ZyLAB®

eDiscovery & Intelligent Information Governance

Algorithms for approximate string matching

Petro Protsyk (28 October 2016)

Agenda

- About ZyLAB
- Overview
- Algorithms
  - Brute-force recursive Algorithm
  - Wagner and Fischer Algorithm
  - Algorithms based on Automaton
  - Bitap
- Q&A

About ZyLAB

ZyLAB is a software company headquartered in Amsterdam.

In 1983 ZyLAB was the first company providing a **full-text** search program for MS-DOS called ZyINDEX

In 1991 ZyLAB released ZyIMAGE software bundle that included **a fuzzy string search** algorithm to overcome scanning and OCR errors.

Currently ZyLAB develops eDiscovery and Information Governance solutions for On Premises, SaaS and Cloud installations. ZyLAB continues to develop new versions of its proprietary full-text search engine.

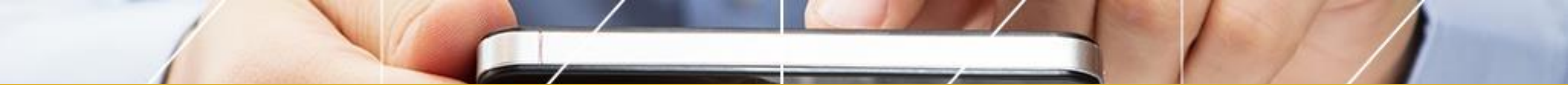
About ZyLAB

- We develop for Windows environments only, including Azure
- Software developed in C++, .Net, SQL
- Typical clients are from law enforcement, intelligence, fraud investigators, law firms, regulators etc.
  - FTC (Federal Trade Commission), est. >50 TB (peak 2TB per day)
  - US White House (all email of presidential administration), est. >20 TB
  - Dutch National Police, est. >20 TB

ZyLAB Search Engine

Full-text search engine

- Boolean and Proximity search operators
- Support for Wildcards, Regular Expressions and Fuzzy search
- Support for numeric and date searches
- Search in document fields
- Near duplicate search



ZyLAB Search Engine

- Highly parallel indexing and searching
- Can index Terabytes of data and millions of documents
- Can be distributed
- Supports 100+ languages, Unicode characters

Overview

Approximate string matching or **fuzzy search** is the technique of finding strings in text or dictionary that match given pattern approximately with respect to chosen **edit distance**.

The **edit distance** is the number of primitive operations necessary to convert the string into an exact match.

The usual primitive operations are: **insertion**, **deletion**, **substitution**, **transposition**.

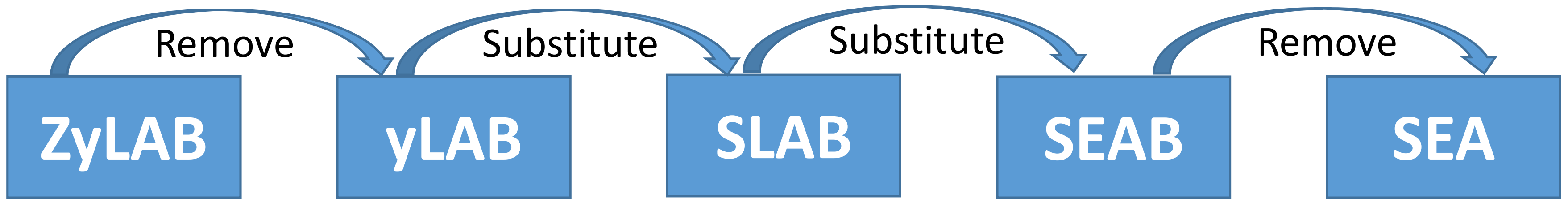
All operations have the same cost.

Overview

Levenshtein distance between two words is the minimum number of single-character insertions, deletions or substitutions required to change one word into the other.

Named after Vladimir Levenshtein, who described this distance in 1965.

Example: Levenshtein distance between "ZyLAB" and "SEA" is 4:

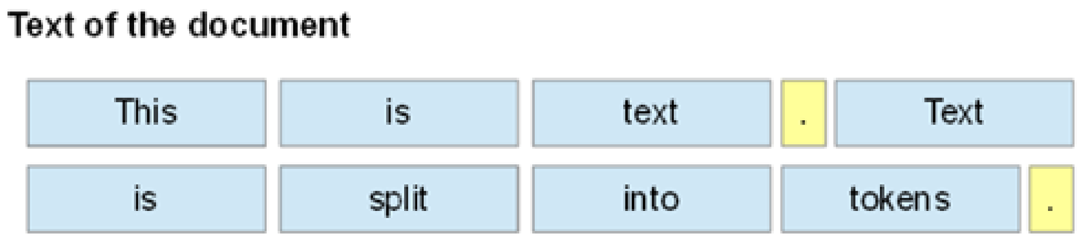


ZyLAB®

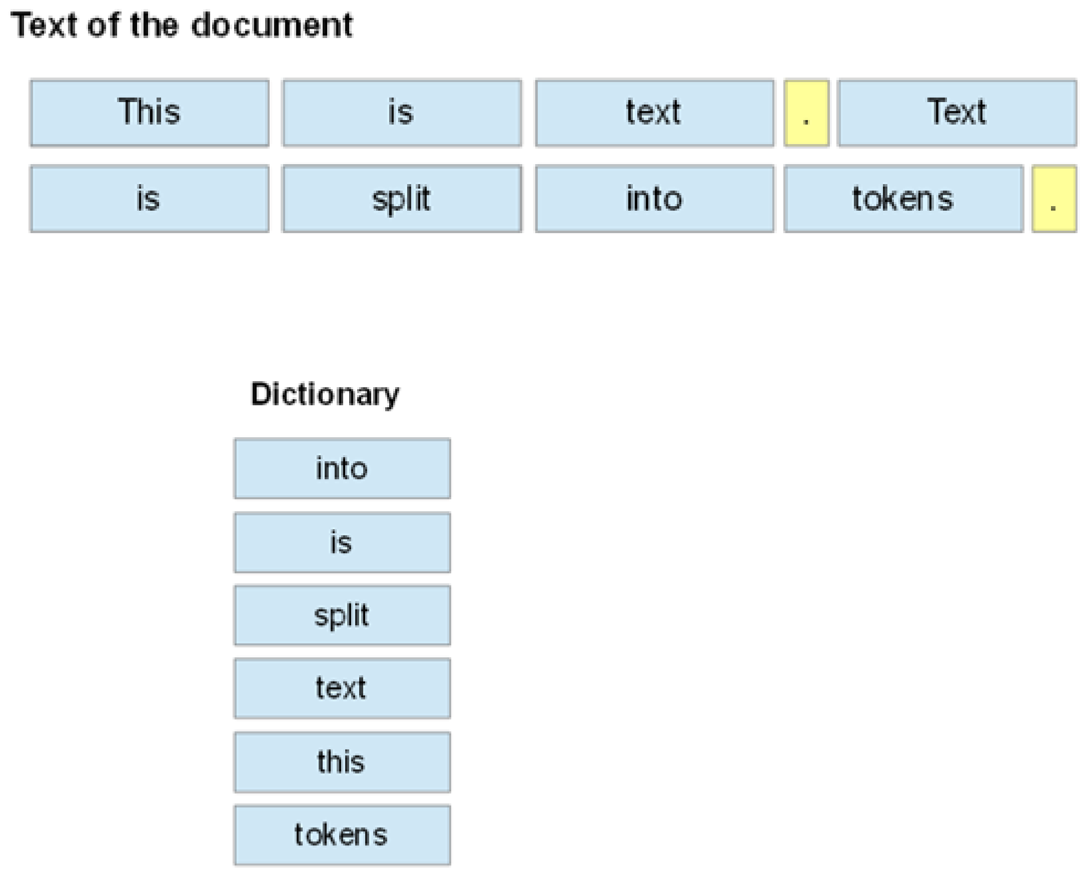
Overview

Applications: The most common applications of approximate string matching are spell checking, syntax error correction, spam filtering, correction of OCR errors, full-text search.

