UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

# Error Tolerant Pattern Matching II

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

# Overview

## Previous Lecture

- Pattern Matching with respect to edit distance
- **Semiglobal** Alignment:
  Compare one full string (pattern) against substrings of a longer string (text)
- Basic algorithm
- Ukkonen's speed-up
- Ideas of Myers' bit vector algorithm

# Overview

## Previous Lecture

- Pattern Matching with respect to edit distance
- **Semiglobal** Alignment:
  Compare one full string (pattern) against substrings of a longer string (text)
- Basic algorithm
- Ukkonen's speed-up
- Ideas of Myers' bit vector algorithm

## Today's Lecture

- Adapting NFAs (and Shift-And) to error tolerant search
- Combining NFAs and full text indexing (FM index)
- Four Russians' Method

# Reminder: Error Tolerant Pattern Matching

## Problem Definition

- For two strings $P, T \in \Sigma^*$, find **approximate occurrences** of $P$ in $T$.
- **Formally:** Find **intervals** $[i, j]$ of $T$ such that the **edit distance** between $P$ and $T[i \dots j]$ is below a given threshold $k$.

## Variants

- **Decision Problem:** Is there an interval ... ?
- **Counting Problem:** How many intervals ... ?
- **Enumeration Problem:** List all intervals ....
- **Optimization Problem:** Find an interval $[i, j]$ with the smallest edit distance between $P$ and $T[i \dots j]$ among all (no threshold $k$ given).

# Reminder: NFA for the Exact Pattern Matching Problem

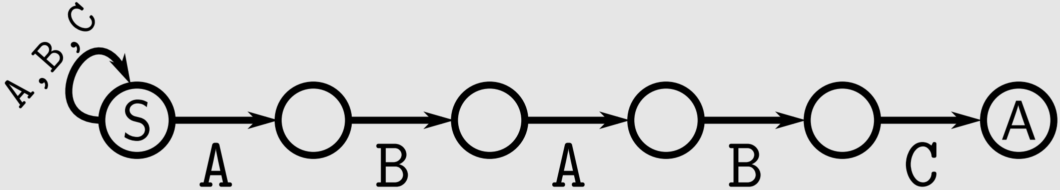
## Goal

For given pattern $P \in \Sigma^*$, construct NFA that recognizes all strings $\Sigma^* P$.

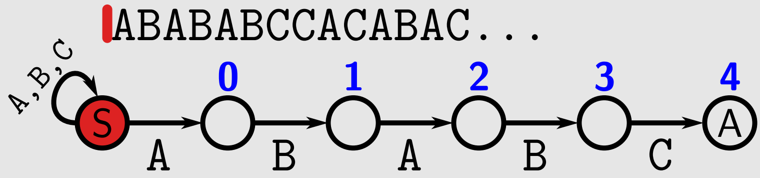
## Approach

- "Linear chain" of states
- Start state remains always active

## Example: $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



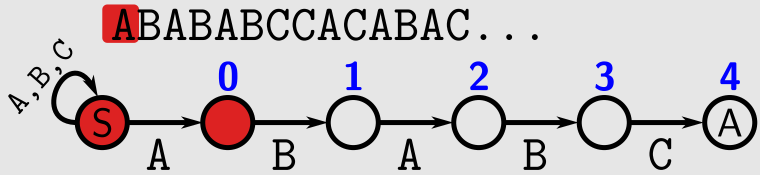
# Reminder: The Shift-And Algorithm




| | | | | | |
|---|---|---|---|---|---|
| *D*: | 0 | 0 | 0 | 0 | 0 |
| *mask*(A): | 1 | 0 | 1 | 0 | 0 |
| *mask*(B): | 0 | 1 | 0 | 1 | 0 |
| *mask*(C): | 0 | 0 | 0 | 0 | 1 |

$$D \leftarrow \big((D \ll 1)|1\big)\ \&\ \mathrm{mask}[c]$$

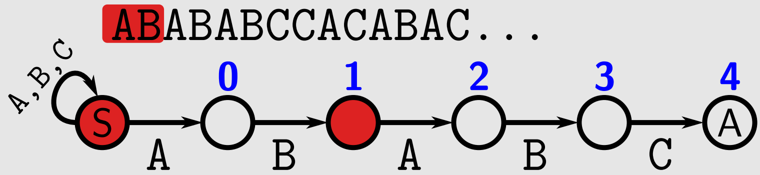
# Reminder: The Shift-And Algorithm




| | | | | | |
|---|---|---|---|---|---|
| $D$: | 1 | 0 | 0 | 0 | 0 |
| $mask(\mathtt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\mathtt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\mathtt{C})$: | 0 | 0 | 0 | 0 | 1 |

$D \leftarrow ((D \ll 1)|1)\ \&\ \text{mask}[c]$

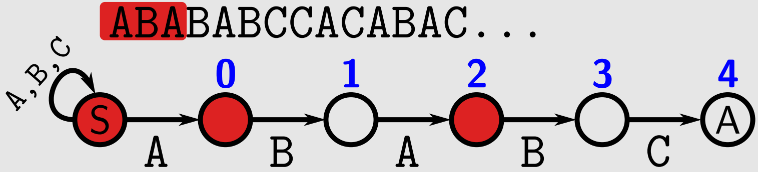
# Reminder: The Shift-And Algorithm




| | | | | | |
|---|---|---|---|---|---|
| $D$: | 0 | 1 | 0 | 0 | 0 |
| $mask(\texttt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\texttt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\texttt{C})$: | 0 | 0 | 0 | 0 | 1 |

$$D \leftarrow \big((D \ll 1)|1\big)\ \&\ \mathrm{mask}[c]$$

# Reminder: The Shift-And Algorithm

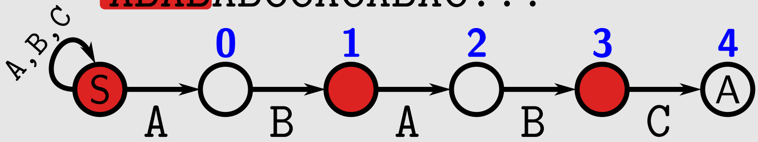



| | | | | | |
|---|---|---|---|---|---|
| $D$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\mathtt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\mathtt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\mathtt{C})$: | 0 | 0 | 0 | 0 | 1 |

