17th String Processing and Information Retrieval Symposium (SPIRE2010)
21st International Workshop on Combinatorial Algorithms (IWOCA'10), Also, in JDA '12

Faster Broadword Pattern Matching Algorithms for Regular Expressions and Trees

Hiroki Arimura
Graduate School of IST, Hokkaido University, Japan

Joint work with
Yusaku Kaneta, Shin-ichi Minato, Shingo Yoshizawa, Yoshikazu Miyanaga, and Rajeev Raman

Motivation

- Regular expression (Regex) matching problem
  - Classical problem in string matching
  - Many applications: Software tools, Programming lang., OS

Regular expression matching problem

- Input: a regular expression **R** of size *m* and a text **T** = **T**[1..n] of length n.
- Task: Output all (end) positions of R in T.

- Algorithms
  - Solved in O(nm) time and O(m) words [Thompson, CACM1968]
  - Many faster algorithms

Motivation

- Regular expression (Regex) matching problem
  - Classical problem in string matching
  - Many applications: Software tools, Programming lang., OS
- Demands for new applications
  - NIDS (Network Intrusion detection), CEP (Complex event processing), System log search, …
- Characteristics
  - Thousands of patterns
  - Classes of relatively simple regexs (beyond strings)
  - High-speed (throughput of several Gbps)

An Example: Network Intrusion Detection

NIDS SNORT ver2.4, 3 regular expressions (acyclic & depth 2)

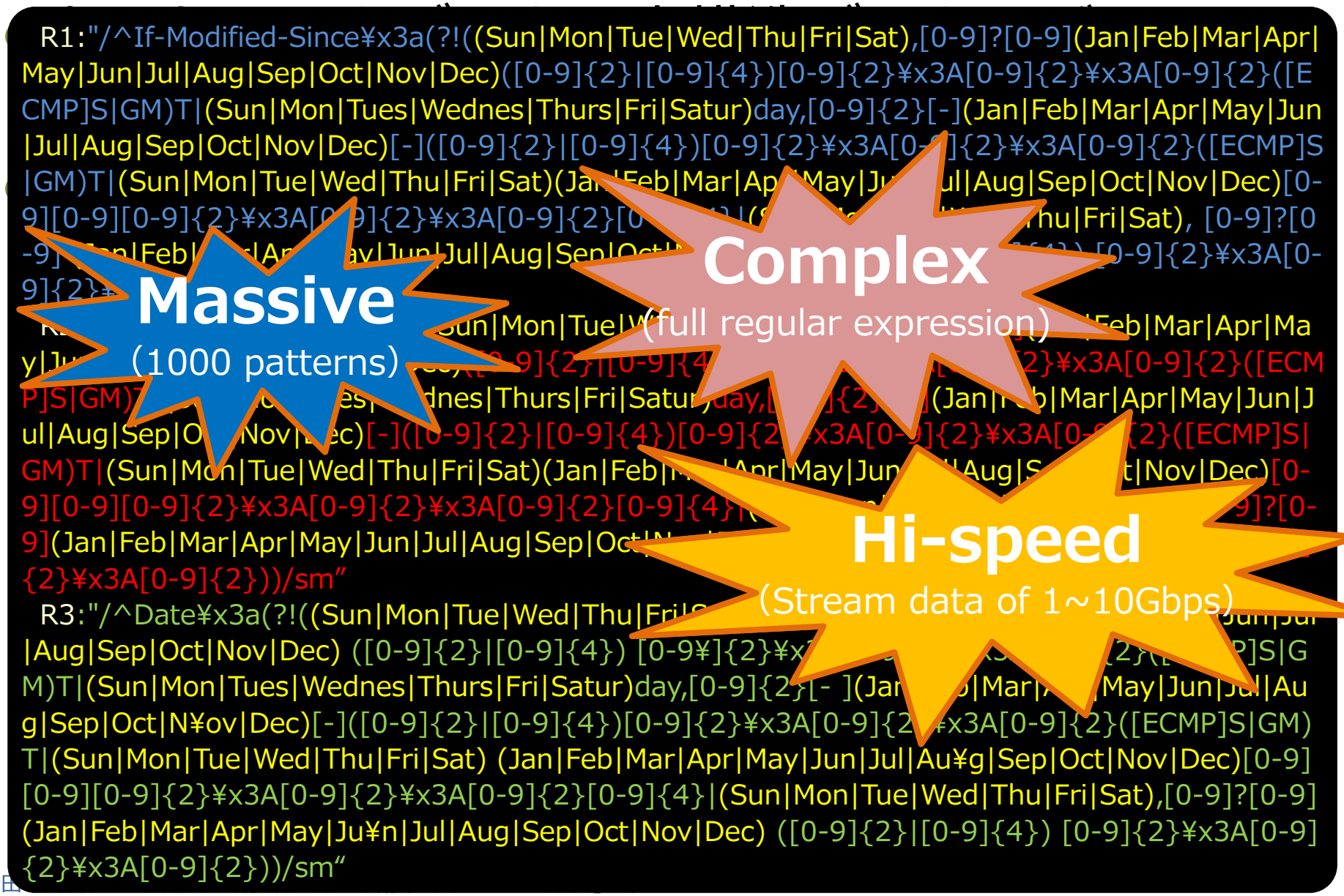
R1:"/^If-Modified-Since¥x3a(?!((Sun|Mon|Tue|Wed|Thu|Fri|Sat),[0-9]?[0-9](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)([0-9]{2}|[0-9]{4})[0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tues|Wednes|Thurs|Fri|Satur)day,[0-9]{2}[-](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[-]([0-9]{2}|[0-9]{4})[0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tue|Wed|Thu|Fri|Sat)(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[0-9][0-9][0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}[0-9]{4}|(Sun|Mon|Tue|Wed|Thu|Fri|Sat), [0-9]?[0-9] (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) ([0-9]{2}|[0-9]{4}) [0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}))/sm"

R2:"/^Last-Modified¥x3a(?!((Sun|Mon|Tue|Wed|Thu|Fri|Sat),[0-9]?[0-9](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)([0-9]{2}|[0-9]{4}) [0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tues|Wednes|Thurs|Fri|Satur)day,[0-9]{2}[-](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[-]([0-9]{2}|[0-9]{4})[0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tue|Wed|Thu|Fri|Sat)(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[0-9][0-9][0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}[0-9]{4}|(Sun|Mon|Tue|Wed|Thu|Fri|Sat),[0-9]?[0-9](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)([0-9]{2}|[0-9]{4})[0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}))/sm"

R3:"/^Date¥x3a(?!((Sun|Mon|Tue|Wed|Thu|Fri|Sat),[0-9]?[0-9](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) ([0-9]{2}|[0-9]{4}) [0-9¥]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tues|Wednes|Thurs|Fri|Satur)day,[0-9]{2}[-](Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|N¥ov|Dec)[-]([0-9]{2}|[0-9]{4})[0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}([ECMP]S|GM)T|(Sun|Mon|Tue|Wed|Thu|Fri|Sat) (Jan|Feb|Mar|Apr|May|Jun|Jul|Au¥g|Sep|Oct|Nov|Dec)[0-9][0-9][0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}[0-9]{4}|(Sun|Mon|Tue|Wed|Thu|Fri|Sat),[0-9]?[0-9](Jan|Feb|Mar|Apr|May|Ju¥n|Jul|Aug|Sep|Oct|Nov|Dec) ([0-9]{2}|[0-9]{4}) [0-9]{2}¥x3A[0-9]{2}¥x3A[0-9]{2}))/sm"

An Example: Network Intrusion Detection

NIDS SNORT ver2.4, 3 regular expressions (acyclic & depth 2)



Motivation

- Demands for new applications
  - NIDS (Network Intrusion detection), CEP (Complex event processing), System log search, …
- Characteristics
  - Thousands of patterns
  - Classes of relatively simple regexs (beyond strings)
  - High-speed (throughput of several Gbps)
- Further applications
  - Reconfigurable Hardware & Cryptgraphy
- Classical solutions are no longer sufficient!