Full-text index

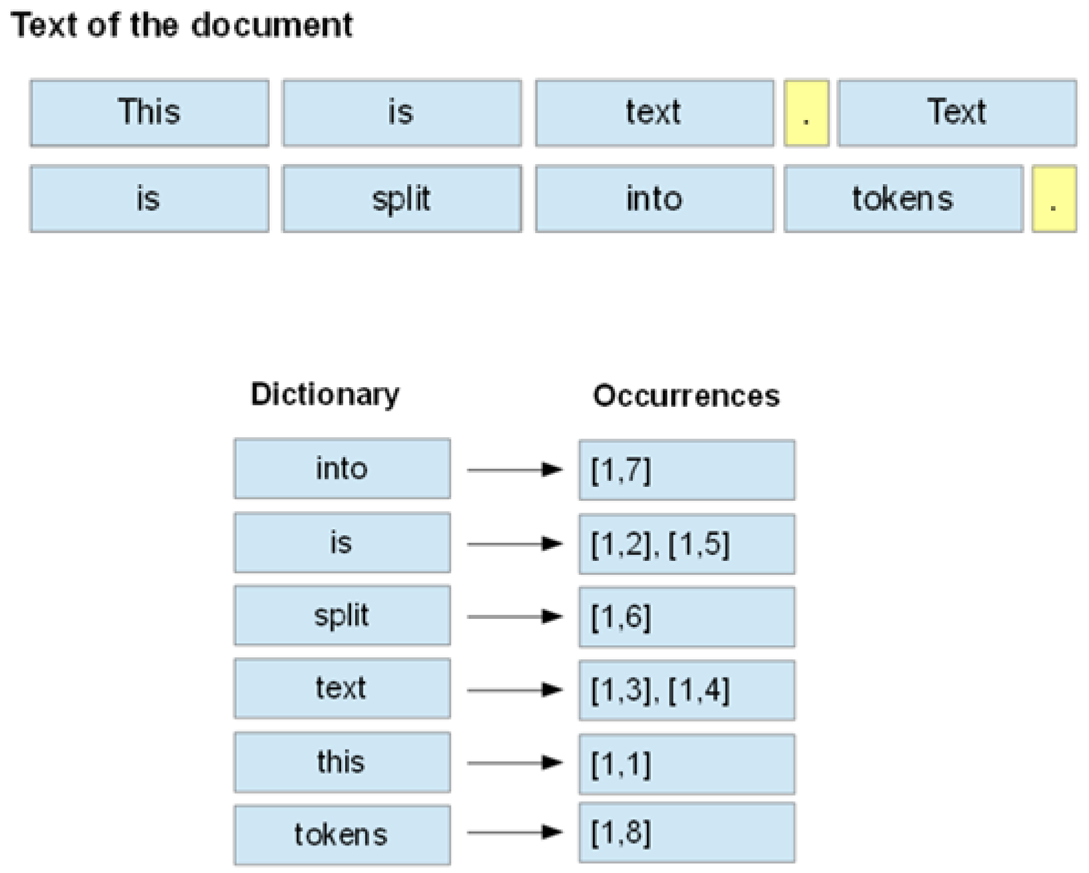


Full-text index

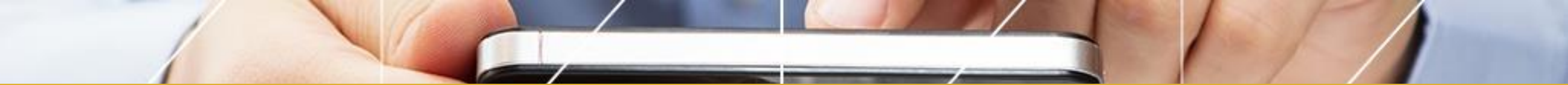


ZyLAB®

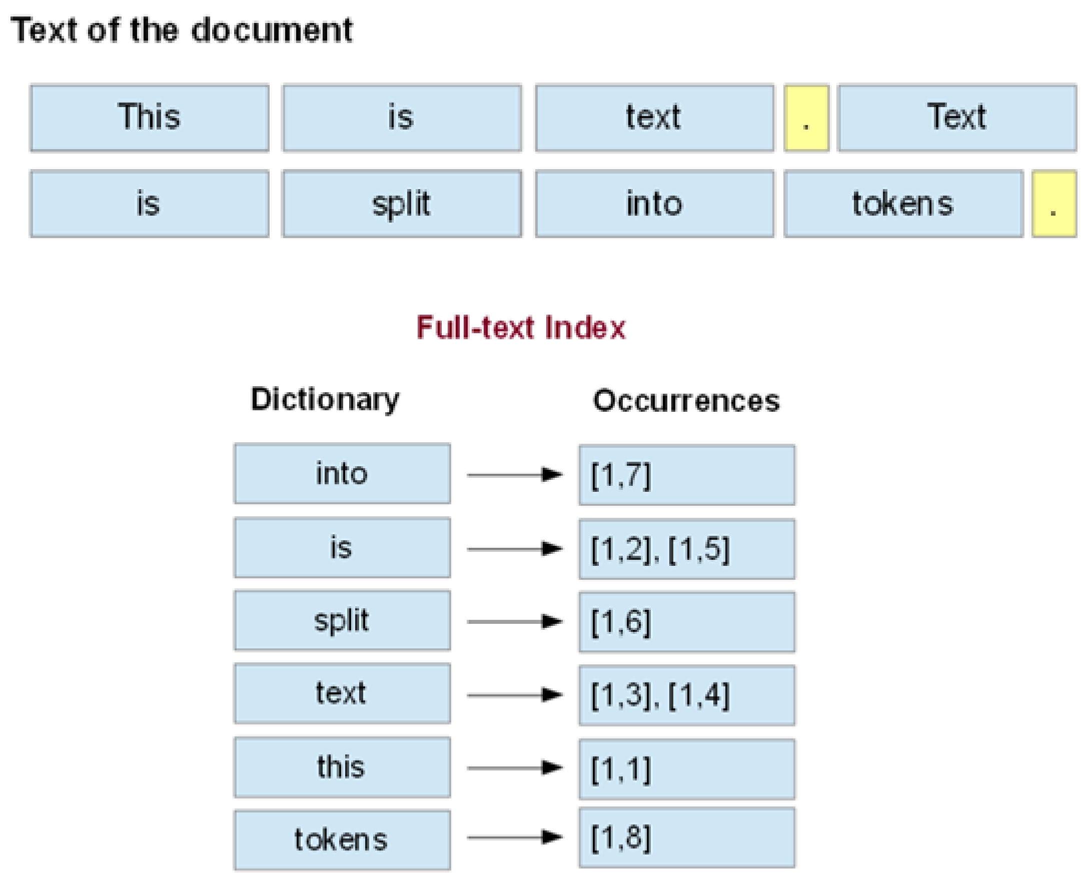
Full-text index



ZyLAB®



Full-text index



Dictionary	Occurrences
into	[1,7]
is	[1,2], [1,5]
split	[1,6]
text	[1,3], [1,4]
this	[1,1]
tokens	[1,8]

ZyLAB®

Fuzzy query in ZyLAB search engine:

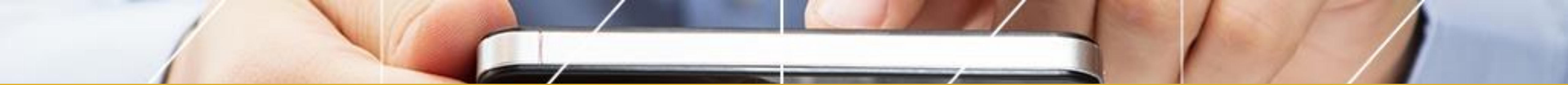
- All words with Levenshtein distance 2 or less: **word~2**
- All words with Levenshtein distance 2: **word~2 – {word~1}**

Overview

Problem: Find terms in the dictionary that match the pattern approximately.

environment~2:

environment
environment**s**
env**e**ronment
environment**al**



Algorithms

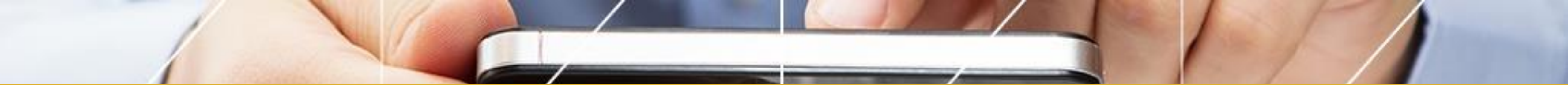
Brute-force search
(recursive, 1965)

Wagner and Fischer
(Dynamic Programming, 1968-1974)

Levenshtein Automaton
(Automata theory, 1992-2002)

Bitap algoritm
(1964-1992)

ZyLAB®



Algorithms

Mathematically, the Levenshtein distance between two strings a,b is given by the following formula ***lev*$_{a,b}$*(|a|, |b|)***:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

lev$_{a,b}$(i,j) is the distance between the first *i* characters of *a*, and the first *j* characters of *b*. $1_{(a_i \neq b_j)}$ is equal to 1 when a_i=b_j and 0 otherwise.

