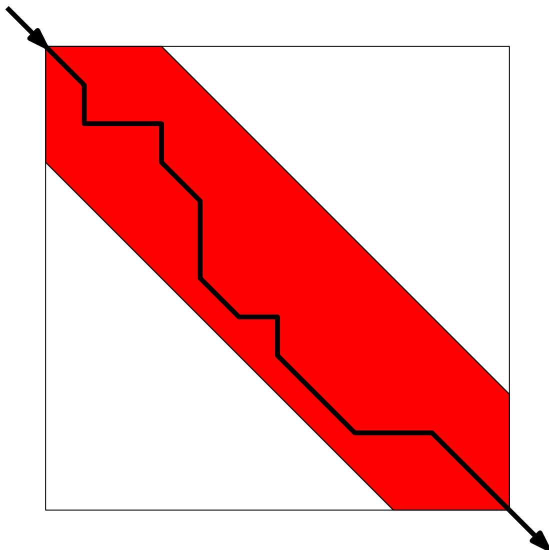
Lemma

$$d(i, j) \geq |i - j|$$

Assume that we know the value of D (we don't, but don't worry about that for the time being). Do we need to compute all $d(i,j)$?



Assume that we know the value of D (we don't, but don't worry about that for the time being). Do we need to compute all $d(i,j)$?



Observation

If the edit distance is D, it is enough to compute the values of $d(s, t)$ in the diagonal strip of "width" $2D$ in $\mathcal{O}(nD)$ time.

Notice that we really need that the values of $d(s[1..i], t[1..j])$ are non-decreasing here, otherwise it might make sense to go far away from the strip and then go back.

But we don't know the value of D!

Observation

If the edit distance is D, it is enough to compute the values of $d(s, t)$ in the diagonal strip of "width" $2D$ in $\mathcal{O}(nD)$ time.

Notice that we really need that the values of $d(s[1..i], t[1..j])$ are non-decreasing here, otherwise it might make sense to go far away from the strip and then go back.

But we don't know the value of D!

Observation

If the edit distance is D, it is enough to compute the values of $d(s,t)$ in the diagonal strip of "width" $2D$ in $\mathcal{O}(nD)$ time.

Notice that we really need that the values of $d(s[1..i], t[1..j])$ are non-decreasing here, otherwise it might make sense to go far away from the strip and then go back.

But we don't know the value of D!

Final trick

Say that we choose some value of D. If $d(s, t) \leq D$, the above method will return the correct answer. If $d(s, t) > D$, our result will exceed D, too. Hence we can verify whether our guess was correct.

Try $D = 1, 2, 3, \ldots$. What is the resulting complexity?
OK, so maybe try $D = 2^0, 2^1, 2^2, \ldots$. What is the resulting complexity now?

Final trick

Say that we choose some value of D. If $d(s,t) \leq D$, the above method will return the correct answer. If $d(s,t) > D$, our result will exceed D, too. Hence we can verify whether our guess was correct.

Try $D = 1, 2, 3, \ldots$. What is the resulting complexity?

OK, so maybe try $D = 2^0, 2^1, 2^2, \ldots$. What is the resulting complexity now?

Final trick

Say that we choose some value of D. If $d(s,t) \leq D$, the above method will return the correct answer. If $d(s,t) > D$, our result will exceed D, too. Hence we can verify whether our guess was correct.

Try $D = 1, 2, 3, \ldots$. What is the resulting complexity?
OK, so maybe try $D = 2^0, 2^1, 2^2, \ldots$. What is the resulting complexity now?

So we have an $\mathcal{O}(nD)$ time algorithm. Can we do better?

Edit distance can be computed in time $\mathcal{O}(n + D^2)$.

So we have an $\mathcal{O}(nD)$ time algorithm. Can we do better?

Edit distance can be computed in time $\mathcal{O}(n + D^2)$.

Look at a single diagonal. What can we say about the consecutive values of $d(i,j)$ there?