$$D \leftarrow \big((D \ll 1)|1\big) \ \& \ \mathrm{mask}[c]$$

# Reminder: The Shift-And Algorithm

`ABABABCCACABAC...`

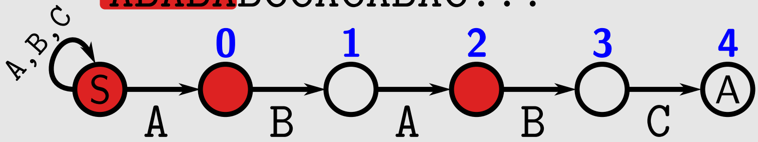



| | | | | | |
|---|---|---|---|---|---|
| *D*: | 0 | 1 | 0 | 1 | 0 |
| *mask*(A): | 1 | 0 | 1 | 0 | 0 |
| *mask*(B): | 0 | 1 | 0 | 1 | 0 |
| *mask*(C): | 0 | 0 | 0 | 0 | 1 |

$D \leftarrow ((D \ll 1)|1)\ \&\ \text{mask}[c]$

# Reminder: The Shift-And Algorithm

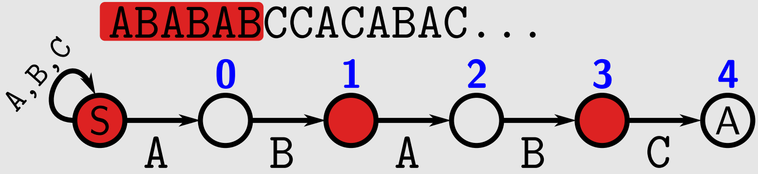
`ABABABCCACABAC...`




| | | | | | |
|---|---|---|---|---|---|
| $D$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\texttt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\texttt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\texttt{C})$: | 0 | 0 | 0 | 0 | 1 |

$$D \leftarrow \big((D \ll 1)|1\big)\ \&\ \mathrm{mask}[c]$$

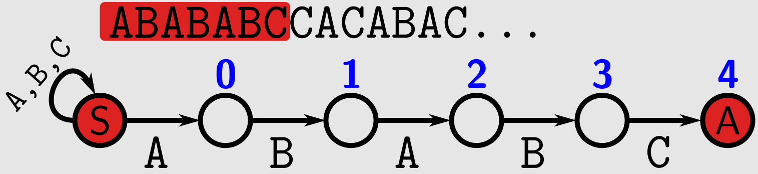
# Reminder: The Shift-And Algorithm




| | | | | | |
|---|---|---|---|---|---|
| $D$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\texttt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\texttt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\texttt{C})$: | 0 | 0 | 0 | 0 | 1 |

$D \leftarrow ((D \ll 1)|1)$ & $\mathrm{mask}[c]$

# Reminder: The Shift-And Algorithm




| | | | | | |
|---|---|---|---|---|---|
| $D$: | 0 | 0 | 0 | 0 | 1 |
| $mask(\texttt{A})$: | 1 | 0 | 1 | 0 | 0 |
| $mask(\texttt{B})$: | 0 | 1 | 0 | 1 | 0 |
| $mask(\texttt{C})$: | 0 | 0 | 0 | 0 | 1 |

$$D \leftarrow \big((D \ll 1)|1\big) \ \& \ \mathrm{mask}[c]$$

# Extension to Error Tolerant Pattern Search

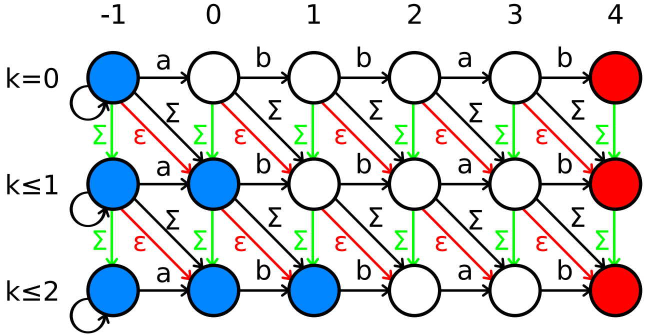
## Idea

- Start from NFA for exact pattern search
- Add $k$ additional "rows" account for up to $k$ errors
- State space: $Q = \{0, \ldots, k\} \times \{-1, \ldots, m-1\}$ for pattern $P$ with $|P| = m$.

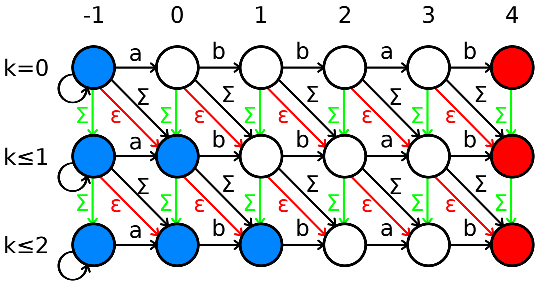
# Extension to Error Tolerant Pattern Search

## Idea

- Start from NFA for exact pattern search
- Add $k$ additional "rows" account for up to $k$ errors
- State space: $Q = \{0, \ldots, k\} \times \{-1, \ldots, m-1\}$ for pattern $P$ with $|P| = m$.