Algorithms

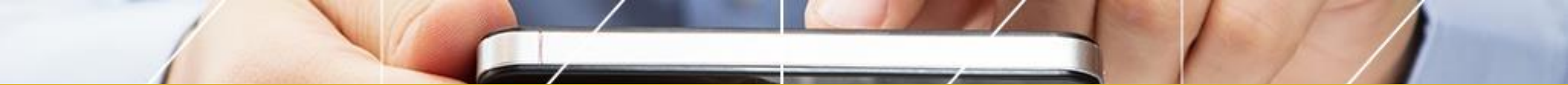
$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \boxed{\operatorname{lev}_{a,b}(i-1,j) + 1} \\ \boxed{\operatorname{lev}_{a,b}(i,j-1) + 1} \\ \boxed{\operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)}} \end{cases} & \text{otherwise.} \end{cases}$$

red – deletion of character
blue – insertion of character
green – substitution or match

Brute-force search

Brute-force search

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$



Brute-force search

```
private static int CalculateRecursive(string a, string b, int m, int n)
{
    if (Math.Min(m, n) == 0)
    {
        return Math.Max(m, n);
    }

    var subCost = ((a[m - 1] == b[n - 1]) ? 0 : 1) + CalculateRecursive(a, b, m - 1, n - 1);
    var delCost = 1 + CalculateRecursive(a, b, m, n - 1);
    var insCost = 1 + CalculateRecursive(a, b, m - 1, n);

    return Math.Min(subCost, Math.Min(insCost, delCost));
}
```

Brute-force search

1) Easy to implement using $\boldsymbol{lev_{a,b}(|a|, |b|)}$ formula
2) Exponential complexity of $|a|+|b|$
3) Not used in commercial applications

Demo

Wagner and Fischer Algorithm

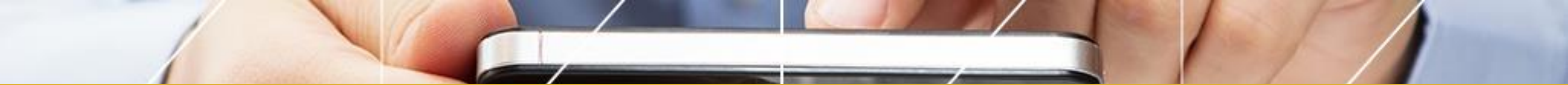
Wagner and Fischer Algorithm

Idea of algorithm

Store Levenshtein distances between all prefixes of the first string and all prefixes of the second in the matrix.

To compute next cell, only values of three other neighbor cells are needed:

	prefix(b, |b|-1)	prefix(b, |b|)
prefix(a, |a|-1)	*lev(|a|-1, |b|-1)*	*lev(|a|-1, |b|)*
prefix(a, |a|)	*lev(|a|, |b|-1)*	lev(|a|, |b|)



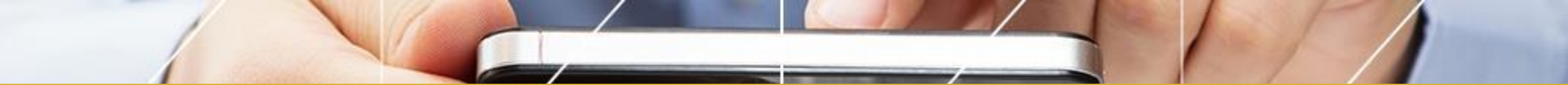
Wagner and Fischer Algorithm

car $\Longrightarrow$ **cdar**

Wagner and Fischer Algorithm

car $\longrightarrow$ **cdar**

					c
				c	d
			c	d	a
	_	c	d	a	r
_	0	1	2	3	4



Wagner and Fischer Algorithm

car $\Rightarrow$ **cdar**

					c
				c	d
			c	d	a
	_	c	d	a	r
_	0	1	2	3	4
c	1	0	1	2	3

Wagner and Fischer Algorithm

car $\Longrightarrow$ **cdar**

						c
					c	d
				c	d	a
		_	c	d	a	r
	_	0	1	2	3	4
	c	1	0	1	2	3
c	a	2				

Wagner and Fischer Algorithm

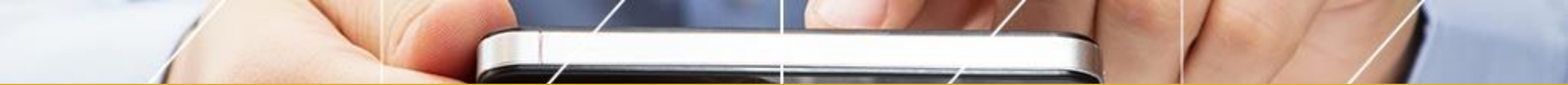
car $\Longrightarrow$ **cdar**

						c
					c	d
				c	d	a
		_	c	d	a	r
	_	0	1	2	3	4
	c	1	0	1	2	3
c	a	2	1			

Wagner and Fischer Algorithm

car ⇨ **cdar**

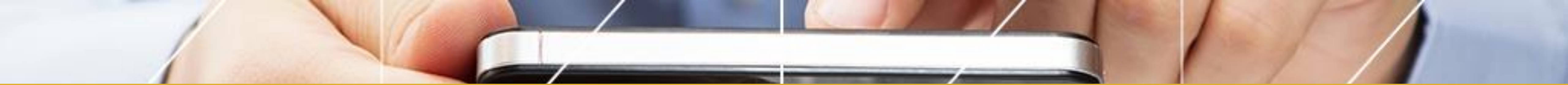
						c
					c	d
				c	d	a
		_	c	d	a	r
	_	0	1	2	3	4
	c	1	0	1	2	3
c	a	2	1	1		



Wagner and Fischer Algorithm

car $\Rightarrow$ **cdar**

						c
					c	d
				c	d	a
		_	c	d	a	r
	_	0	1	2	3	4
	c	1	0	1	2	3
c	a	2	1	1	1	



Wagner and Fischer Algorithm

car $\Longrightarrow$ **cdar**

						c
					c	d
				c	d	a
		_	c	d	a	r
	_	0	1	2	3	4
	c	1	0	1	2	3
c	a	2	1	1	1	2

Wagner and Fischer Algorithm

car $\Longrightarrow$ **cdar**

							c
						c	d
					c	d	a
			_	c	d	a	r
		_	0	1	2	3	4
		c	1	0	1	2	3
	c	a	2	1	1	1	2
c	a	r	3	2	2	2	1

Wagner and Fischer Algorithm

car ⟹ **cdar**

							c
						c	d
					c	d	a
			_	c	d	a	r
		_	0	1	2	3	4
		c	1	0	1	2	3
	c	a	2	1	1	1	2
c	a	r	3	2	2	2	1

Wagner and Fischer Algorithm

```
public static int CalculateDynamic(string a, string b)
{
    var d = new int[a.Length + 1, b.Length + 1];

    for (int i = 0; i <= a.Length; i++)
        d[i, 0] = i;
    for (int i = 0; i <= b.Length; i++)
        d[0, i] = i;

    for (int i = 0; i < a.Length; i++) {
        for (int j = 0; j < b.Length; j++) {
            if (a[i] == b[j]) {
                d[i + 1, j + 1] = d[i, j];
            } else {
                d[i + 1, j + 1] = 1 + Math.Min(d[i, j], Math.Min(d[i, j + 1], d[i + 1, j]));
            }
        }
    }

    return d[a.Length, b.Length];
}
```

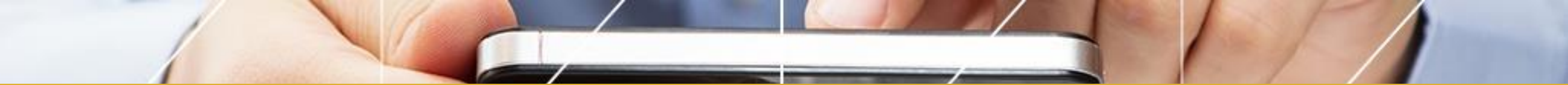
Wagner and Fischer Algorithm

R Wagner, M Fischer The String-to-String Correction Problem (1974)

1) Easy to implement using $\boldsymbol{lev_{a,b}(|a|, |b|)}$ formula
2) Require ***O(|a|*|b|)*** steps, does not depend on input characters
3) Require ***O(min(|a|, |b|))*** memory
4) Matching dictionary requires ~$\boldsymbol{|a| * \sum |t_i|}$ steps

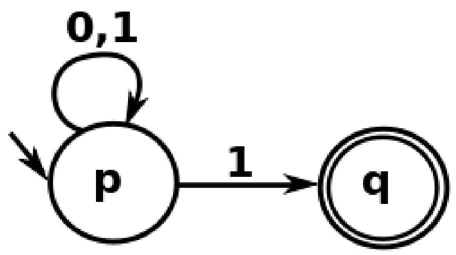
Demo

Levenshtein Automaton

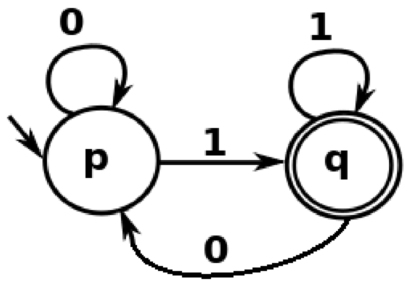


Levenshtein Automaton

Nondeterministic finite automaton (NFA). States, alphabet, transitions, set of final states, initial state. For each symbol and state there can multiple transitions.