Our approach

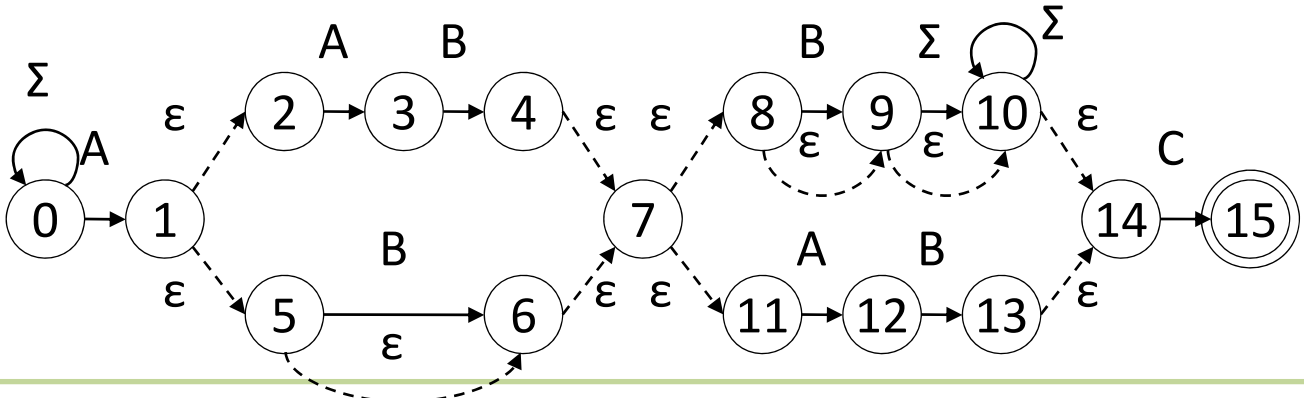
- **Bit-parallel matching approach** (in narrow sense)
  - Using parallelism inside a computer word (register) on Word RAM
  - Boolean and arithmetic operations (of AC^0)

- **SHIFT-AND method** [Baeza-Yates & Gonnet '92; Wu & Manber '92]
  - For string patterns ("exact match")
  - Simulate an NFA using Boolean operations (&, <<) and letter tables on Word RAM
  - T = O(mn/w) time, S = O(m/w + letter table) words
  - Simple and space efficient. No table lookup (unlike "Four Russian" of Myers).

Our target class of Regexs

Extended network expressions

- Acyclic (Network expressions by E. Myers)
- Base strings can have operators (Extended strings by Navarro & Raffinot ‘01)
  - Letter classs [ABC…], Wildcards Σ, Unbounded gap (.*), Bounded gaps (.{lo,hi}), Option (?) Letter repeats (A*).
- Small constant height h (from one to two)
  - Allowing unbounded branching

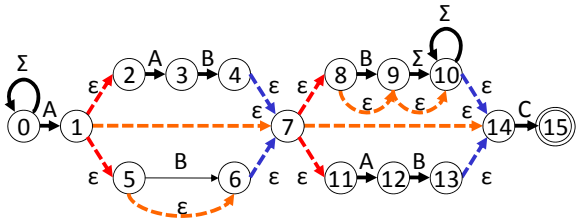


Our approach

- **Extended SHIFT-AND** [Navarro and Raffinot '01]
  - For the class of extended strings
    - A string allowing the operations: Letter classs [ABC…], Wildcards Σ, Unbounded gap (.*), Bounded gaps (.{lo,hi}), Option (?) Letter repeats (A*).
  - Uses Boolean (&, |, ~, <<) and integer addtion **(+,-)** operations
  - **Still simple and efficient**

Our results

- **Fast bit-parallel matching algorithm** for **extended network expressions** of height *d* that runs in
  - **O(ndm/w)** time
  - **O(dm/w)** space
  - **O(dm)** preprocessing



- One of the first extensions of SHIFT-AND approaches to EXNET and REG.
- **Keys**
  - New broadword simulation of NFA with "branching" operations: **Scatter**, **Gather**, and **Propagate**
  - **Bi-monotonicity lemma** for Thompson NFA

m: size of R, **n**: size of T, **d**: depth of R, **w**: word length

Summary of our results

- Our algorithm is the first extension of SHIFT-AND and Extended SHIFT-AND to the following classes:
  - **REG**: Regular expressions
  - **EXNET**: Extended network expressions

Algorithm	Class	Time	Space (in words)
This Branching Ext. S.A.	**EXNET**	**O(ndm/w)**	**O(dm/w)**
Extended SHIFT-AND (d = 1, gaps, options) [Navarro & Raffinot'01]	**EXT**	**O(nm/w)**	**O(m/w)**
SHIFT-AND (d = 1) [Baeza-Yates & Gonnet '92]	**STR**	**O(nm/w)**	**O(m/w)**

m: size of R, **n**: size of T, **d**: depth of T, **w**: word length

Comparison: Time & Space complexities

Algorithm	Class	Time	Space (in words)
Arithmetic BP (Broadword) approaches			
Ext^2 SHIFT-AND (In this talk)	**REG**	**O(n(d+(m/w)) log(w))**	**O((d+(m/w)) log(w))**
	EXNET	**O(ndm/w)**	**O(dm/w)**
Extended SHIFT-AND [Navarro, Raffinot '01]	**EXT**	**O(nm/w)**	**O(m/w)**
SHIFT-AND [BYG '92]	**STR**	**O(nm/w)**	**O(m/w)**
Bille [ICALP'06]	**REG**	**O(nlog(w)m/w)**	**O(log(w)m/w)**
Four Russian (Table-lookup)			
Myers [JACM, '92]	**REG**	**O(nm/log n)**	**O(nm/log n)**

- **m**: size of R, **n**: size of T, **d**: depth of T, **w**: word length
- Our algorithm is **most efficient for EXNET of small depth *d***.
  If *d* = O(1), our algorithm is the only algorithm that achieves O(nm/w) time and O(m/w) space simulteneously for class EXNET

Privious results

Related work

- Thompson's algorithm [Thompson, CACM1968]
  - O(nm) time and O(m) words as baseline
- Myers' algorithm
  - O(nm/w) time and word space.
  - The first algorithm breaking the "O(mn) barrier" using "four Russian" technique of table-lookup
  - Practically, memory consuming
  - A version by Grushkov NFA [Navarro]

Broadword approach ("Arithmetic Bit-parallel")

- SHIFT-AND [Baeza-Yates & Gonnet '92; Wu & Manber '92]
- Extended SHIFT-AND [Navarro & Raffinot '01]
- Bille's algorithm [Bille ICALP'06]

1. Thompson's NFA simulation

- **Classical solution**
- **Outline of Thompson's algorithm**

Preprocessing

- Transform an input regular expression **R** to the equivalent "Thompson" NFA $\mathbf{N_R}$ (TNFA).