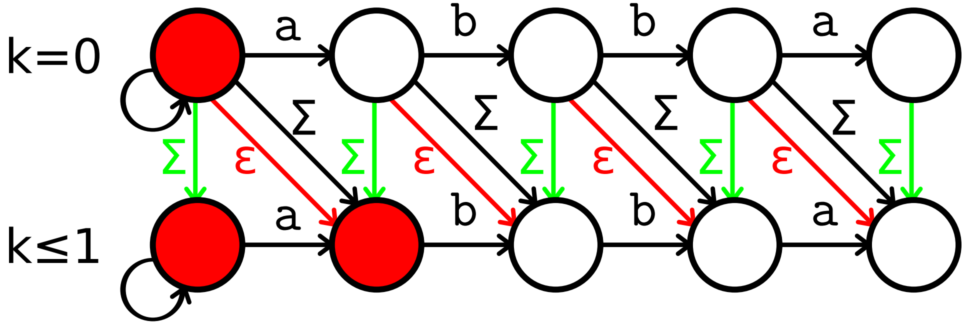
# Extension to Error Tolerant Pattern Search




- NFA for $P = \texttt{abbab}$ with edit distance up to 2 and $\Sigma = \{a, b\}$
- blue states: start states, red states: accepting states
- green vertical edges: **insertions in** $T$
- red diagonal edges: $\varepsilon$ edges for **deletions in** $P$
- black diagonal edges: $\Sigma$ edges for **mismatches in** $P$