Observation

$$d(i+1,j+1) \in \{d(i,j), d(i,j)+1\}$$

Hence the values on a single diagonal are non-decreasing. As in the $\mathcal{O}(nD)$ time algorithm, as soon as $d(i,j) > D$, we don't care about the exact value, as it cannot possible correspond to a path of total cost at most D.

Look at a single diagonal. What can we say about the consecutive values of $d(i, j)$ there?

Observation

$$d(i+1, j+1) \in \{d(i, j), d(i, j)+1\}$$

Hence the values on a single diagonal are non-decreasing. As in the $\mathcal{O}(nD)$ time algorithm, as soon as $d(i, j) > D$, we don't care about the exact value, as it cannot possible correspond to a path of total cost at most D.

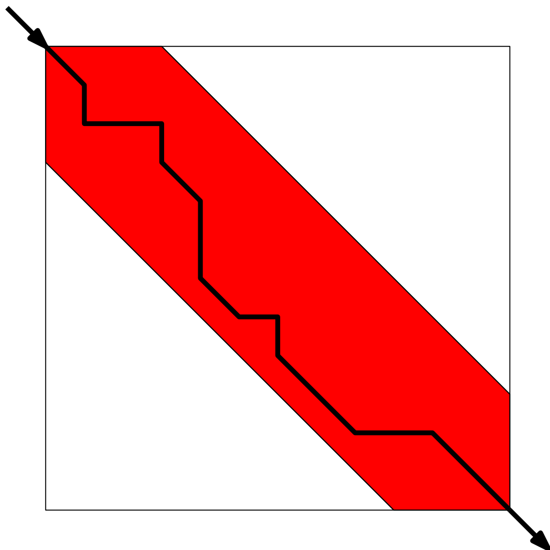
Look at a single diagonal. What can we say about the consecutive values of $d(i,j)$ there?

Observation

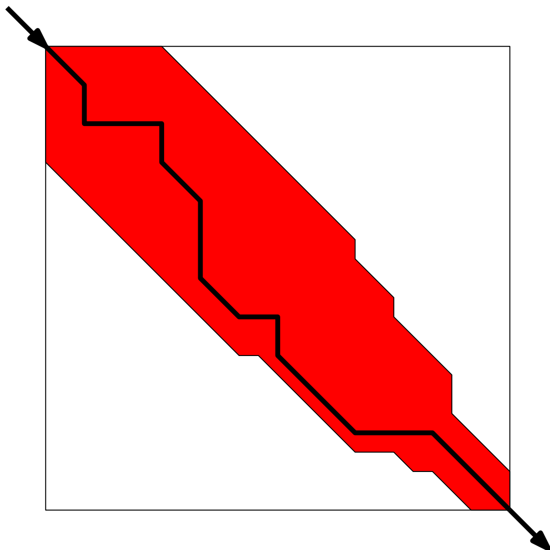
$$d(i+1, j+1) \in \{d(i,j), d(i,j)+1\}$$

Hence the values on a single diagonal are non-decreasing. As in the $\mathcal{O}(nD)$ time algorithm, as soon as $d(i,j) > D$, we don't care about the exact value, as it cannot possible correspond to a path of total cost at most D.

On each diagonal $i - j = \delta$ we compute only those values that are at most D, i.e., we look at (i, j) such that $|i - j| \leq D$ and $d(i, j) \leq D$.



On each diagonal $i - j = \delta$ we compute only those values that are at most D, i.e., we look at (i, j) such that $|i - j| \leq D$ and $d(i, j) \leq D$.



Now this doesn't really decrease the complexity. We need one more observation.