Run-time:

- Simulate TNFA $\mathbf{N_R}$ on an input text

- **Complexity:** O(nm) time, and O(m) preprocessing and space

1. Thompson's NFA simulation

- **Outline of algorithm**

Preprocessing

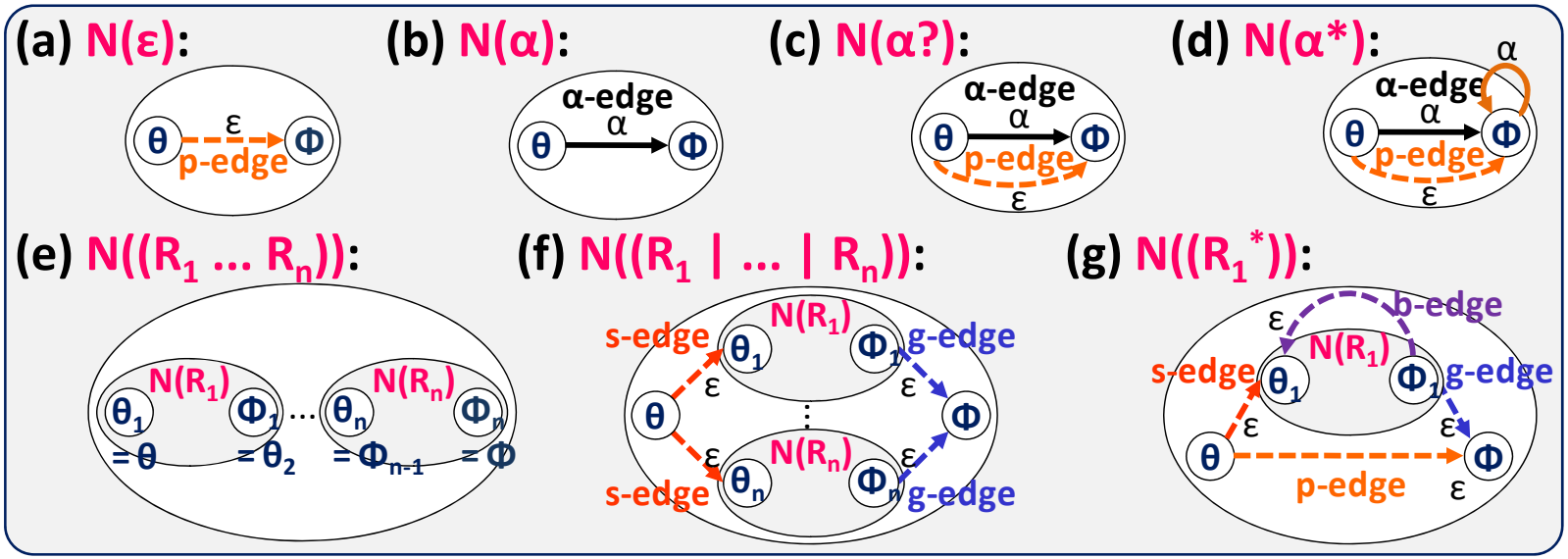
- Transform an input regular expression **R** to the equivalent NFA $\mathbf{N_R}$ (Thompson NFA, TNFA).
- Construct a set of bit-masks for TNFA.

Run-time:

- Simulate TNFA $\mathbf{N_R}$ on an input text using the bitmasks based on bit-parallel method.

- **Key**: Efficient bit-parallel implementation of three ε-move operations: **Scatter**, **Gather**, and **Propagate**

Thompson NFA [Thompson, CACM1968]



- **A NFA equivalent to** a regular expression **R**
  - recursively constructed from R
- **Depths** of subexpressions and states: # of nesting of OR ("|")
- The **source θ** and the **sink ϕ**
- TNFA for EXNET has letter edges and ε-edges

1. Thompson's NFA simulation

- Maintain the set of active states by a list D of state IDs
- Simulates letter-moves
- Compute "ε-closure" (transitive closure of ε-edges)

RunTNFA(T=t_1 ... t_n: input text)

1. **Initialize D**
2. **D ← EpsClo$_N$(D)**;
3. **for i ← 1, ... , n do**
4. **D ← Move$_N$(D, t_i)**;
5. **Initialize D**
6. **D ← EpsClo$_N$(D)**;
7. **if Accept then**
8. output Match;
9. **end for**

2. Four Russian algorithm by Myers

- Encodes a state set of TNFA in a m-bit mask D
- Simulates letter-moves and ε-moves by table look-up for x-bit blocks
- Time T = O((1/ε)nm/w) for x = O(ε log **n**) for $0 \leqq \varepsilon \leqq 1$
- Space S = $O(2^x(m/w)) = O(n^{\varepsilon}(m/w))$. Exponential in x.

```
RunTNFA(T=t_1 ... t_n: input text)
1.  D ← Init_N;
2.  D ← EpsClo_N(D);
3.  for i ← 1, ... , n do
4.    D ← Move_N(D, t_i);
5.    D ← X | Init_N;
6.    D ← EpsClo_N(D);
7.    if D & Accept_N ≠ 0 then
8.      output Match;
9.  end for
```

2’. BP-Thompson [Baeza-Yates and Gonnet ’92]

- Encodes a state set of TNFA in a m-bit mask D
- Simulates ε-moves by table look-up for x-bit blocks as well. We can simulates letter-moves (e.g. **$\mathrm{Move}_N(D, \alpha)$** by SHIFT-AND method
- Time and space do not change. (Practically fast)

RunTNFA(T=$t_1 \ldots t_n$: input text)

1. **D ← Init_N;**
2. **D ← $\mathrm{EpsClo}_N(D)$;**
3. **for i ← 1, … , n do**
4. **D ← $\mathrm{Move}_N(D, t_i)$;**
5. **D ← X | Init_N;**
6. **D ← $\mathrm{EpsClo}_N(D)$;**
7. **if D & Accept_N ≠ 0 then**
8. output Match;
9. **end for**

$\mathrm{Move}_N(X, \alpha)$ ≡

1. **X ← (X«1) & Chr[α];**
2. **return X;**

Init_N ≡ return BIT($\{\theta_R\}$);

Accept_N ≡ return BIT($\{\Phi_R\}$);

BIT(D) denotes the bitmask representation for state set **D**

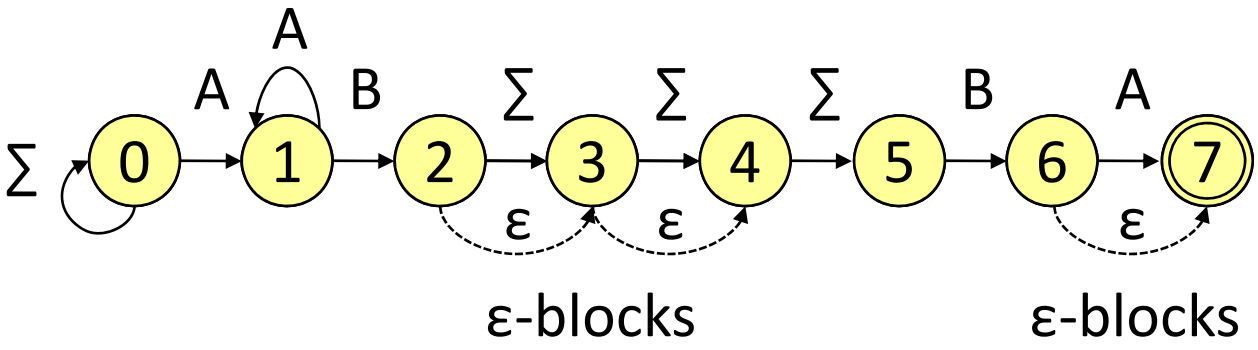
Our approach

- **Extended SHIFT-AND** [Navarro and Raffinot '01]
  - For the class of extended strings
    - A string allowing the operations: Letter classs [ABC…], Wildcards Σ, Unbounded gap (.*), Bounded gaps (.{lo,hi}), Option (?) Letter repeats (A*).
  - Uses Boolean (&, |, ~, <<) and integer addtion **(+,-)** operations
  - **Still simple and efficient**

Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

- Bit-parallel matching method for extended strings
  - Example) R = A+B.{1,3}BA?