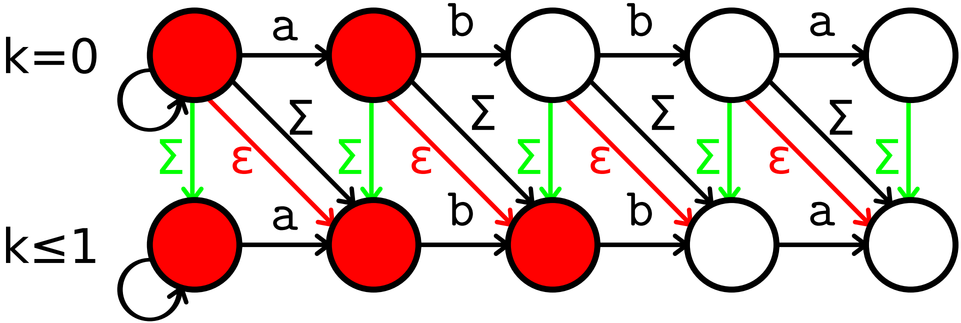
# Example

abcabba



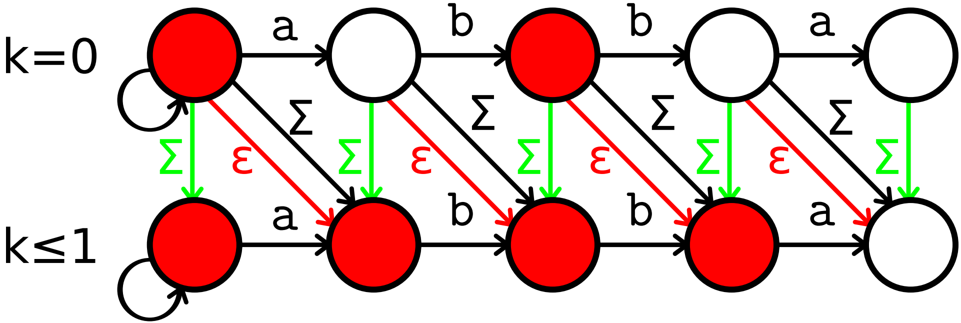
# Example

abcabba



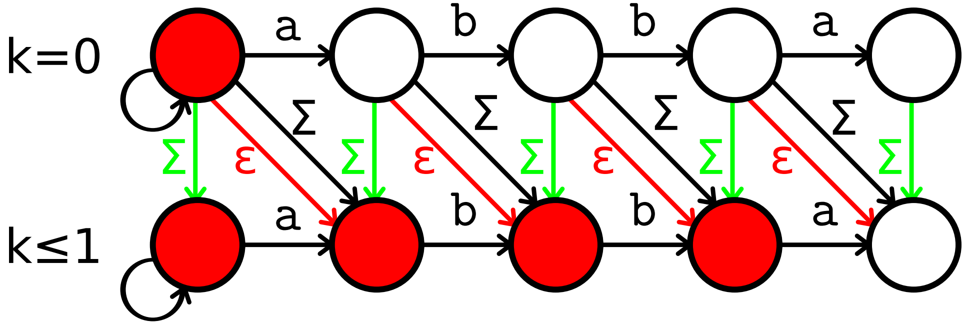
# Example

abcabba



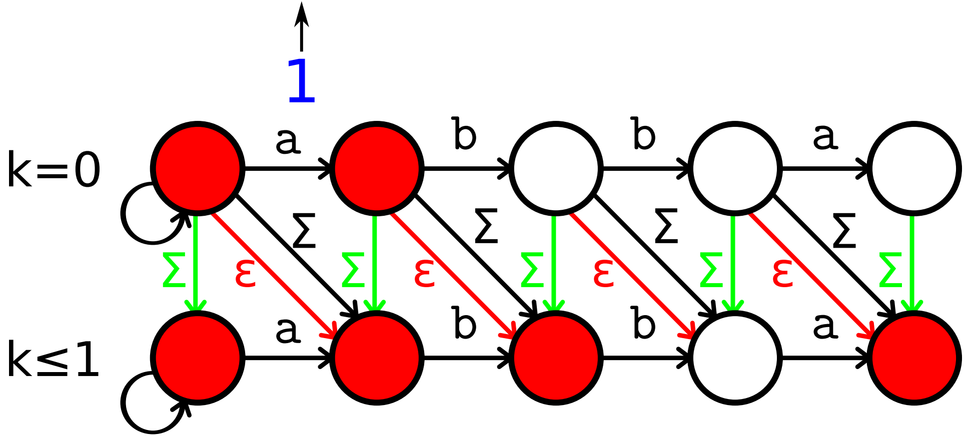
# Example

abcabba

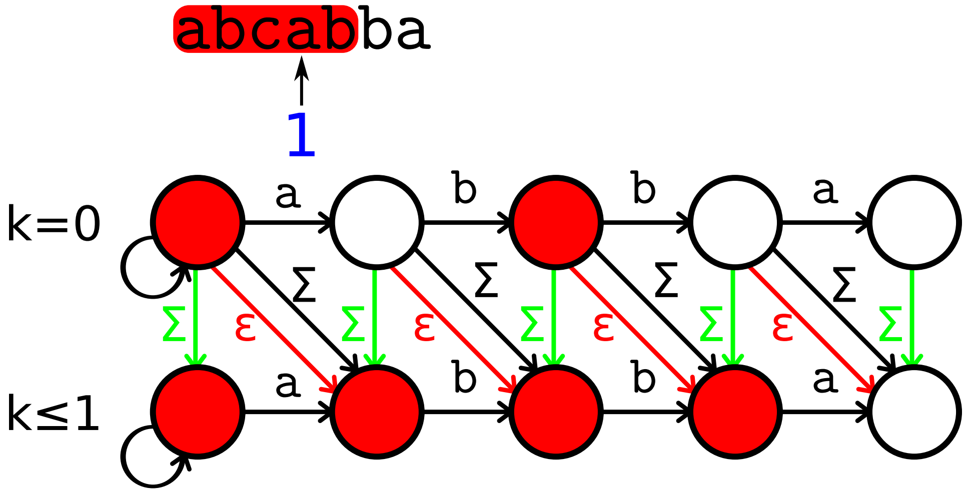


# Example

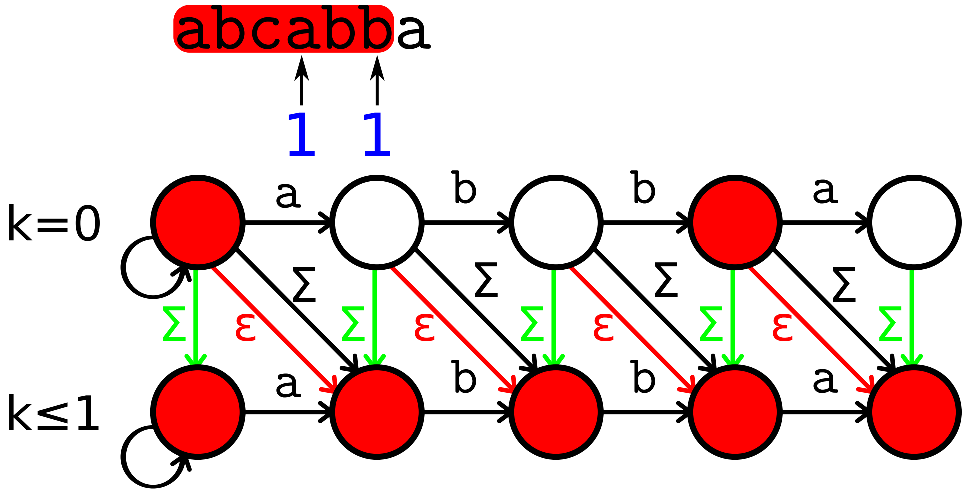
abcabba



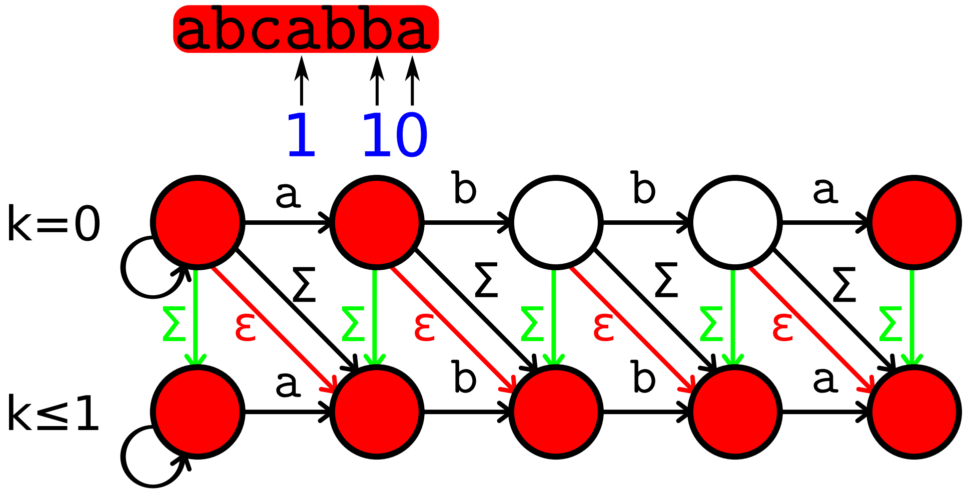
# Example



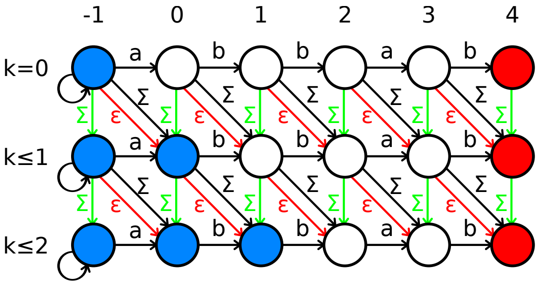
# Example



# Example



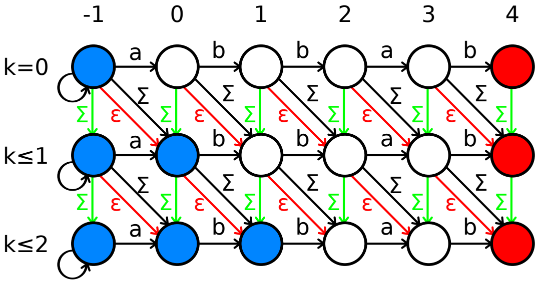
# Bit-Parallel Implementation




$$A_0 \leftarrow (A_0^{(\text{old})} \ll 1|1)\ \&\ mask[c]$$

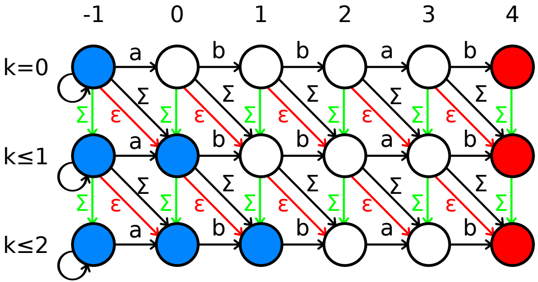
$$A_i \leftarrow ((A_i^{(\text{old})} \ll 1|1)\ \&\ mask[c])\ |\ \underbrace{(A_{i-1}^{(\text{old})})}_{\text{insertions}}\ |\ \underbrace{(A_{i-1}^{(\text{old})} \ll 1)}_{\text{mismatches}}\ |\ \underbrace{(A_{i-1} \ll 1)}_{\text{deletions}} \text{ for } 0 < i \leq k.$$

# Observations




- As usual, only practical for $|P| = m \leq 64$
- Needs a loop from 0 to $k$: only efficient for small $k$.
- Flexible: use generalized strings, gaps of bounded length, optional characters...