Succinct description of diagonals

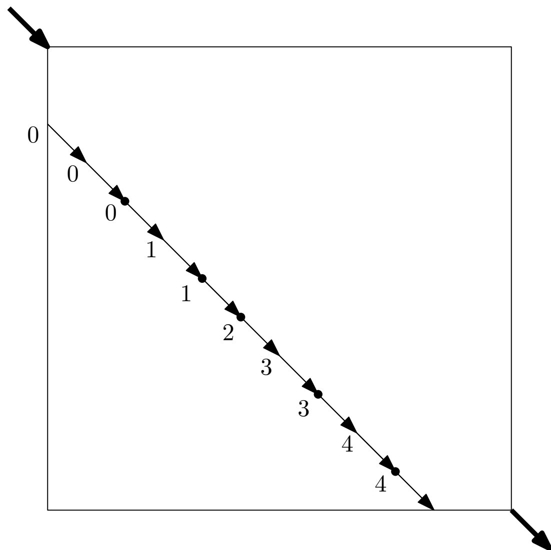
To fully describe the values of $d(i,j)$ for all (i,j) such that $i-j=\delta$ and $d(i,j) \le D$ it is enough to compute for each $x=0,1,\ldots,D$ the last (i,j) on the diagonal such that $d(i,j)=x$.

Now this doesn't really decrease the complexity. We need one more observation.

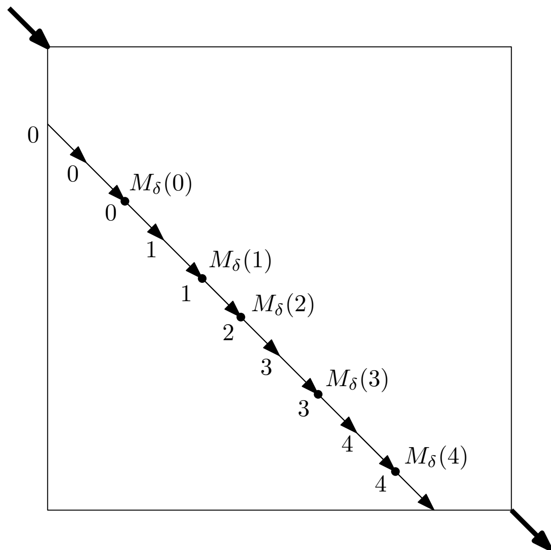
Succinct description of diagonals

To fully describe the values of $d(i,j)$ for all (i,j) such that $i - j = \delta$ and $d(i,j) \leq D$ it is enough to compute for each $x = 0, 1, \ldots, D$ the last (i,j) on the diagonal such that $d(i,j) = x$.

To make the description simpler, let $M_\delta(x)$ be the last (i, j) on the diagonal $i - j = \delta$ such that $d(i, j) = x$.



To make the description simpler, let $M_\delta(x)$ be the last (i, j) on the diagonal $i - j = \delta$ such that $d(i, j) = x$.



Now the question is whether there is some clever way to compute all $\mathcal{O}(D^2)$ values $M_\delta(x)$ in a reasonable time.

Yes!

All values of $M_\delta(x+1)$ can be computed in $\mathcal{O}(D)$ time given all values of $M_\delta(x)$.

The idea is that we look at $m_\delta(x+1)$, which is the first (i,j) on the diagonal such that $d(i,j) = x+1$. Then it must originate from a neighboring diagonal, say $i'-j' = \delta+1$ with, with $d(i',j') = x$! Hence we can compute all $m_\delta(x+1)$ in $\mathcal{O}(D)$ time. Then for each of them we look how far the “free” edges (i.e., with cost 0) extend. This is exactly what we have the longest common prefix for!

Now the question is whether there is some clever way to compute all $\mathcal{O}(D^2)$ values $M_\delta(x)$ in a reasonable time.

Yes!

All values of $M_\delta(x+1)$ can be computed in $\mathcal{O}(D)$ time given all values of $M_\delta(x)$.

The idea is that we look at $m_\delta(x+1)$, which is the first (i,j) on the diagonal such that $d(i,j) = x+1$. Then it must originate from a neighboring diagonal, say $i' - j' = \delta + 1$ with, with $d(i',j') = x$! Hence we can compute all $m_\delta(x+1)$ in $\mathcal{O}(D)$ time. Then for each of them we look how far the “free” edges (i.e., with cost 0) extend. This is exactly what we have the longest common prefix for!