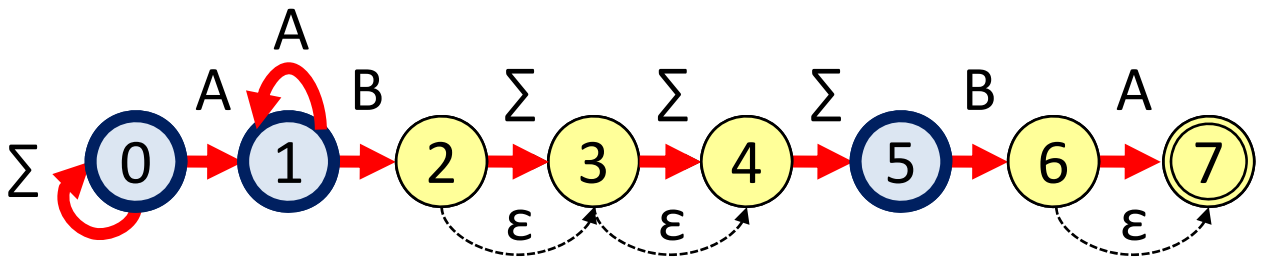


Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

1. a

MOVE = (STATE >> 1 | INIT) & CHR[t];
REPEAT = STATE & REP[t];
STATE = MOVE | REPEAT;



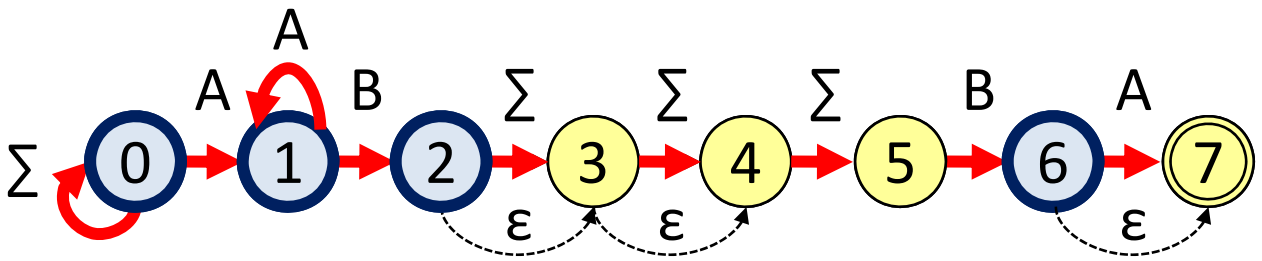
STATE	**1**	**1**	0	0	0	**1**	0	0
STATE >> 1 | INIT	1	**1**	**1**	0	0	0	**1**	0
CHR[B]	0	0	1	1	1	1	1	0
MOVE	0	**0**	**1**	0	0	0	**1**	0

Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

1. a

MOVE = (STATE >> 1 | INIT) & CHR[t];
REPEAT = STATE & REP[t];
STATE = MOVE | REPEAT;



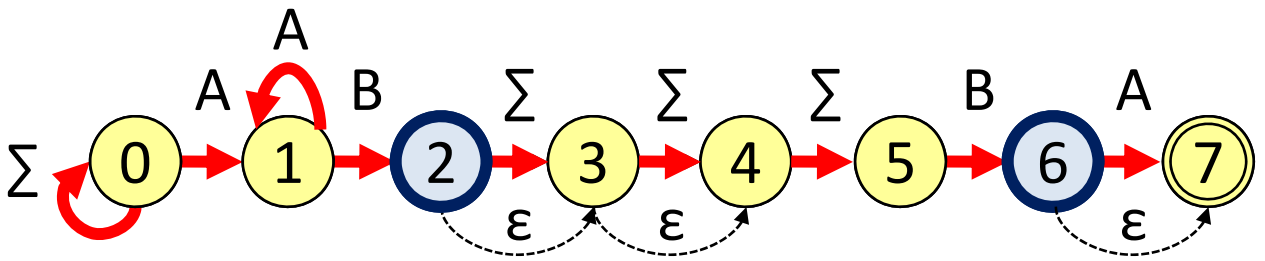
STATE	1	1	0	0	0	1	0	0
STATE >> 1 | INIT	1	1	1	0	0	0	1	0
CHR[B]	0	0	1	1	1	1	1	0
MOVE	0	0	1	0	0	0	1	0

Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

2. To not activate wrong state, we perform bitwise-AND with bitmask CHR[t]

```
MOVE   = (STATE >> 1 | INIT) & CHR[t];
REPEAT = STATE & REP[t];
STATE  = MOVE | REPEAT;
```



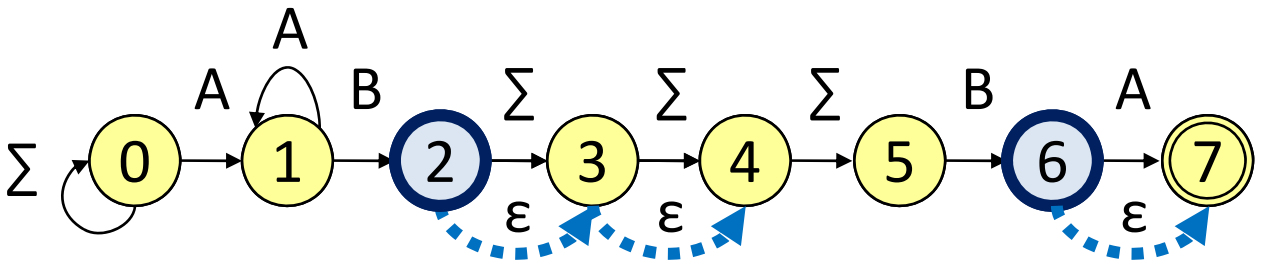
STATE	1	1	0	0	0	1	0	0
STATE >> 1 | INIT	1	1	1	0	0	0	1	0
CHR[B]	0	0	1	1	1	1	1	0
MOVE	0	0	1	0	0	0	1	0

Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

2. To detect the lowest 1 in each ε-blocks, we perform integer subtraction

```
HIGH = STATE | EpsEND;
LOW = HIGH – EpsBEG;
STATE = (EpsBLK & (~LOW ⊕ HIGH)) | STATE;
```



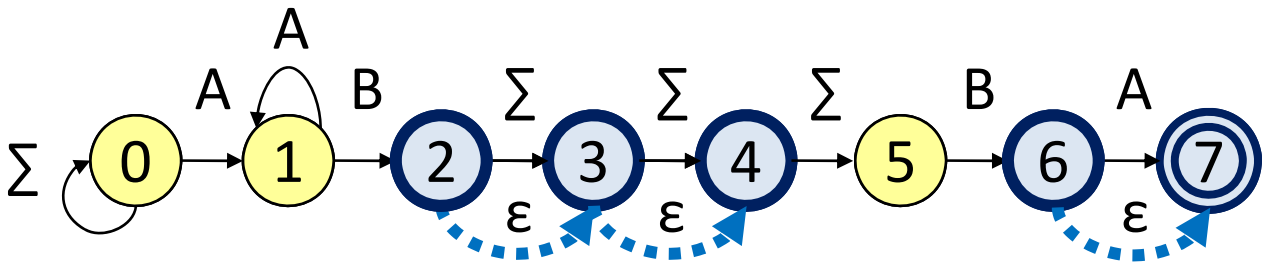
STATE	0	0	**1**	0	0	0	**1**	0
HIGH	0	0	**1**	0	**1**	0	**1**	**1**
EpsBEG	0	0	**1**	0	0	0	**1**	0
LOW	0	0	0	0	**1**	0	0	**1**

Extended SHIFT-AND method

[Navarro, Raffinot, RECOMB'01]

2. a

HIGH = STATE | EpsEND;
LOW = HIGH – EpsBEG;
STATE = (EpsBLK & (~LOW ⊕ HIGH)) | STATE;



~LOW	1	1	**1**	1	0	1	**1**	0
HIGH	0	0	1	0	**1**	0	1	**1**
~LOW ⊕ HIGH	1	1	**0**	1	1	1	**0**	1
EpsBLK	0	0	**1**	1	1	0	**1**	0
LK & ~LOW ⊕ HIGH	0	0	**0**	1	1	0	**0**	1