Deterministic finite automaton (DFA). States, alphabet, transitions, set of final states, initial state. For each symbol and state there can be only one transition.

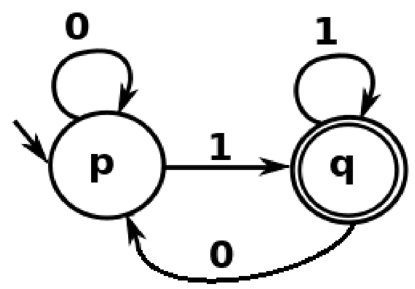


Levenshtein Automaton

Automata can be used to recognize patterns in text

1) Begin from initial state
2) Follow transitions for each character in text
3) If resulting state is the final state – it is a match.

Example: 0 1 0 0 1



Levenshtein Automaton

Matching dictionary with DFA is easy:

Difficult part

```
var dfa = CreateAutomaton(pattern, d);

foreach (var term in dictionary)
{
    var s = 0;
    foreach(var c in term)
    {
        s = dfa.Next(s, c);
    }
    if( dfa.IsFinal(s))
    {
        yield return term;
    }
}
```

ZyLAB®

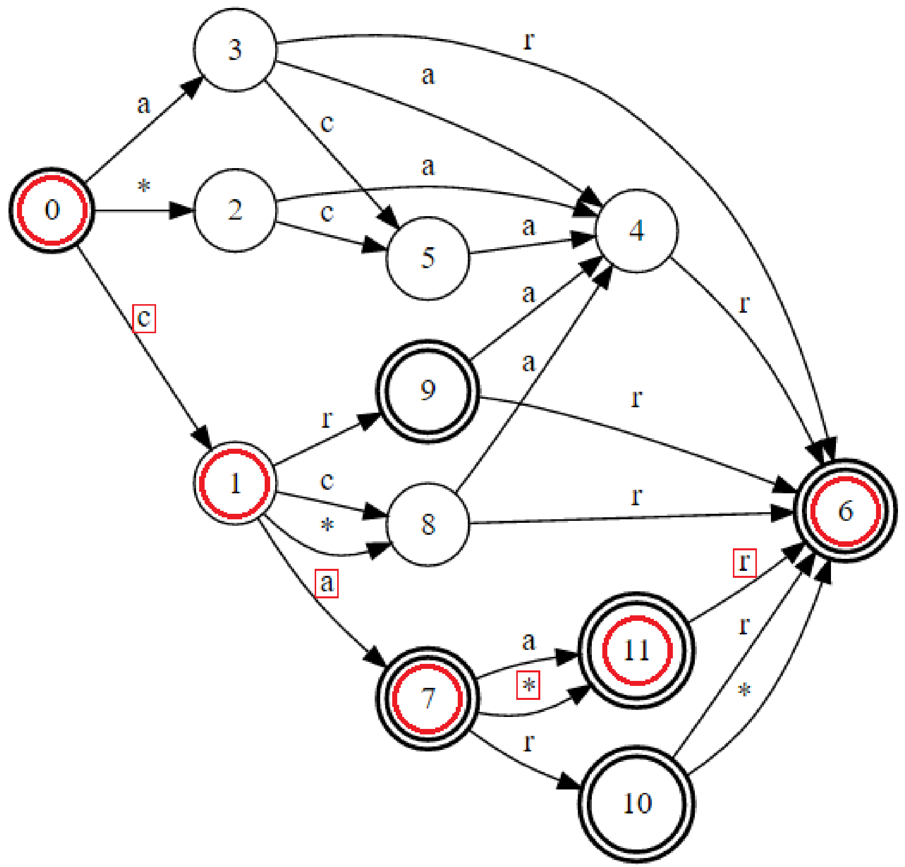
Levenshtein Automaton

Levenshtein automaton for a string **w** and a number **d** is a finite state automaton that can recognize the set of all strings whose Levenshtein distance from **w** is at most **d**

Levenshtein Automaton

Automaton that recognizes: **car~1**

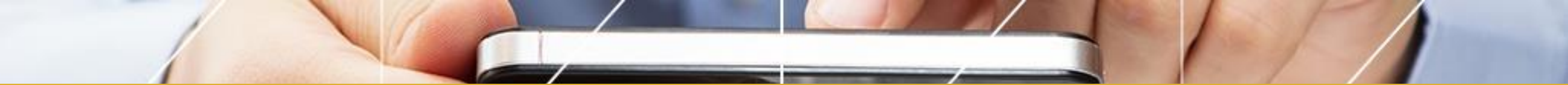
Matching cadr



A match require at most |b| steps

Levenshtein Automaton

How to create Levenshtein Automaton for word **w** and a number **d**?



Construction of NFA

Automaton to recognize **car**

$$\begin{tikzcd} 0 \end{tikzcd}$$

$$\begin{tikzcd} 0 \arrow[r, "c"] & 1 \end{tikzcd}$$

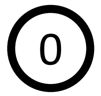
$$\begin{tikzcd} 0 \arrow[r, "c"] & 1 \arrow[r, "a"] & 2 \end{tikzcd}$$

$$\begin{tikzcd} 0 \arrow[r, "c"] & 1 \arrow[r, "a"] & 2 \arrow[r, "r"] & \boxed{3} \end{tikzcd}$$

ZyLAB®

Construction of NFA

Automaton to recognize **car~1**



Construction of NFA

Automaton to recognize **car~1**

$$\begin{tikzcd}
0 \arrow[r, "c"] \arrow[d, "*"] \arrow[dr, "*"] \arrow[dr, bend right=15, "\varepsilon"'] & 1 \arrow[d, "*"] \\
4 \arrow[r, "c"'] & 5
\end{tikzcd}$$

Recognize: **_, a, c, ac, ca**

blue – insertion

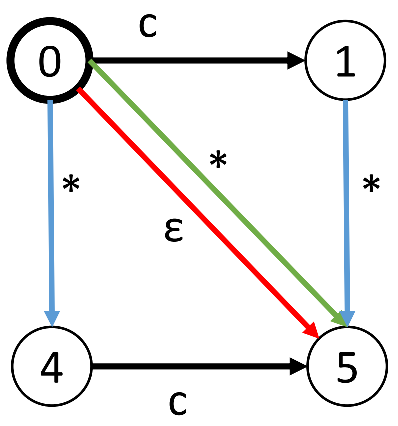
red – deletion

green - substitution

ZyLAB®

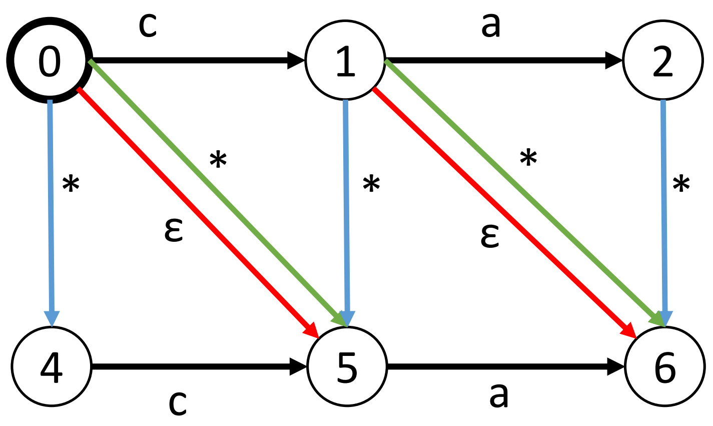
Construction of NFA

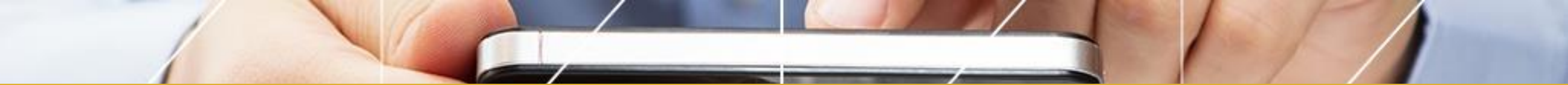
Automaton to recognize **car~1**



Construction of NFA

Automaton to recognize **car~1**


$$\begin{tikzcd}
0 \arrow[r, "c"] \arrow[d, "*"] \arrow[dr, "*"] \arrow[dr, "\varepsilon"'] & 1 \arrow[r, "a"] \arrow[d, "*"] \arrow[dr, "*"] \arrow[dr, "\varepsilon"'] & 2 \arrow[d, "*"] \\
4 \arrow[r, "c"'] & 5 \arrow[r, "a"'] & 6
\end{tikzcd}$$