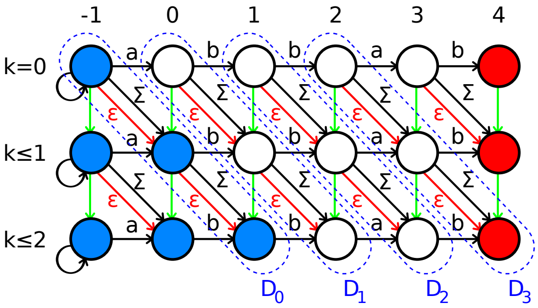
# Observations




- As usual, only practical for $|P| = m \leq 64$
- Needs a loop from 0 to $k$: only efficient for small $k$.
- Flexible: use generalized strings, gaps of bounded length, optional characters...
- One more trick: Remove loop over $k$ for small patterns, less flexible

# Diagonal Encoding

- Instead of encoding the rows of the NFA, encode the **diagonals** in **one** bit vector.
- All states in diagonals together plus separator bits must fit into 64 bits.
- Can do update in time independent of $k$ then (no loop).
- Loss of flexibility; details omitted here.



# Error Tolerant Backward Search

# Error Tolerant Backward Search

## So far

- Online algorithms, no full-text index: all have $O(n)$ time factor

## Now

- Assume that we have a full-text index, e.g., FM index.
- Achieves running times independent of $|T| = n$ for exact pattern search.
- Generalization for error tolerant pattern search?

# Error Tolerant Backward Search

## So far

- Online algorithms, no full-text index: all have $O(n)$ time factor

## Now

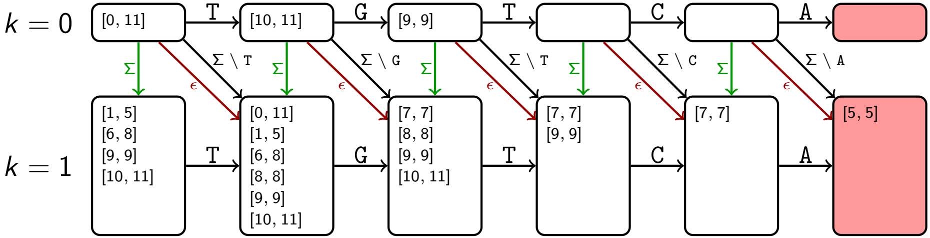
- Assume that we have a full-text index, e.g., FM index.
- Achieves running times independent of $|T| = n$ for exact pattern search.
- Generalization for error tolerant pattern search?

## Idea

- Hybrid approach between backward search and NFA.
- NFA states do not contain bits ("activity"), but sets of suffix array intervals.
- Intervals evolve along edges according to backward search.

# Example: Error Tolerant Backward Search

$T = \texttt{AAAACGTACCT\$}, \quad P = \texttt{ACTGT}, \quad \Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$:




- Green edges: insertions
- Red edges ($\varepsilon$): deletions
- Black edges: matches (horizontal) and mismatches (diagonal)

# Formal Description: Error Tolerant Backward Search

- $(k+1) \times (m+1)$ matrix $M = (M[0 \dots k, 0 \dots m])$
- $M[i,j]$: set of intervals $[L,R]$, such that
  the length-$j$ suffix of $P$ occurs with $i$ edit operations at $\mathrm{pos}[r]$ for all $r \in [L,R]$.
- $M[0,0] = \{[0..n-1]\}$

# Formal Description: Error Tolerant Backward Search

- $(k+1) \times (m+1)$ matrix $M = (M[0 \ldots k, 0 \ldots m])$
- $M[i,j]$: set of intervals $[L, R]$, such that
  the length-$j$ suffix of $P$ occurs with $i$ edit operations at $\mathrm{pos}[r]$ for all $r \in [L, R]$.
- $M[0,0] = \{[0..n-1]\}$
- For each matrix entry $M[i,j]$ and each interval $[L, R]$ within:
  1. Process matches: $[L, R]$ is updated by prepending the next letter from $P$, giving $[L^+, R^+]$ (backward search), which is added to $M[i, j+1]$ if not empty.

- Matches are found whenever an accepting state contains an interval.

# Formal Description: Error Tolerant Backward Search

- $(k+1) \times (m+1)$ matrix $M = (M[0 \ldots k,\, 0 \ldots m])$
- $M[i,j]$: set of intervals $[L,R]$, such that
  the length-$j$ suffix of $P$ occurs with $i$ edit operations at $\mathrm{pos}[r]$ for all $r \in [L,R]$.
- $M[0,0] = \{[0..n-1]\}$
- For each matrix entry $M[i,j]$ and each interval $[L,R]$ within:
  1. Process matches: $[L,R]$ is updated by prepending the next letter from $P$, giving $[L^+, R^+]$ (backward search), which is added to $M[i,j+1]$ if not empty.
  2. Process deletions: interval $[L,R]$ is copied to $M[i+1,j+1]$.

- Matches are found whenever an accepting state contains an interval.

# Formal Description: Error Tolerant Backward Search

- $(k+1) \times (m+1)$ matrix $M = (M[0 \ldots k,\, 0 \ldots m])$
- $M[i,j]$: set of intervals $[L, R]$, such that
  the length-$j$ suffix of $P$ occurs with $i$ edit operations at $\mathrm{pos}[r]$ for all $r \in [L, R]$.
- $M[0,0] = \{[0..n-1]\}$
- For each matrix entry $M[i,j]$ and each interval $[L, R]$ within:
  1. Process matches: $[L, R]$ is updated by prepending the next letter from $P$, giving $[L^+, R^+]$ (backward search), which is added to $M[i, j+1]$ if not empty.
  2. Process deletions: interval $[L, R]$ is copied to $M[i+1, j+1]$.
  3. Process insertions: For all $s \in \Sigma$, interval $[L, R]$ is updated by prepending $s$; the non-empty results $[L_s^+, R_s^+]$ are inserted into $M[i+1, j]$.