Now the question is whether there is some clever way to compute all $\mathcal{O}(D^2)$ values $M_\delta(x)$ in a reasonable time.

Yes!

All values of $M_\delta(x+1)$ can be computed in $\mathcal{O}(D)$ time given all values of $M_\delta(x)$.

The idea is that we look at $m_\delta(x+1)$, which is the first (i,j) on the diagonal such that $d(i,j) = x+1$. Then it must originate from a neighboring diagonal, say $i' - j' = \delta + 1$ with, with $d(i',j') = x$! Hence we can compute all $m_\delta(x+1)$ in $\mathcal{O}(D)$ time. Then for each of them we look how far the "free" edges (i.e., with cost 0) extend. This is exactly what we have the longest common prefix for!

Now the question is whether there is some clever way to compute all $\mathcal{O}(D^2)$ values $M_\delta(x)$ in a reasonable time.

Yes!

All values of $M_\delta(x+1)$ can be computed in $\mathcal{O}(D)$ time given all values of $M_\delta(x)$.

The idea is that we look at $m_\delta(x+1)$, which is the first (i,j) on the diagonal such that $d(i,j) = x+1$. Then it must originate from a neighboring diagonal, say $i' - j' = \delta + 1$ with, with $d(i',j') = x$! Hence we can compute all $m_\delta(x+1)$ in $\mathcal{O}(D)$ time. Then for each of them we look how far the "free" edges (i.e., with cost 0) extend. This is exactly what we have the longest common prefix for!

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \le j$ such that $d(t[i..j], p[1..m]) \le k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Recall that we actually want more than just computing the edit distance:

Pattern matching with k errors

Given text $t[1..n]$ and pattern $p[1..m]$, is there $i \leq j$ such that $d(t[i..j], p[1..m]) \leq k$?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

The idea is very similar to the $\mathcal{O}(nD)$ time algorithm for computing the edit distance:

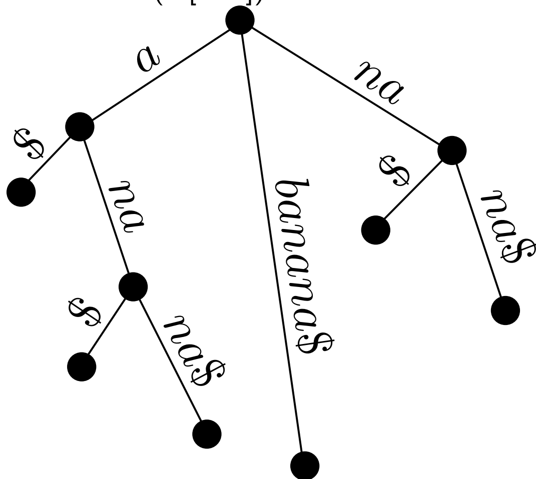
1. start with a dynamic programming formulation. Now the path can start anywhere in the first row instead of the upper-left corner.
2. Observe that the values on every diagonal are non-decreasing, hence again we have up to k "important"positions on every diagonal.
3. Show that these important positions can be computed one-by-one in constant time each with lcp queries.

Now we will briefly look at approximate text indexing.

Recall the definition of $ST(w[1..n])$:

Now we will briefly look at approximate text indexing.
Recall the definition of $ST(w[1..n])$:

Now we will briefly look at approximate text indexing.
Recall the definition of $ST(w[1..n])$:



We will start with a simpler version, where we actually do not allow any errors/mismatches, but the pattern is allowed to contain don't care characters. For instance, what are the occurrences of `?n`?

Naive solution

For each `?`, check all $|\Sigma|$ possibilities.

We will see how to decrease the query time to $\mathcal{O}(m + 2^k \log\log n)$. However, the size of the structure will be around $\mathcal{O}(n \log^k n)$.

We will start with a simpler version, where we actually do not allow any errors/mismatches, but the pattern is allowed to contain don't care characters. For instance, what are the occurrences of `?n`?