Construction of NFA

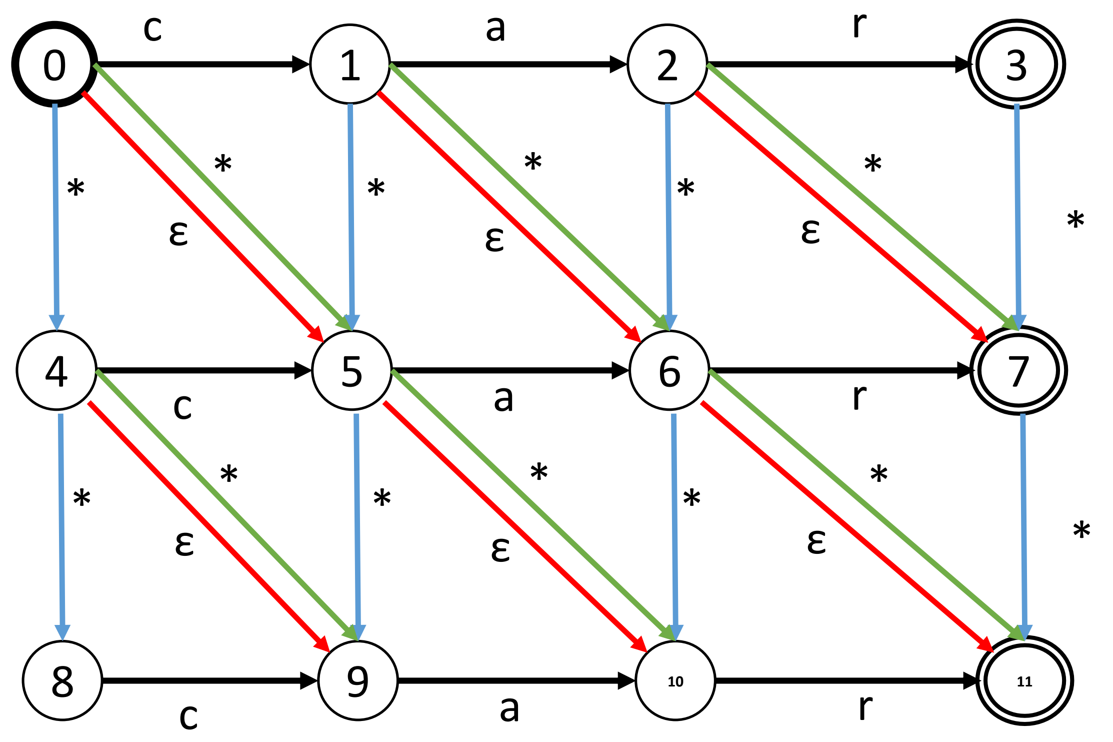
Automaton to recognize **car~1**

car~1

$$\begin{tikzcd}
0 \arrow[r, "c"] \arrow[d, "*"] \arrow[dr, bend left=10, "*"] \arrow[dr, bend right=10, "\varepsilon"'] & 1 \arrow[r, "a"] \arrow[d, "*"] \arrow[dr, bend left=10, "*"] \arrow[dr, bend right=10, "\varepsilon"'] & 2 \arrow[r, "r"] \arrow[d, "*"] \arrow[dr, bend left=10, "*"] \arrow[dr, bend right=10, "\varepsilon"'] & 3 \arrow[d, "*"] \\
4 \arrow[r, "c"'] & 5 \arrow[r, "a"'] & 6 \arrow[r, "r"'] & 7
\end{tikzcd}$$

Construction of NFA

car~2

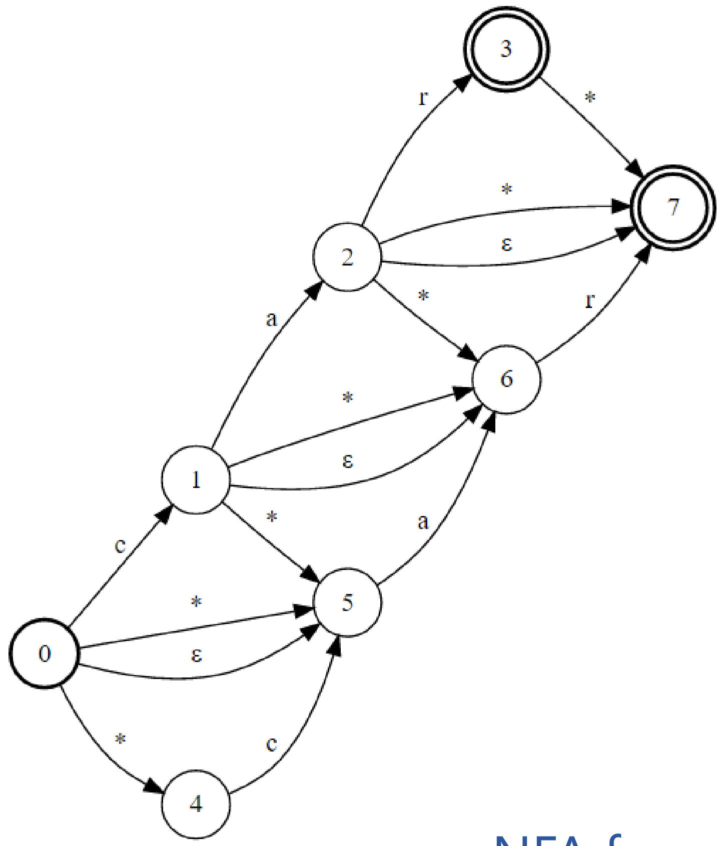


Each fuzzy degree adds one level

Levenshtein Automaton

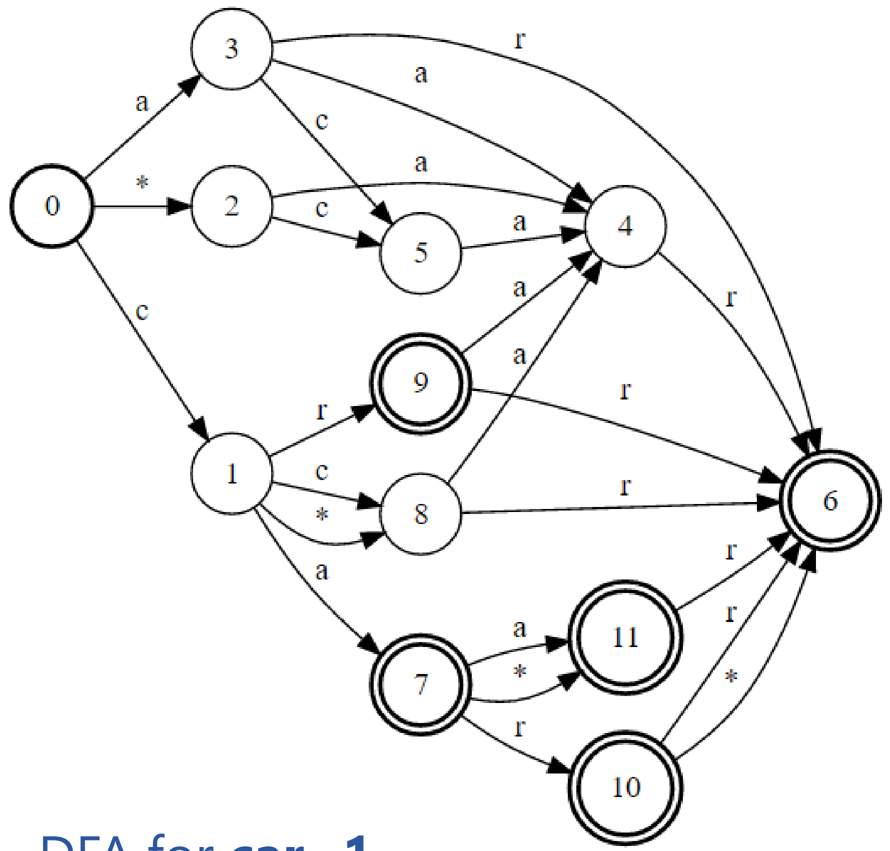
Every **NFA** can be converted to equivalent **DFA**