- Matches are found whenever an accepting state contains an interval.

# Formal Description: Error Tolerant Backward Search

- $(k+1) \times (m+1)$ matrix $M = (M[0 \dots k,\, 0 \dots m])$
- $M[i,j]$: set of intervals $[L, R]$, such that
  the length-$j$ suffix of $P$ occurs with $i$ edit operations at $\mathrm{pos}[r]$ for all $r \in [L, R]$.
- $M[0,0] = \{[0..n-1]\}$
- For each matrix entry $M[i,j]$ and each interval $[L, R]$ within:
  1. Process matches: $[L, R]$ is updated by prepending the next letter from $P$, giving $[L^+, R^+]$ (backward search), which is added to $M[i, j+1]$ if not empty.
  2. Process deletions: interval $[L, R]$ is copied to $M[i+1, j+1]$.
  3. Process insertions: For all $s \in \Sigma$, interval $[L, R]$ is updated by prepending $s$; the non-empty results $[L_s^+, R_s^+]$ are inserted into $M[i+1, j]$.
  4. Process substitutions: the same $[L_s^+, R_s^+]$ are also inserted into $M[i+1, j+1]$.
- Matches are found whenever an accepting state contains an interval.

# The Four Russians Method

# The Four Russians Method

**Question**

Can we reduce the time for edit distance computation to sub-quadratic?

# The Four Russians Method

**Question**

Can we reduce the time for edit distance computation to sub-quadratic?

**Answer**

No, not really and not generally.

# The Four Russians Method

### Question

Can we reduce the time for edit distance computation to sub-quadratic?

### Answer

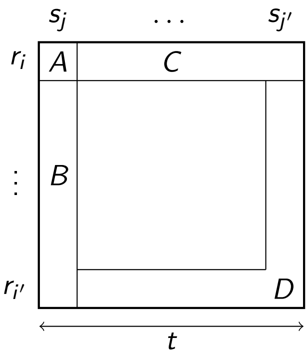
No, not really and not generally.

### However...

We can save a log factor by tabulation of all possible sub-matrices.
This is called the **Method of Four Russians** (1970),
according to its inventors Arlazarov, Dinic, Kronrod and Faradzev.

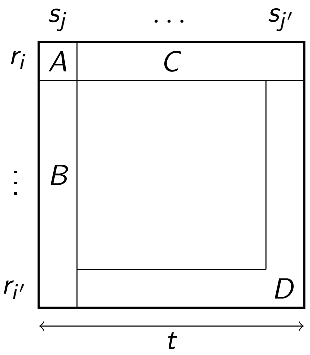
# Basic Idea

- Within a submatrix as shown the results $D$ depend only on the inputs $A, B, C$ and on the substrings $r' = r[i \dots i'],\ s' = s[j \dots j']$.
- **Definition:** A $t$-block is a $t \times t$ submatrix.



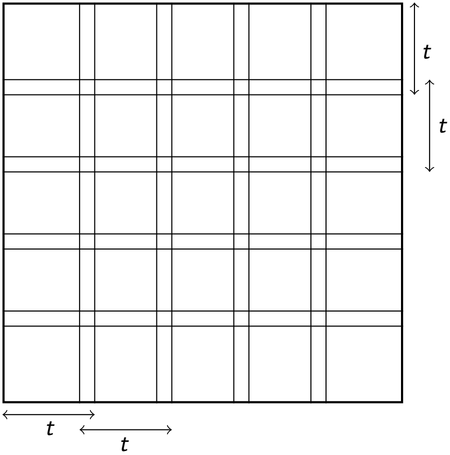
# Basic Idea

- Within a submatrix as shown the results $D$ depend only on the inputs $A, B, C$ and on the substrings $r' = r[i \ldots i'],\ s' = s[j \ldots j']$.
- **Definition:** A $t$-block is a $t \times t$ submatrix.
- **Idea:** Subdivide matrix into $t$-blocks. Pre-compute results ($D$) for all combinations of $(A, B, C, r', s')$.
- Avoid redundancies.



# Basic Idea

Subdivide matrix into overlapping $t$-blocks: aufteilen:



# Reminder: Matrix Properties

Let $T$ be a DP matrix satisfying the edit distance recurrence.

**Lemma: Vertical Property**

The value difference between any two **vertically adjacent** cells is at most 1:
$|T[i,j] - T[i-1,j]| \leq 1$.

**Lemma: Horizontal Property**

The value difference between any two **horizontally adjacent** cells is at most 1:
$|T[i,j] - T[i,j-1]| \leq 1$.

**Lemma: Diagonal Property**

The value of **diagonally adjacent** cells is non-decreasing,
and the value difference is at most 1, i.e., $0 \leq T[i,j] - T[i-1,j-1] \leq 1$.

# Observation

If the input values (areas $A, B, C$) in two $t$-blocks differ by one offset,
**and** if the substrings are identical,
then the output values (area $D$) differ by the same offset.

| | a | b | b | a |
|---|---|---|---|---|
| b | 5 | 6 | 5 | 4 |
| a | 6 | 6 | 6 | 5 |
| b | 6 | 6 | 6 | 6 |
| a | 7 | 7 | 7 | 6 |

| | a | b | b | a |
|---|---|---|---|---|
| b | 2 | 3 | 2 | 1 |
| a | 3 | 3 | 3 | 2 |
| b | 3 | 3 | 3 | 3 |
| a | 4 | 4 | 4 | 3 |

# Preprocessing

To avoid pre-computing $t$-blocks for (infinitely) many combinations of $A, B, C$, we consider $A$ as an offset, and difference vectors $\delta_B$, $\delta_C$:
Let $\delta_B[0] := 0$, and $\delta_B[i] := B[i] - B[i-1]$.
Let $\delta_C[0] := 0$, and $\delta_C[j] := C[j] - C[j-1]$

| | a | b | b | a |
|---|---|---|---|---|
| b | 5 | 6 | 5 | 4 |
| a | 6 | | | |
| b | 6 | | | |
| a | 7 | | | |

$\rightarrow$

| | a | b | b | a |
|---|---|---|---|---|
| b | 0 | 1 | -1 | -1 |
| a | 1 | | | |
| b | 0 | | | |
| a | 1 | | | |

$$B = [5, 6, 6, 7] \quad \rightarrow \quad \delta_B = [(0, )1, 0, 1]$$

$$C = [5, 6, 5, 4] \quad \rightarrow \quad \delta_C = [(0, )1, -1, -1]$$

# Preprocessing

## Running time for pre-computing all blocks

- Because $\delta_B[0] = \delta_C[0] = 0$ and the other $\delta$-values are limited to $\{-1, 0, 1\}$, there are at most $3^{2(t-1)}$ combinations for $(\delta_B, \delta_C)$.