Naive solution

For each `?`, check all $|\Sigma|$ possibilities.

We will see how to decrease the query time to $\mathcal{O}(m + 2^k \log\log n)$. However, the size of the structure will be around $\mathcal{O}(n \log^k n)$.

We will start with a simpler version, where we actually do not allow any errors/mismatches, but the pattern is allowed to contain don't care characters. For instance, what are the occurrences of `?n`?

Naive solution

For each `?`, check all $|\Sigma|$ possibilities.

We will see how to decrease the query time to $\mathcal{O}(m + 2^k \log\log n)$. However, the size of the structure will be around $\mathcal{O}(n \log^k n)$.

Heavy path decomposition

Every node select the edge leading to the child with the largest number of nodes in its subtree.

A few properties:

1. this decomposes the tree into node-disjoint paths,
2. the number of paths above any leaf is at most $\log n$.

Heavy path decomposition

Every node select the edge leading to the child with the largest number of nodes in its subtree.

A few properties:

1. this decomposes the tree into node-disjoint paths,
2. the number of paths above any leaf is at most $\log n$.

Heavy path decomposition

Every node select the edge leading to the child with the largest number of nodes in its subtree.

A few properties:

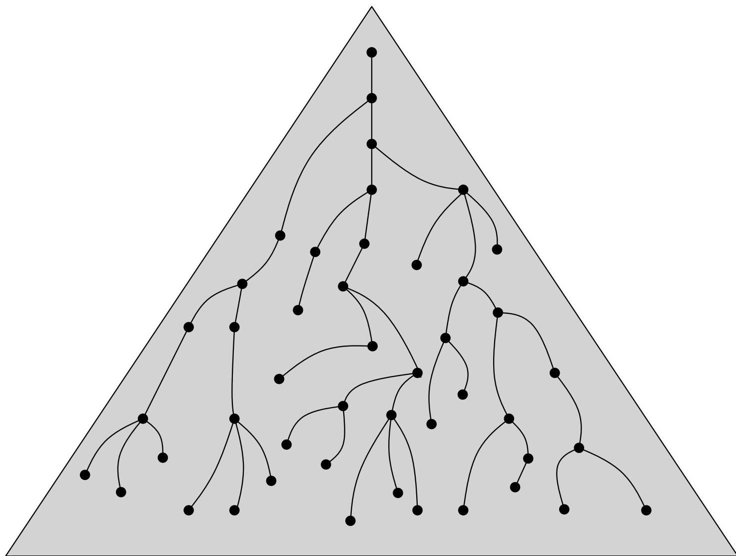
1. this decomposes the tree into node-disjoint paths,
2. the number of paths above any leaf is at most $\log n$.