Levenshtein Automaton

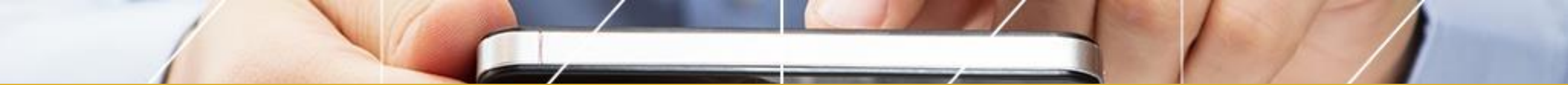


NFA for **car~1**

Powerset construction



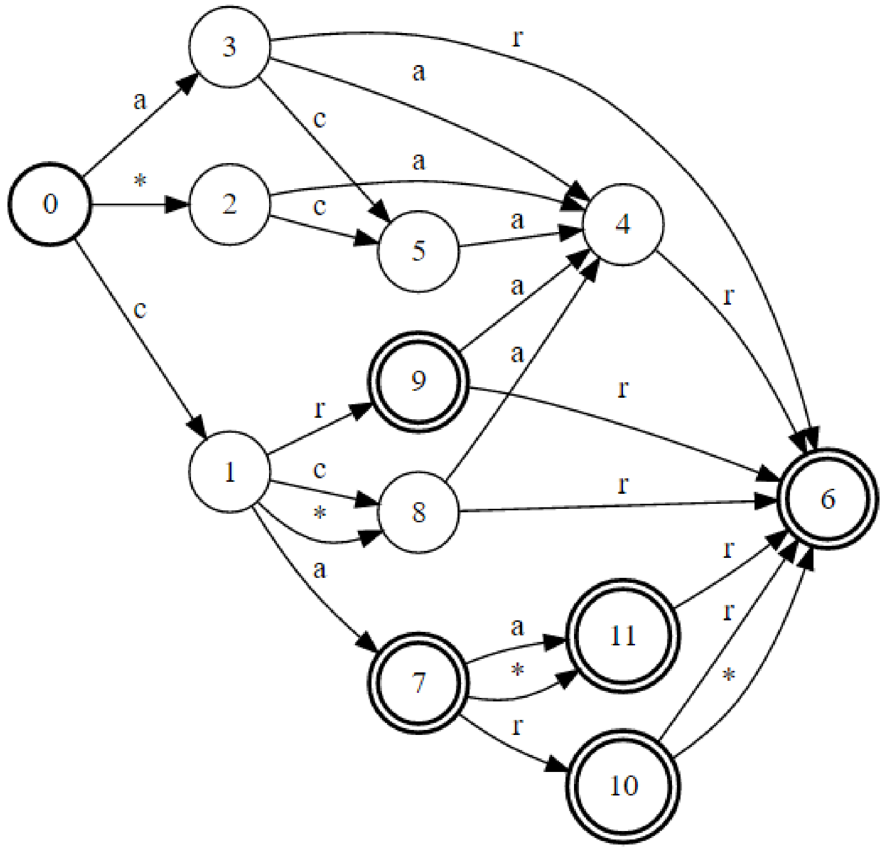
DFA for **car~1**



Levenshtein Automaton

car~1

Linear construction



Levenshtein Automaton

Matching dictionary with DFA is easy:

Difficult part

```
var dfa = CreateAutomaton(pattern, d);

foreach (var term in dictionary)
{
    var s = 0;
    foreach(var c in term)
    {
        s = dfa.Next(s, c);
    }
    if( dfa.IsFinal(s))
    {
        yield return term;
    }
}
```

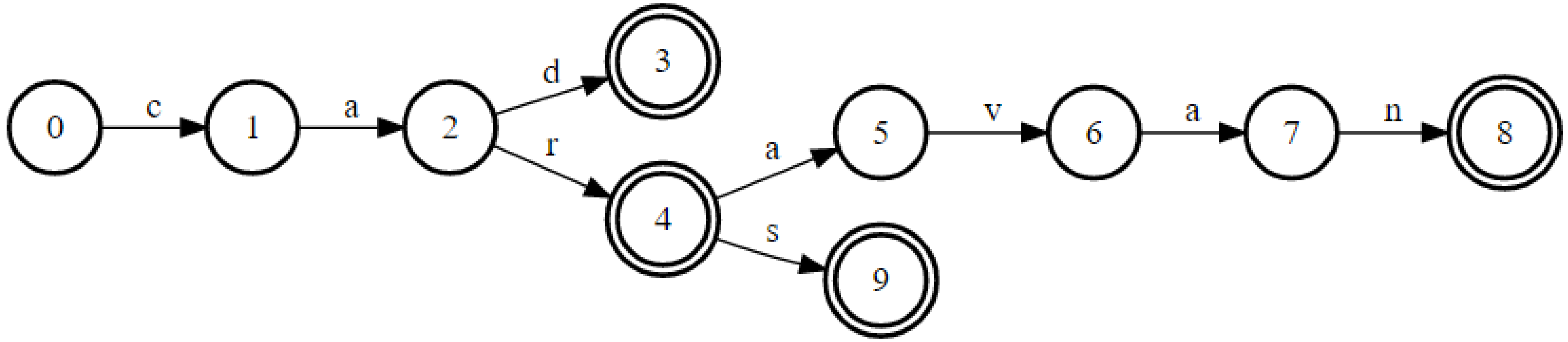
When current term and previous term share prefix, algorithm can start matching from the prefix