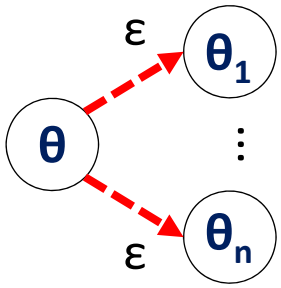
Our algorithm

Our bit-parallel algorithm

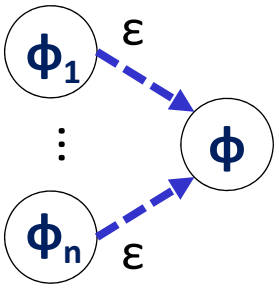
- **Key:** Efficient Computation of ε-closure $\mathbf{EpsClo_N(D)}$

- Two ingredients:

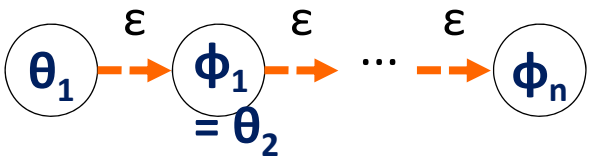
1. **Bi-monotonicity lemma** for Thompson NFA
2. Broadword (Bit-parallel) implementation three ε-move operations: **Scatter**, **Gather**, and **Propagate**



Scatter

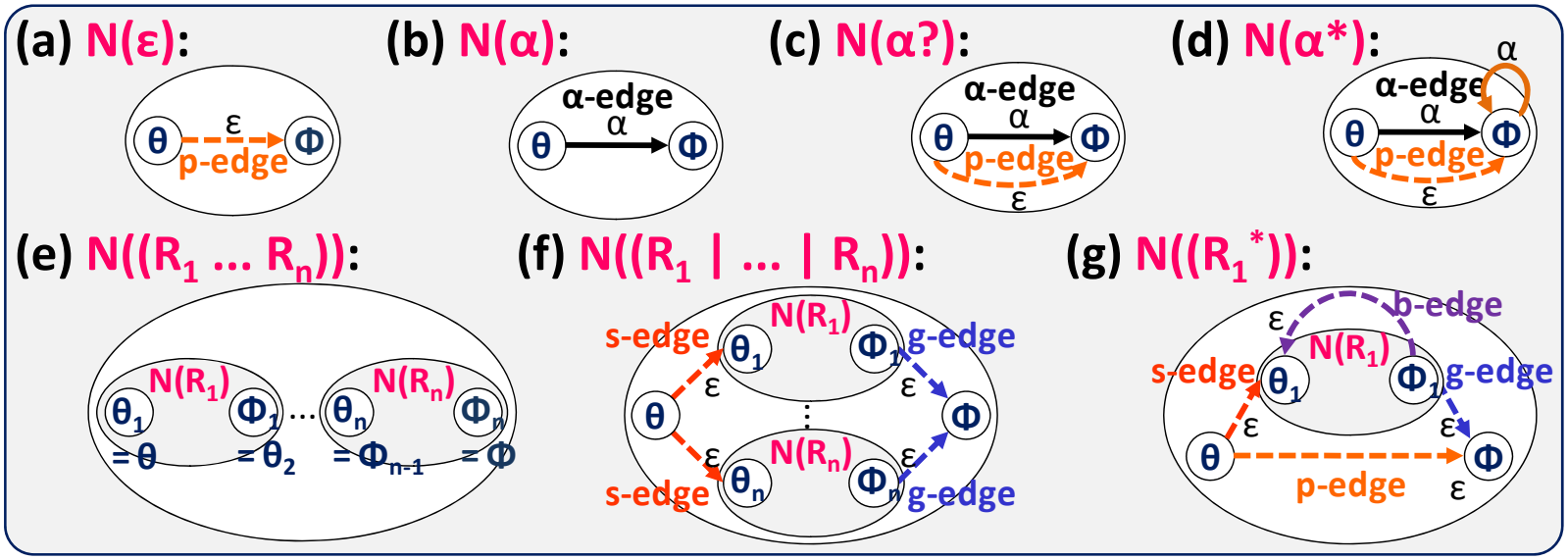


Gather



Propagate

Thompson NFA [Thompson, CACM1968]



- **A NFA equivalent to** a regular expression **R**
  - recursively constructed from R
- **Depths** of subexpressions and states: # of nesting of OR ("|")
- The **source θ** and the **sink ϕ**
- TNFA for EXNET has three types of ε-edges
  - **Scatter eges, Gather edges, Propagate edges**
- TNFA for REG has one more type of ε-edges
  - **Back edges**

Our bit-parallel algorithm

- Encodes a state set of TNFA in a bitmask D
- Simulates α-moves **$\mathrm{Move}_N(D, \alpha)$** by SHIFT-AND method [Baeza-Yates and Gonnet '92]
- Simulates ε-closure **$\mathrm{EpsClo}_N(D)$** by the combination of three operations **Scatter, Gather,** and **Propagate** [This paper]

RunTNFA(T=t_1 ... t_n: input text)

1. D ← Init_N;
2. D ← $\mathrm{EpsClo}_N(D)$;
3. **for** i ← 1, ... , n **do**
4. D ← $\mathrm{Move}_N(D, t_i)$;
5. D ← X | Init_N;
6. D ← $\mathrm{EpsClo}_N(D)$;
7. **if** D & Accept_N ≠ 0 **then**
8. output Match;
9. **end for**

$\mathrm{Move}_N(X, \alpha)$ ≡

1. X ← (X«1) & Chr[α];
2. **return** X;

Init_N ≡ **return** BIT($\{\theta_R\}$);

Accept_N ≡ **return** BIT($\{\Phi_R\}$);

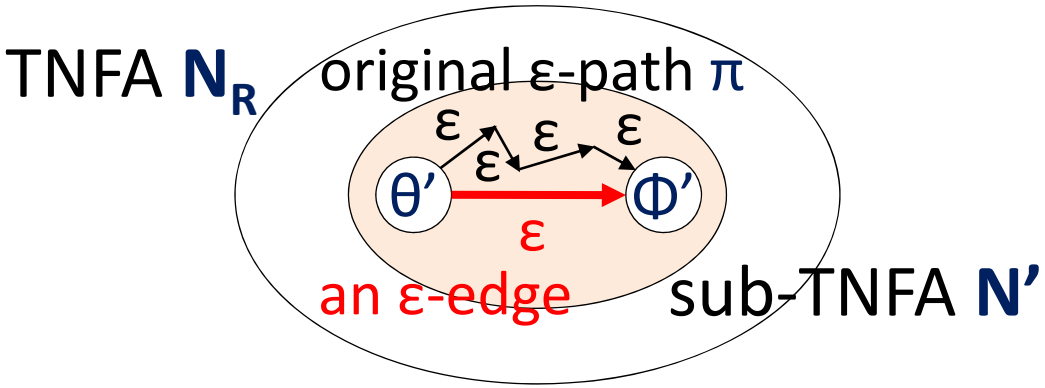
BIT(D) denotes the bitmask representation for state set **D**

Bimonotonicity lemma

- Let $\mathbf{N_R} = (V, E, \theta, \phi)$ be TNFA with source θ and sink ϕ

Procedure Bypassing

1. Visit every **sub-TNFA N'** of $\mathbf{N_R}$ whose source θ' and sink ϕ' are connected by an ε-path π
2. Add **an ε-edge** directly connecting θ' and ϕ'