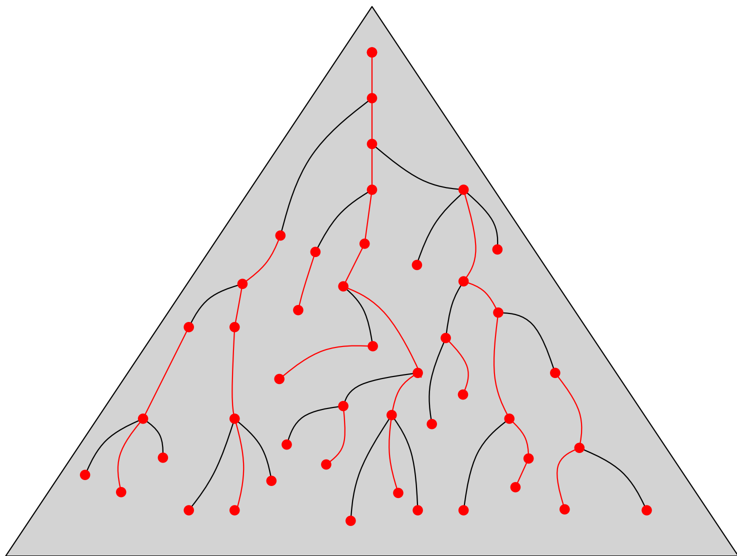
Heavy path decomposition

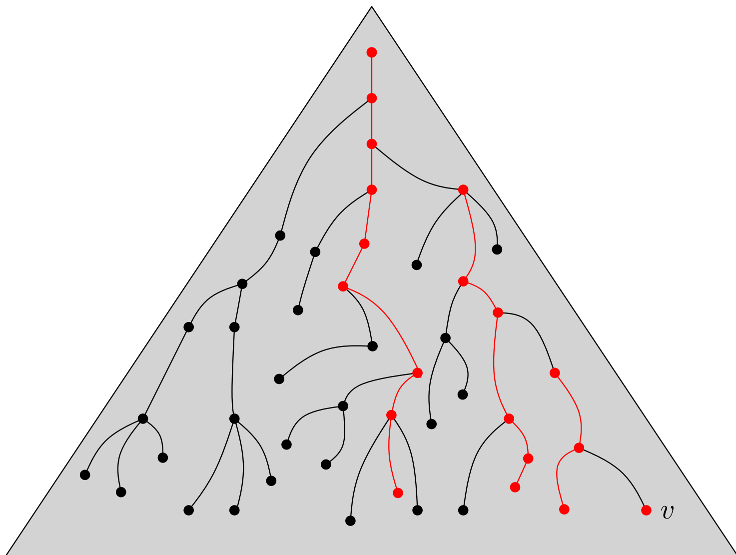
Every node select the edge leading to the child with the largest number of nodes in its subtree.

A few properties:

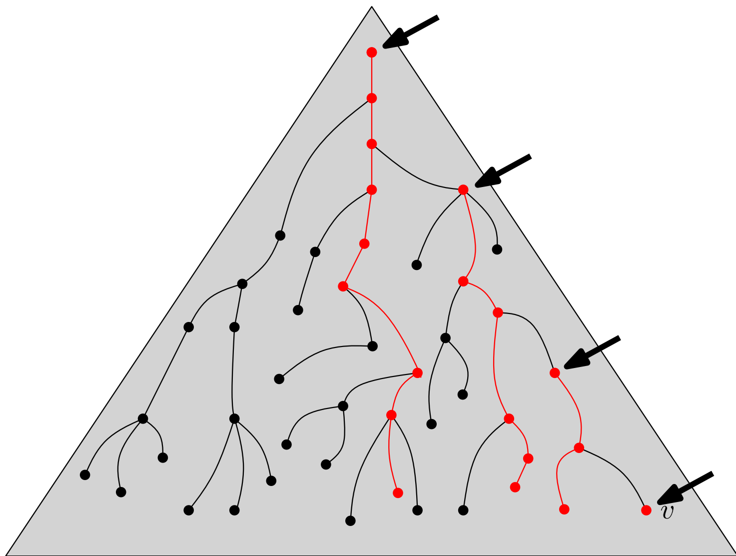
1. this decomposes the tree into node-disjoint paths,
2. the number of paths above any leaf is at most $\log n$.







v


v

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log \log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log \log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log \log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log \log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log \log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log \log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Let the pattern be $p = p_1 ? p_2$. We first find the node v corresponding to p_1 in the suffix tree. Now, instead of trying all $|\Sigma|$ possibilities, we want to check only two of them:

1. continuing along the heavy path containing v,
2. descending to some off-path child of v.

The first case is straightforward to implement. For the second case, we gather all off-path subtrees hanging from v and merge them into one. Then, we can descend into the larger merged subtree instead of descending into every subtree separately.

LCP structure

We need an efficient method for locating the node v (naively, this takes $\mathcal{O}(m)$ time). However, $\mathcal{O}(\log\log n)$ is possible. Intuitively, because we can binary search over the leaves of our tree to find the one with the largest longest common prefix to our p_1. To obtain $\mathcal{O}(\log\log n)$ instead of $\mathcal{O}(\log n)$, binary search is replaced with a predecessor structure (essentially, a smarter version of a balanced search tree).

Questions?