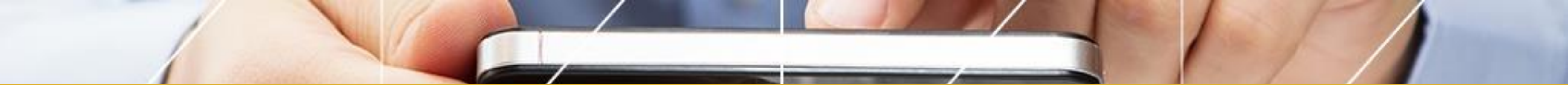
ZyLAB®

Levenshtein Automaton

Matching a Prefix Tree based dictionaries

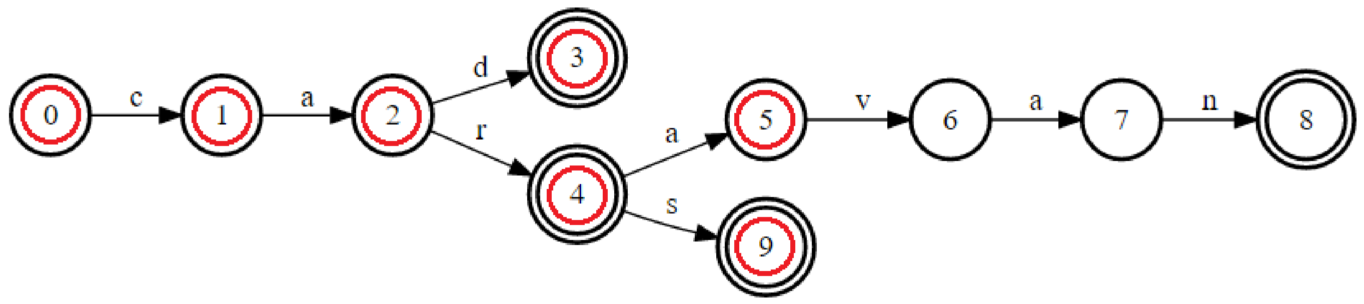
cad
car
caravan
cars



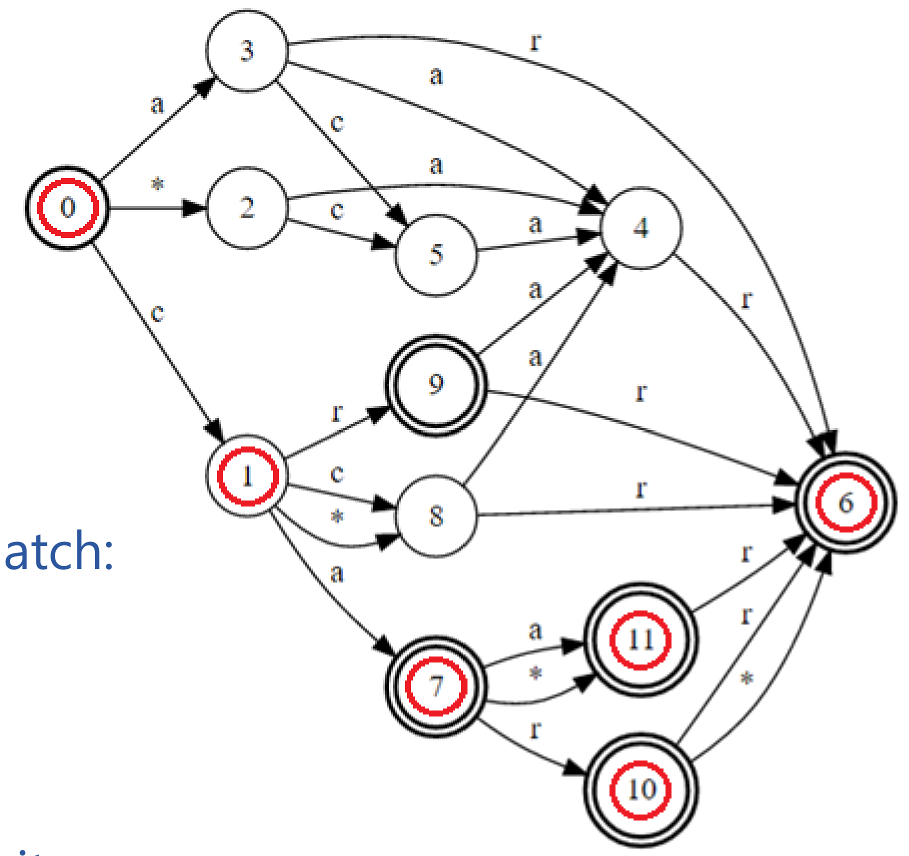


Matching dictionary with Automaton

Dictionary DFA



Fuzzy term DFA



When both DFAs are in the final state (double circle) we have a match:

- cad
- car
- cars

Fuzzy DFA can be used to generate all possible terms that match it.

ZyLAB®

Levenshtein Automaton

1) Difficult to implement
2) DFA construction might be slow for large degree **d**. In practice used when ***d* ∈ {1,2,3,4}**
3) Matching dictionary requires less than **|a+d|*N** steps, where N number of terms in dictionary

Levenshtein Automaton

H. Bunke, A Fast Algorithm for Finding the Nearest Neighbor of a Word in a Dictionary (**1993**)

*Converting NFA to DFA using Rabin–Scott powerset construction (1959). if the NFA has **n** states, the resulting DFA may have up to* $\mathbf{2^n}$ *states*

K. Schulz , S Mihov, Fast String Correction with Levenshtein-Automata (**2002**)

*How to compute, for any fixed degree **d** and any input word W, a deterministic Levenshtein-automaton in time linear in the length of W.*

Demo

Bitap algorithm

Bitap

Bitap, Shift-Or, **Shift-And** or **Baeza-Yates–Gonnet** algorithm

R. Baeza-Yates, G. Gonnet. A New Approach to Text Searching (1992)

Bálint Dömölki, An algorithm for syntactical analysis (1964)

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_
A	1
B	1
A	1
B	1

R_0

The update procedure for the R vector is:

$\boldsymbol{R_0[i] = 1}$ **for all** $\boldsymbol{i = 1..m}$

$R_{j+1}[1] = 0$ when $p_1 = t_{j+1}$

$R_{j+1}[i] = 0$ when $R_j[i-1] = 0$ and $p_i = t_{j+1}$

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_	A
A	1	0
B	1	1
A	1	1
B	1	1

R_0 R_1

The update procedure for the *R* vector is:

$R_0[i] = 1$ for all $i = 1..m$

$\mathbf{R_{j+1}[1] = 0}$ **when** $\mathbf{p_1 = t_{j+1}}$

$R_{j+1}[i] = 0$ when $R_j[i-1] = 0$ and $p_i = t_{j+1}$

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_	A	B
A	1	0	1
B	1	1	0
A	1	1	1
B	1	1	1

R_0 R_1 R_2

The update procedure for the *R* vector is:

$R_0[i] = 1$ for all $i = 1..m$

$R_{j+1}[1] = 0$ when $p_1 = t_{j+1}$

$R_{j+1}[i] = 0$ when $R_j[i-1] = 0$ and $p_i = t_{j+1}$

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_	A	B	A
A	1	0	1	0
B	1	1	0	1
A	1	1	1	0
B	1	1	1	1
	R_0	R_1	R_2	R_3

The update procedure for the *R* vector is:

$R_0[i] = 1$ for all $i = 1..m$

$\mathbf{R_{j+1}[1] = 0}$ **when** $\mathbf{p_1 = t_{j+1}}$

$\mathbf{R_{j+1}[i] = 0}$ **when** $\mathbf{R_j[i-1] = 0}$ **and** $\mathbf{p_i = t_{j+1}}$

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_	A	B	A	B
A	1	0	1	0	1
B	1	1	0	1	0
A	1	1	1	0	1
B	1	1	1	1	0
					^
	R_0	R_1	R_2	R_3	R_4

Match when value in the last row is 0

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

	_	A	B	A	B	A	B	C
A	1	0	1	0	1	0	1	1
B	1	1	0	1	0	1	0	1
A	1	1	1	0	1	0	1	1
B	1	1	1	1	0	1	0	1
					^		^	
	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7

ZyLAB®

Bitap

When pattern is small, vector R can be represented by unsigned integral value:

uint	–	for patterns with up to 32 characters
ulong	–	for patterns with up to 64 characters

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

Preprocessing step:

Build T vectors for every character C in pattern:
T[C][i] = 1 if P_i=C, 0 otherwise.

	T[A]	T[B]	T[C]
A	0	1	1
B	1	0	1
A	0	1	1
B	1	0	1

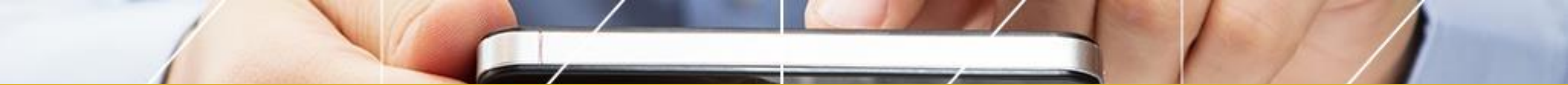
The update procedure for the *R* vector:

R_0 = 2^n-1 (n number of bits in R)
R_j+1 = (R_j << 1) | T[t_j]

Match when:

R_j & (mask) == 0

Mask = 1<<(|Pattern|-1)



Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

T[A]	T[B]	T[C]
0	1	1
1	0	1
0	1	1
1	0	1

	Bit	A
A	0	**0**
B	1	1
A	2	**0**
B	3	1
	4	0
	5	0
	6	0
	7	0
		R_3

Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

T[A]	T[B]	T[C]
0	1	1
1	0	1
0	1	1
1	0	1

	Bit	A	A<<1
A	0	**0**	0
B	1	1	**0**
A	2	**0**	1
B	3	1	**0**
	4	0	1
	5	0	0
	6	0	0
	7	0	0
		R_3	

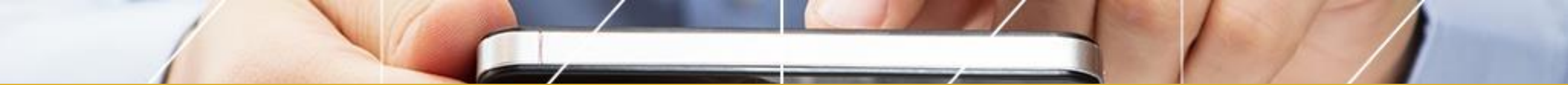
Bitap (Exact Match)

Text: ABABABC

Pattern: ABAB

T[A]	T[B]	T[C]
0	1	1
1	0	1
0	1	1
1	0	1

	Bit	A	A<<1	T[B]	(A<<1)|T[B]
A	0	**0**	0	1	1
B	1	1	**0**	0	0
A	2	**0**	1	1	1
B	3	1	**0**	0	0
	4	0	1	0	1
	5	0	0	0	0
	6	0	0	0	0
	7	0	0	0	0
		R_3			



Bitap (Exact Match)

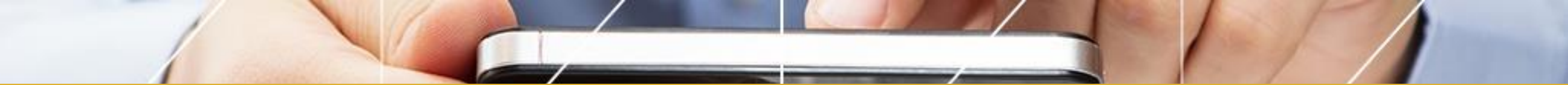
Text: ABABABC

Pattern: ABAB

T[A]	T[B]	T[C]
0	1	1
1	0	1
0	1	1
1	0	1

	Bit	A	A<<1	T[B]	(A<<1)|T[B]	B
A	0	**0**	0	1	1	1
B	1	1	**0**	0	0	**0**
A	2	**0**	1	1	1	1
B	3	1	**0**	0	0	**0**
	4	0	1	0	1	0
	5	0	0	0	0	0
	6	0	0	0	0	0
	7	0	0	0	0	0
		R_3				R_4