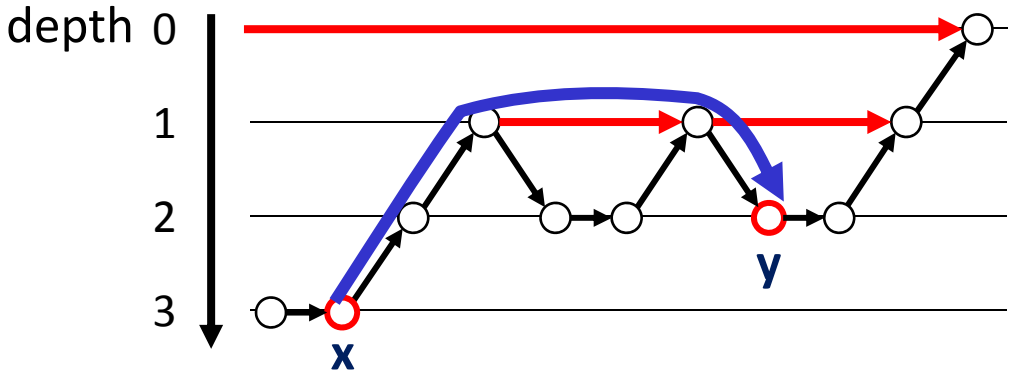


- Bypassing can be done in $O(m)$ time in preprocessing
- **Expand(R)** denotes the NFA obtained from $\mathbf{N_R}$ by bypassing

Bimonotonicity lemma

- Let x, y be any states in the TNFA N_R for **Expand(R)**
- For any states x, y, define $d(x) \leq_1 d(y)$ iff $d(x) - d(y) \leq 1$

Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path $\pi' = (x_1 = x, \ldots, x_n = y)$ from **x** to **y** in N_R such that $d(x_1) \geq_1 \cdots \geq_1 d(x_k)$ and $d(x_k) \leq_1 \cdots \leq_1 d(x_n)$



Step 2.
There exists a bimonotone ε-path connecting x and y

Bimonotonicity lemma

- Let x, y be any states in the TNFA N_R for **Expand(R)**
- For any states x, y, define $d(x) \leq_1 d(y)$ iff $d(x) - d(y) \leq 1$

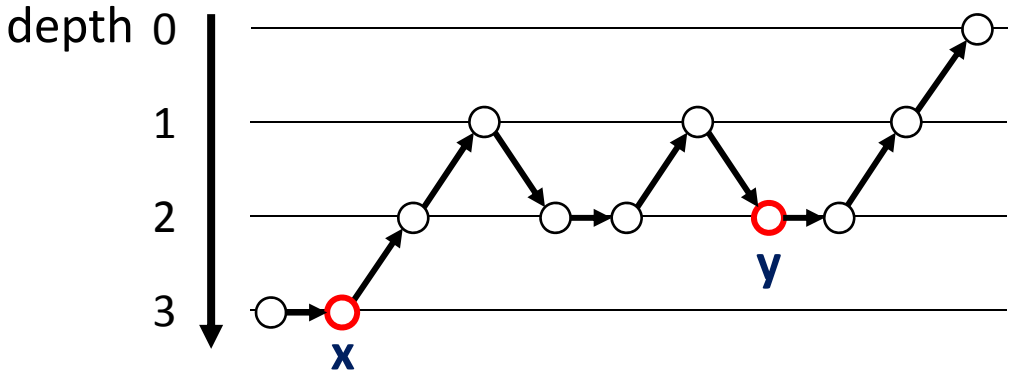
Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path $\pi' = (x_1 = x, \ldots, x_n = y)$ from **x** to **y** in N_R such that $d(x_1) \geq_1 \cdots \geq_1 d(x_k)$ and $d(x_k) \leq_1 \cdots \leq_1 d(x_n)$ (Eq1)

Errata: There is **an error** in the definition of the bimonotonicity in Page XXX of the conference proceedings. **The direction of inequality $\leq_1$ is opposite to the correct one :-)**
Please correct it as in the above (Eq1).

Bimonotonicity lemma

- Let x, y be any states in the TNFA N_R for **Expand(R)**
- For any states x, y, define $d(x) \leq_1 d(y)$ iff $d(x) - d(y) \leq 1$

Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path $\pi' = (x_1 = x, \ldots, x_n = y)$ from **x** to **y** in N_R such that $d(x_1) \geq_1 \cdots \geq_1 d(x_k)$ and $d(x_k) \leq_1 \cdots \leq_1 d(x_n)$

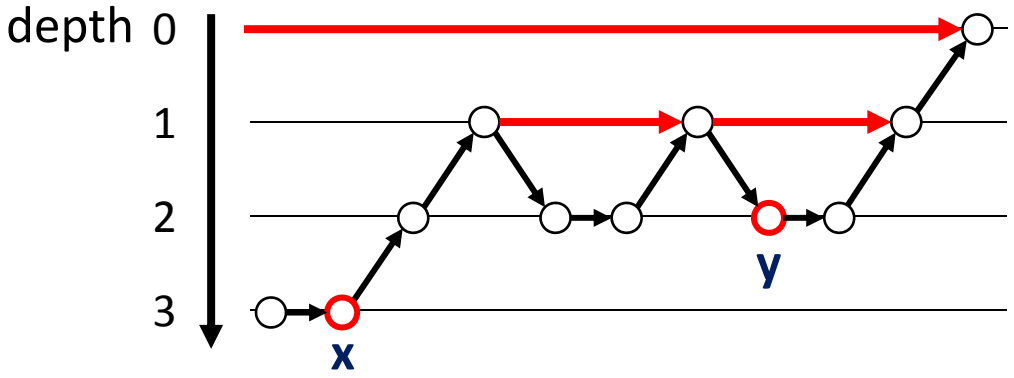


Step 0.
Given a TNFA with an ε-path π from state **x** to state **y**

Bimonotonicity lemma

- Let x, y be any states in the TNFA N_R for **Expand(R)**
- For any states x, y, define $d(x) \leq_1 d(y)$ iff $d(x) - d(y) \leq 1$

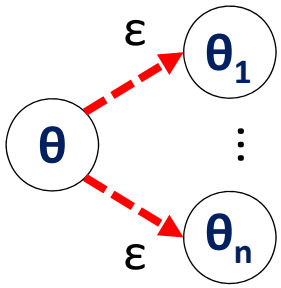
Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path $\pi' = (x_1 = x, \ldots, x_n = y)$ from **x** to **y** in N_R such that $d(x_1) \geq_1 \cdots \geq_1 d(x_k)$ and $d(x_k) \leq_1 \cdots \leq_1 d(x_n)$



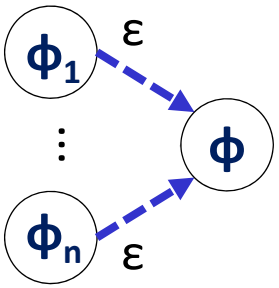
Step 1.
Applying bypassing transformation to the TNFA N_R

Bit-parallel implementation of three ε-move operations

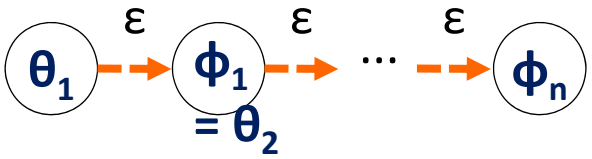
- **Scatter** simulates ε-moves by scatter edges
- **Gather** simulates ε-moves by gather edges
- **Propagate** simulates ε-closure by propagate edges
- Operations are separately done for each depth by a specified order.