# Preprocessing

## Running time for pre-computing all blocks

- Because $\delta_B[0] = \delta_C[0] = 0$ and the other $\delta$-values are limited to $\{-1, 0, 1\}$, there are at most $3^{2(t-1)}$ combinations for $(\delta_B, \delta_C)$.
- There are $\sigma^{2(t-1)}$ different substring combinations.
- Pre-computing a $t$-block takes $O((t-1)^2)$ time.

# Preprocessing

## Running time for pre-computing all blocks

- Because $\delta_B[0] = \delta_C[0] = 0$ and the other $\delta$-values are limited to $\{-1, 0, 1\}$, there are at most $3^{2(t-1)}$ combinations for $(\delta_B, \delta_C)$.
- There are $\sigma^{2(t-1)}$ different substring combinations.
- Pre-computing a $t$-block takes $O((t-1)^2)$ time.
- Total time for all blocks: $O\left(3^{2(t-1)}\sigma^{2(t-1)}(t-1)^2\right)$.

# Preprocessing

## Running time for pre-computing all blocks

- Because $\delta_B[0] = \delta_C[0] = 0$ and the other $\delta$-values are limited to $\{-1, 0, 1\}$, there are at most $3^{2(t-1)}$ combinations for $(\delta_B, \delta_C)$.
- There are $\sigma^{2(t-1)}$ different substring combinations.
- Pre-computing a $t$-block takes $O((t-1)^2)$ time.
- Total time for all blocks: $O\big(3^{2(t-1)}\sigma^{2(t-1)}(t-1)^2\big)$.
- Choose $t := 1 + (\log_{3\sigma} n)/2$:
- Time becomes $O\big(n \cdot (\log_{3\sigma} n)^2\big)$

# Preprocessing

## Running time for pre-computing all blocks

- Because $\delta_B[0] = \delta_C[0] = 0$ and the other $\delta$-values are limited to $\{-1, 0, 1\}$, there are at most $3^{2(t-1)}$ combinations for $(\delta_B, \delta_C)$.
- There are $\sigma^{2(t-1)}$ different substring combinations.
- Pre-computing a $t$-block takes $O((t-1)^2)$ time.
- Total time for all blocks: $O\big(3^{2(t-1)}\sigma^{2(t-1)}(t-1)^2\big)$.
- Choose $t := 1 + (\log_{3\sigma} n)/2$:
- Time becomes $O\big(n \cdot (\log_{3\sigma} n)^2\big)$

## Memory requirements

- To store the result ($D$ region) for one block: $O(t)$ bits (difference-encoded)
- Total: $O(n \log n)$ bits, or $n$ integers.

# Computation

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | | | | | | | |
| b | | | | | | | |
| b | | | | | | | |
| a | | | | | | | |
| a | | | | | | | |
| b | | | | | | | |
| a | | | | | | | |

# Computation

1. Initialize row and column 0

$\delta_B =$
$\delta_C =$
$A =$
$D =$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | | | | |
| b | 2 | | | | | | |
| a | 3 | | | | | | |
| a | 4 | | | | | | |
| b | 5 | | | | | | |
| a | 6 | | | | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:

$\delta_B =$

$\delta_C =$

$A = 0$

$D =$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | | | | |
| b | 2 | | | | | | |
| a | 3 | | | | | | |
| a | 4 | | | | | | |
| b | 5 | | | | | | |
| a | 6 | | | | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$
   (or re-use old difference-encoded $D$)

$\delta_B = 0, 1, 1, 1$
$\delta_C = 0, 1, 1, 1$
$A = 0$
$D =$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | | | | |
| b | 2 | | | | | | |
| a | 3 | | | | | | |
| a | 4 | | | | | | |
| b | 5 | | | | | | |
| a | 6 | | | | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$
   (or re-use old difference-encoded $D$)
4. Lookup:
   $D = F[\delta_B, \delta_C, r[i' : i''], s[j' : j'']]$.

$\delta_B = 0, 1, 1, 1$
$\delta_C = 0, 1, 1, 1$
$A = 0$
$D = 2, 2, 2, 1, 2$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | | | | |
| b | 2 | | | | | | |
| a | 3 | | | | | | |
| a | 4 | | | | | | |
| b | 5 | | | | | | |
| a | 6 | | | | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$ (or re-use old difference-encoded $D$)
4. Lookup: $D = F[\delta_B, \delta_C, r[i' : i''], s[j' : j'']]$.
5. Keep track of offset $A$.

$$\delta_B = 0, 1, 1, 1$$
$$\delta_C = 0, 1, 1, 1$$
$$A = 0$$
$$D = 2, 2, 2, 1, 2$$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | 2 | | | |
| b | 2 | | | 1 | | | |
| a | 3 | 2 | 2 | 2 | | | |
| a | 4 | | | | | | |
| b | 5 | | | | | | |
| a | 6 | | | | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$ (or re-use old difference-encoded $D$)
4. Lookup: $D = F[\delta_B, \delta_C, r[i' : i''], s[j' : j'']]$.
5. Keep track of offset $A$.

$\delta_B = 0, 1, 1, 1$
$\delta_C = 0, -1, 0, 0$
$A = 3$
$D = 2, 1, 1, 0, 0$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | 2 | | | |
| b | 2 | | | 1 | | | |
| a | 3 | 2 | 2 | 2 | | | |
| a | 4 | | | 3 | | | |
| b | 5 | | | 3 | | | |
| a | 6 | 5 | 4 | 4 | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$ (or re-use old difference-encoded $D$)
4. Lookup: $D = F[\delta_B, \delta_C, r[i' : i''], s[j' : j'']]$.
5. Keep track of offset $A$.