ZyLAB®

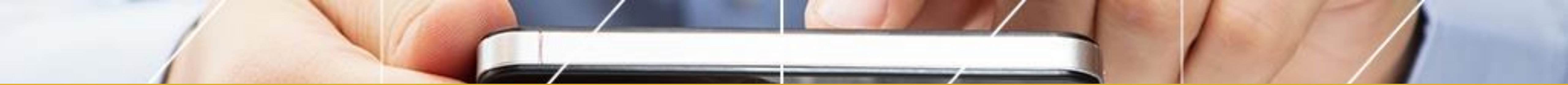


Bitap

```
/* Initialize characteristic vectors T */
for (int i = 0; i < pattern.Length; ++i)
{
    T[pattern[i]] &= ~(1ul << i);
}

/* Initialize the bit array R. */
UInt64 R = ~(0ul);
UInt64 mask = 1ul << (pattern.Length - 1);
for (int i = 0; i < text.Length; ++i)
{
    R = (R << 1) | T[text[i]];
    if ((R & mask) == 0)
    {
        return (i - pattern.Length) + 1;
    }
}
return -1;
```

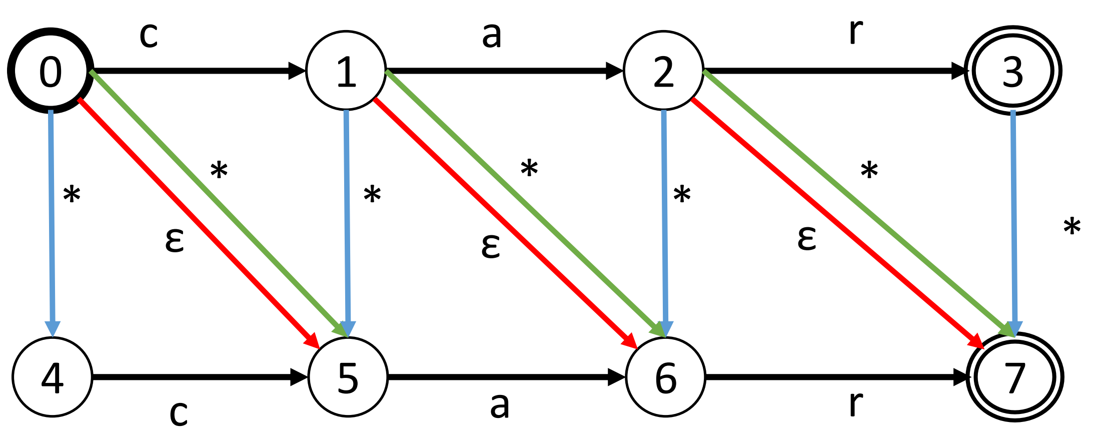
ZyLAB®



Bitap

Bitap can be used for fuzzy matching with degree **d.**

The idea of fuzzy bitap is to simulate Levenshtein automaton (NFA) using bit-parallelism, so that each level of the automaton fits in a computer word (32, 64 or 128 states per level). For each new character, all transitions of NFA are simulated using bit operations among d+1 computer words.



Bitap

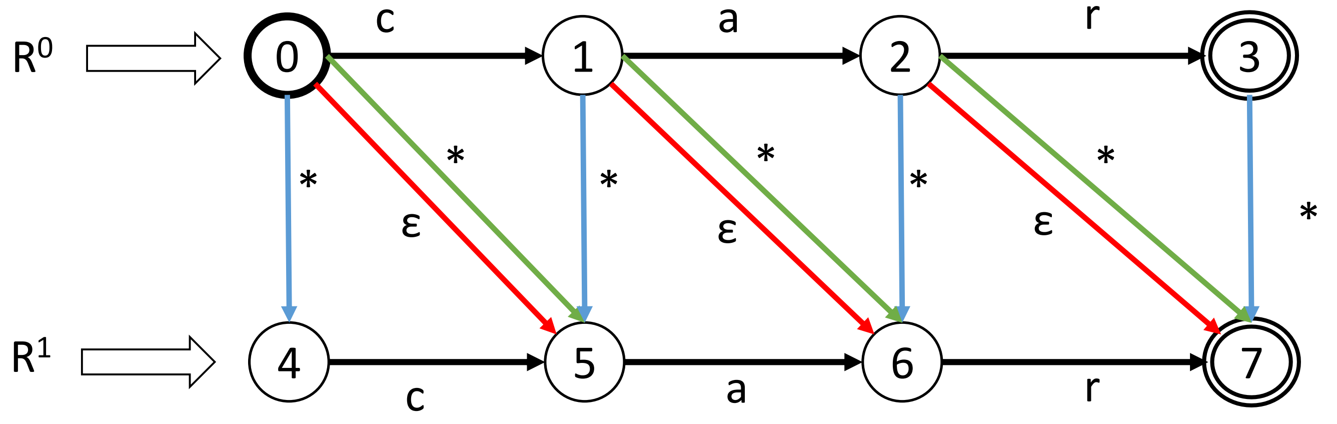
Bitap can be used for fuzzy matching with degree **d.**

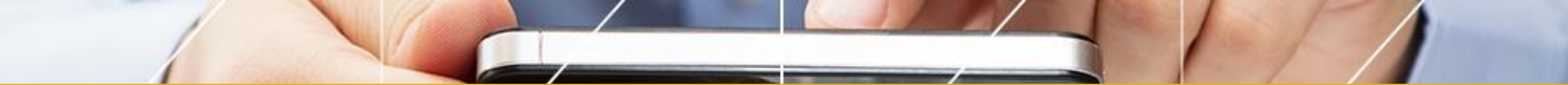
- Instead of 1 vector, we need to keep additional **d+1** bit vectors: R^0_j, R^1_j,…, R^d_j
- R^d – tracks matching with at most **d** differences

Bitap

Bitap can be used for fuzzy matching with degree **d.**

- Instead of 1 vector, we need to keep additional **d+1** bit vectors: R^0_j, R^1_j,..., R^d_j
- R^d – tracks matching with at most **d** differences





Bitap

Bitap can be used for fuzzy matching with degree **d.**

- Instead of 1 vector, we need to keep additional **d+1** bit vectors: R^0_j, R^1_j,..., R^d_j
- R^d – tracks matching with at most **d** differences

Update procedure looks like this:

$$R^d_0 = 1 \ldots 110 \ldots (d \text{ times}) \ldots 0 = (-1) \ll d$$

$$R^d_{j+1} = \underbrace{\boxed{R^{d-1}_j}}_{\text{insertion}} \;\&\; \underbrace{\boxed{(R^{d-1}_{j+1} \ll 1)}}_{\text{deletion}} \;\&\; \boxed{\underbrace{(R^{d-1}_j \ll 1)}_{\text{substitution}} \;\&\; \underbrace{((R^d_j \ll 1) \mid T[t_{j+1}])}_{\text{match}}}$$

Bitap

1) Easy to implement, might be difficult to understand
2) Memory required: O(**|Pattern|*|Alphabet|**)
3) Complexity: O(**|Text|**) when searching for all matches, or O(**|Pattern|**) when searching for the first match
4) Efficient when **|Pattern|** is small: 8,32,64,128

Benchmark

Demo

Benchmark

Dataset: Project Gutenberg, different languages, 2Gb of text

Search Engine: ZyLAB 6.6

Dictionary: 2178239 terms (2Mln)

Query: environment~2

	Time	Hits
Wagner and Fischer	1.585 sec	30771
Levenshtein automaton	**0.292 sec**	30771
Bitap	1.33 sec	30771

ZyLAB®

References

- Levenshtein (1966). "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady
- Wagner, Fischer, Michael (1974). "The String-to-String Correction Problem". Journal of the ACM
- Wu, Manber (1991). "Fast text searching with errors."
- Wu, Manber (1992). "Agrep - A Fast Approximate Pattern-Matching Tool"
- Baeza-Yates, Gonnet (1992). "A New Approach to Text Searching." Communications of the ACM, 35(10)
- Bunke (1993), A Fast Algorithm for Finding the Nearest Neighbor of a Word in a Dictionary
- Navarro, Gonzalo (2001). "A guided tour to approximate string matching". ACM Computing Surveys
- Schulz, Mihov (2002). "Fast String Correction with Levenshtein-Automata"