Scatter edges



Gather edges

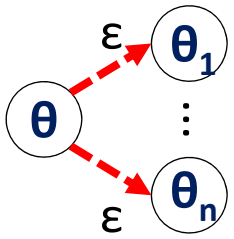


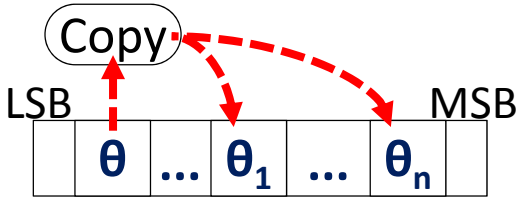
Propagate edges

Scatter operation

- **Basic idea**: carry propagation of integer subtraction

  (The origin of using carry propagation is due to [Navarro & Raffinot, '01])





	1	0	0	0	0	0	0	0
–)	0	0	0	0	0	0	0	**1**
	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Preprocess: for depth **k**, we construct the following bitmasks

- $BLK_S[k]$: the state $j = \theta_n + 1$
- $SRC_S[k]$: the source state $j = \theta$
- $DST_S[k]$: the destination states $j \in \{\theta_1, \ldots, \theta_n\}$

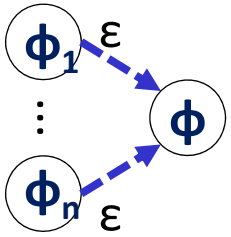
Runtime: for a state set **D** and depth **k**, we perform

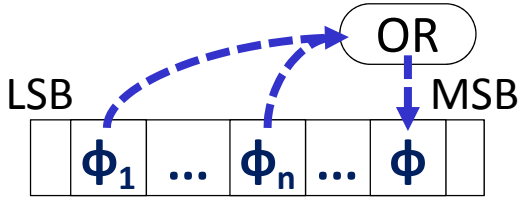
- **Scatter**(D, k) ≡ { $D \leftarrow D \mid ((BLK_S[k] - (D \,\&\, SRC_S[k])) \,\&\, DST_S[k]$; }

Gather operation

- **Basic idea**: carry propagation of integer addition

  (The origin of using carry propagation is due to [Navarro & Raffinot, '01])





	0	0	1	0	1	0	1	0
+)	0	1	1	1	1	1	1	1
	1	0	1	0	1	0	1	1

Preprocess: for depth **k**, we construct the following bitmasks

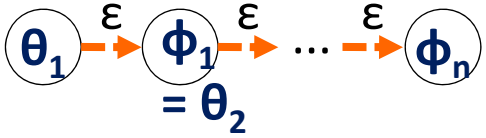
- $BLK_G[k]$: the states **j** in interval $[\Phi_1..\Phi\text{-}1]$
- $SRC_G[k]$: the source states $j \in \{\Phi_1, ..., \Phi_n\}$
- $DST_G[k]$: the destination state $j = \Phi$

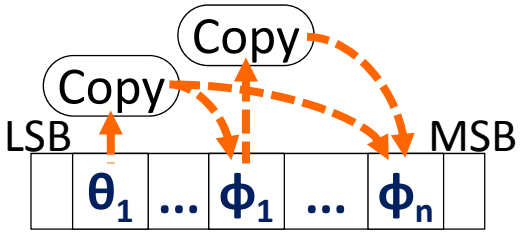
Runtime: for a state set **D** and depth **k**, we perform

$$\text{Gather}(D, k) \equiv \{\ D \leftarrow D \mid ((BLK_G[k] + (D \,\&\, SRC_G[k])) \,\&\, DST_G[k];\ \}$$

Propagate operation

- **Basic idea**: similar to that of Extended SHIFT-AND
- An **ε-block** is a maximal consecutive sequence of ε-edges.
- We can compute **the ε-closure of a set of ε-blocks** by using **the propagate operation** of the Extended SHIFT-AND [Navarro & Raffinot, '01].





Preprocess: for depth **k**, we construct the following bitmasks

- $BLK_P[k]$: the states **j** in ε-block $B = \{\theta_1, \Phi_1 = \theta_2, \cdots, \Phi_n\}$
- $SRC_P[k]$: the least significant state $j = \min(B) = \theta_1$ in ε-block **B**
- $DST_P[k]$: the most significant state $j = \min(B) = \Phi_n$ in ε-block **B**

Runtime: for a state set **D** and depth **k**, we perform

$$\text{Propagate}(D, k) \equiv \{ A \leftarrow (D \,\&\, BLK_P[k]) \mid DST_P[k];$$
$$D \leftarrow D \mid (BLK_P[k] \,\&\, ((\sim(A - SRC_P[k])) \oplus A)); \}$$

Putting them together

- The following algorithm correctly computes the ε-closer EpsClo(D) in O(dm/w) time, O(dm/w), and O(dm) preprocessing

Procedure EpsClo_N(S: a state set**)**

1. **for k ← d(R), ..., 1 do**
2. ** S ← Propagate(S, k);**
3. ** S ← Gather(S, k-1);**
4. **end for**
5. **S ← Propagate(S, 0);**
6. **for k ← d(R), ..., 1 do**
7. ** S ← Scatter(S, k-1);**
8. ** S ← Propagate(S, k);**
9. **end for**
10. **return S;**

Main result for the class EXNET

- Scatter, Gather, and Propagate operations are computable in O(m/w) preprocessing, time, and space.
- Combining $\mathbf{EpsClo_N}$ and $\mathbf{Move_N}$ with our algorithm **BP-Match,** we have:

Theorem 1. Our bit-parallel algorith for **EXNET** solves the regular expression matching problem for **EXNET** of extended network expressions in

- **O(ndm/w)** time
- **O(dm/w)** space
- **O(dm)** preprocessing

m: size of R, **n**: size of T, **d**: depth of T, **w**: word length

Extension to the class REG

- For a modified algorithm for **REG** by the **barrel shifter** technique in VLSI design, we have:

Theorem 2. Our modified algorithm for **REG** solves the regular expression matching problem for **REG** of general regular expressions in

- **O(ndlog(w)m/w)** time
- **O(dlog(w)m/w)** space
- **O(dlog(w)m)** preprocessing

m: size of R, **n**: size of T, **d**: depth of R, **w**: word length

Note:
- If there are at most constant number of back edges with mutually distinct lengths, then we can replace the **O(log w)** term with **O(1)**
- If the O(1)-bit-reversal operation is available, then we can also replace the **O(log w)** term with **O(1)**

Comparison: Time & Space complexities

Algorithm	Class	Time	Space (in words)
Arithmetic BP (Broadword) approaches			
Ext^2 SHIFT-AND (In this talk)	**REG**	**O(n(d+(m/w)) log(w))**	**O((d+(m/w)) log(w))**
	EXNET	**O(ndm/w)**	**O(dm/w)**
Extended SHIFT-AND [Navarro, Raffinot '01]	**EXT**	**O(nm/w)**	**O(m/w)**
SHIFT-AND [BYG '92]	**STR**	**O(nm/w)**	**O(m/w)**
Bille [ICALP'06]	**REG**	**O(nlog(w)m/w)**	**O(log(w)m/w)**
Four Russian (Table-lookup)			
Myers [JACM, '92]	**REG**	**O(nm/log n)**	**O(nm/log n)**

- **m**: size of R, **n**: size of T, **d**: depth of T, **w**: word length
- Our algorithm is **most efficient for EXNET of small depth *d***.
  If *d* = O(1), our algorithm is the only algorithm that achieves O(nm/w) time and O(m/w) space simulteneously for class EXNET

Application1: Hardware implementation

- Implementation of pattern matching systems on modern parallel hardwares
- Reconfigurable Hardware: FPGA
  - Enabling Dynamic reconfigulation
- Separated slides…

Application2: Private Search

- Oblivious Pattern Matching Using Homomorphic Encryption for Extended String Patterns
  - Hiroki Harada (Tsukuba Univ.) Hirohito Sasakawa, Hiroki Arimura (Hokkaido Univ.), Jun Sakuma (Tsukuba Univ.)
  - To appear in Computer Security Symposium Japan (CCS'13), Oct 22, 2013 (In Japanese).