$\delta_B = 0, -1, -1, 1$
$\delta_C = 0, 1, 1, 1$
$A = 3$
$D = -2, -1, 0, 1, 2$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | 2 | | | 5 |
| b | 2 | | | 1 | | | 4 |
| a | 3 | 2 | 2 | 2 | 1 | 2 | 3 |
| a | 4 | | | 3 | | | |
| b | 5 | | | 3 | | | |
| a | 6 | 5 | 4 | 4 | | | |

# Computation

1. Initialize row and column 0
2. For all $i < m/t$ and $j < n/t$:
3. Compute $\delta_B, \delta_C$ from $B, C$
   (or re-use old difference-encoded $D$)
4. Lookup:
   $D = F[\delta_B, \delta_C, r[i' : i''], s[j' : j'']]$.
5. Keep track of offset $A$.

$\delta_B = 0, 1, 0, 1$
$\delta_C = 0, -1, 1, 1$
$A = 2$
$D = 1, 1, 0, 1, 0$

| | - | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | | | 2 | | | 5 |
| b | 2 | | | 1 | | | 4 |
| a | 3 | 2 | 2 | 2 | 1 | 2 | 3 |
| a | 4 | | | 3 | | | 2 |
| b | 5 | | | 3 | | | 3 |
| a | 6 | 5 | 4 | 4 | 3 | 3 | 2 |

# Running Time for Error Tolerant Pattern Search

## Time for one $t$-block

- Compute differences: $O(t)$
- Look-up $F[q]$ in big table: $O(t)$ for computing index $q$
- Keeping track of offset: $O(t)$
- Total time for each block: $O(t)$

# Running Time for Error Tolerant Pattern Search

## Time for one $t$-block

- Compute differences: $O(t)$
- Look-up $F[q]$ in big table: $O(t)$ for computing index $q$
- Keeping track of offset: $O(t)$
- Total time for each block: $O(t)$

## Total time

- Number of $t$-blocks: $mn/t^2$

# Running Time for Error Tolerant Pattern Search

## Time for one $t$-block

- Compute differences: $O(t)$
- Look-up $F[q]$ in big table: $O(t)$ for computing index $q$
- Keeping track of offset: $O(t)$
- Total time for each block: $O(t)$

## Total time

- Number of $t$-blocks: $mn/t^2$
- Total time: $O(t \cdot nm/t^2) = O(nm/t) = O(nm/\log n)$

# Summary: Four Russians Method

## Running Times

- With the Four-Russians trick (difference coding, pre-computation of small blocks), one can compute the edit distance or do pattern search in sub-quadratic time.
- Pre-computation (all possible blocks): $O\left(n\,(\log n)^2\right)$ time
- Computation for two strings: $O(nm/\log n)$ time

# Summary: Four Russians Method

## Running Times

- With the Four-Russians trick (difference coding, pre-computation of small blocks), one can compute the edit distance or do pattern search in sub-quadratic time.
- Pre-computation (all possible blocks): $O\big(n\,(\log n)^2\big)$ time
- Computation for two strings: $O(nm/\log n)$ time

## Practicality

- Because of the high base in the logarithm $(t := 1 + (\log_{3\sigma} n)/2)$, the method is only practical for large $n \geq 10\,000$, especially for small alphabets (DNA: $\sigma = 4$).
- For larger alphabets, much memory is needed.
- Therefore, the Four Russians Method is rarely used in practice.

# Comparison of Running Times

| algorithm | time | advantages | disadvantages |
|---|---|---|---|
| Basic | $O(mn)$ | simple | slow |
| Ukkonen | $O(kn)$ expected | simple | |
| Myers | $O((m/w)n)$ | fast for high $k$ | unintuitive |
| NFA | $O(k(m/w)n)$ | fast for small $m$ or $k$ | slow for large $k$ |
| 4 Russians | $O(mn/\log n)$ | nice idea | only faster for large $n$ |
| NFA-FM | (*) | independent of $n$ | exponential in $\|\Sigma\|$, $m$, $k$ |

(*) NFA-FM time can be shown to be $O(\sqrt{k}\,(1+\sqrt{2})^{2k}\,3^{m-k}|\Sigma|^k)$ for $k \leq m$.

# Comparison of Running Times

| algorithm | time | advantages | disadvantages |
|---|---|---|---|
| Basic | $O(mn)$ | simple | slow |
| Ukkonen | $O(kn)$ expected | simple | |
| Myers | $O((m/w)n)$ | fast for high $k$ | unintuitive |
| NFA | $O(k(m/w)n)$ | fast for small $m$ or $k$ | slow for large $k$ |
| 4 Russians | $O(mn/\log n)$ | nice idea | only faster for large $n$ |
| NFA-FM | (*) | independent of $n$ | exponential in $\lvert\Sigma\rvert$, $m$, $k$ |

(*) NFA-FM time can be shown to be $O(\sqrt{k}\,(1+\sqrt{2})^{2k}\,3^{m-k}|\Sigma|^k)$ for $k \leq m$.

## Notes

- $w$ ist the register size, typically 64 bits.
- Alignments can only be easily derived from the Basic and Ukkonen algorithms.

# Summary

### Today

- NFA for error tolerant pattern matching
- error tolerant pattern matching with FM index (via interval NFA)
- Four Russians' method: tabulation of small submatrices
- Comparison of algorithms

# Possible Exam Questions

- Explain how the Shift-And algorithm can be adjusted to solve the approximate pattern matching problem.
- Explain the semantics of the states in the corresponding NFA.
- Explain the meaning of the different types of edges.
- How many states are always active for a NFA that allows $k$ mismatches?
- How exactly does the bit-parallel update of the active state matrix $A$ work?
- How can backward search be applied to error tolerant search?
- Explain the idea of the Four Russians Technique.
- Why is the block size chosen as $t := 1 + (\log_{3\sigma} n)/2$ in the Four Russians Method?