Title and abstract in CCS Homepage:

- Two party protocol for "Private string matching" (PIR) for the class of extended string pattens (Navarro & Raffinot '01)
- Based on Oblivious NFA simulation by homomorphic cryptgraphy operations.
- Possible application to Private Genome searching
- Contact after the presentation.

Conclusion

- **Fast bit-parallel matching algorithm** for **extended network expressions** that runs in
  - **O(ndm/w)** time
  - **O(dm/w)** space
  - **O(dm)** preprocessing
- Extension for **regular expressions**
- Future works:
  - Tree and XML matching

m: size of R, **n**: size of T, **d**: depth of R, **w**: word length

Hardware implementation

- **Merit of our algorithm BP-Match**
  - **Simple and Efficient:** BP-Match is particularly efficient for expression of small depth and regular structure. Therefore, it is suitable to applications such as NIDS and ESP.
  - **Hardware friendly:** BP-Match is suitable to modern parallel hardwares, such as GPGPUs and FPGA since it uses only simple Boolean and arithmetic operations (+, –) avoiding the heavy use of table-lookup.
- **Implementation**
  - We implemented our algorithm in Verilog HDL.
  - The update formulas of bit-parallel matching are transformed into a fixed circuitry on FPGA in advance.
  - In preprocessing, bitmasks are loaded to block RAMs on FPGA. In run-time, a pattern TNFA is simulated by the circuitry.

NIDS = Network Intrusion Detection System; ESP = Event Stream Processing

Hardware implementation

- Experimantal settings
  - The Verilog code is compiled and simulated on Xilinx Virtex-5 FPGA LX330 (51,840 slices and 1MB block RAMs) using Xilinx ISE Design Suite and Synopsys VCS.
- Expertiment results
  - 128 patterns of length 32 can be installed on the FPGA
  - The FPGA (0.5GHz clock) achieves high throughput (0.5Gbps)

Algorithm	Class	#pat	#op	#add	throughput
Our algorithm [This paper]	**EXNET**	128	20	9	**0.5 Gbps**

#op, **#add**, **#reg**, and **#bram** are the number of 32-bit operaions, 32-bit integer additions, 32-bit registers, and block RAM lines, respectively

Thank you

21st International Workshop on Combinatorial Algorithms (IWOCA'10)

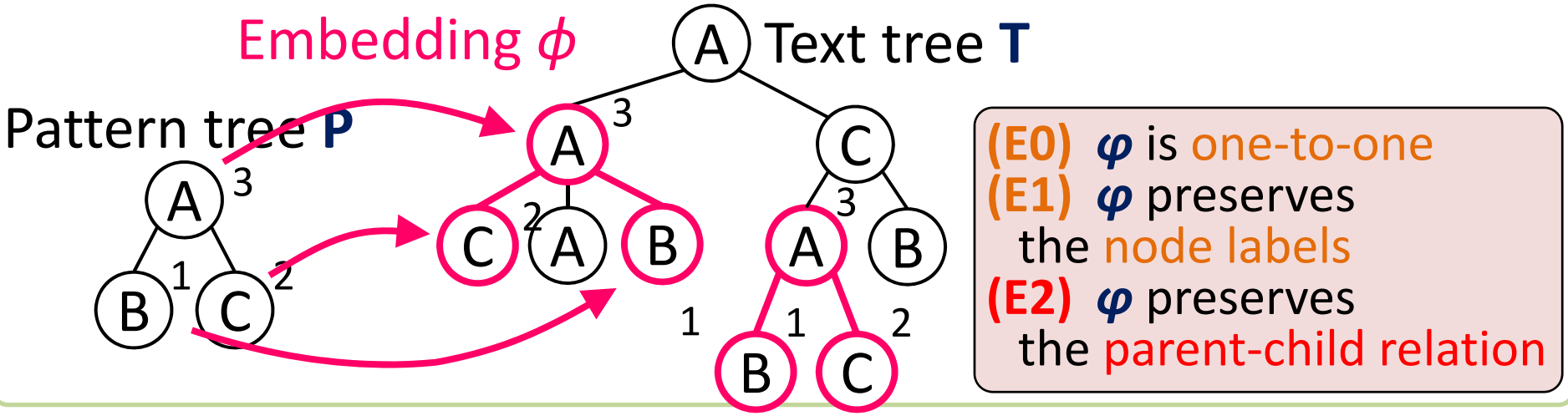
Faster Bit-Parallel Algorithms for Unordered Pseudo-Tree Matching and Tree Homeomorphism

Yusaku Kaneta and Hiroki Arimura

Graduate School of Information Sci. and Tech.
Hokkaido University, Japan

Background: Tree matching problem

- Problem of finding an embedding φ from a pattern tree **P** to a text tree **T**
- Fundamental problem in computer science [Kilpelainen & Mannila, '94]
- It has many applications
- We consider unordered tree matching and its variants (for labeled, rooted tree)



(E0) φ is one-to-one
(E1) φ preserves the node labels
(E2) φ preserves the parent-child relation

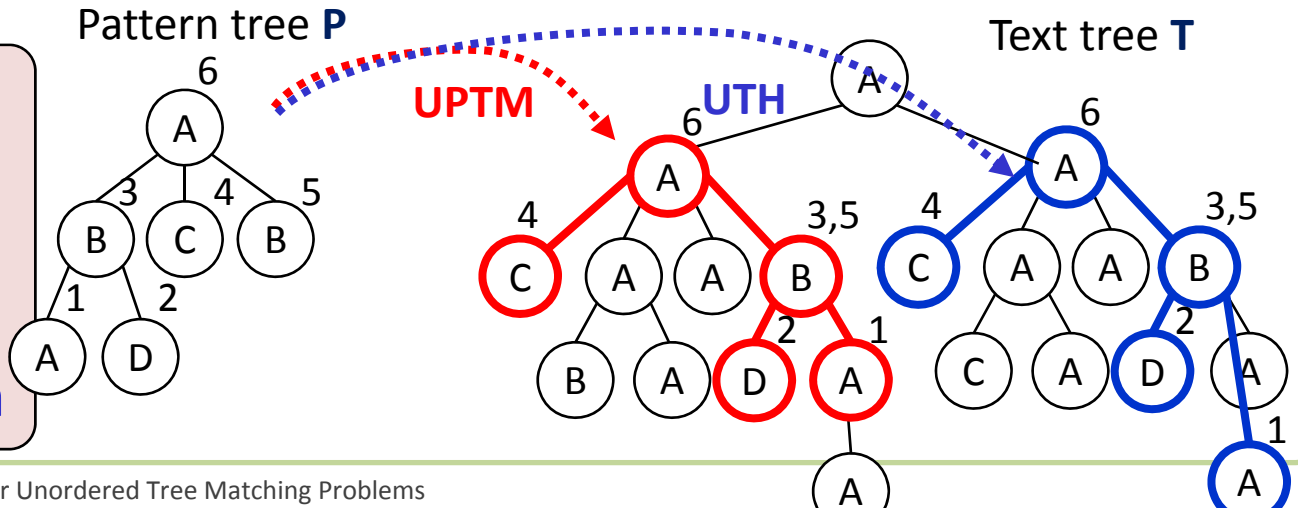
Background: Many-to-one matching

- In original theoretical studies:
  Tree matching with one-to-one mapping has been mainly studied so far
- In recent practical studies: Tree maching with many-to-one mapping attracts much attention
- **Goal:** To develop efficient algorithms for two tree matching problems with many-to-one mappings
  - Unordered pseudo-tree matching problem (**UPTM**)
    ⇔ XPath queries with child axis only
  - Unordered tree homeomorphism problem (**UTH**)
    ⇔ XPath queries with descendant axis only

Definition

- **Unorderd pseudo-tree matching problem (UPTM)**
  - A pattern tree **P** matches a text tree **T** if there is a many-to-one mapping φ: **V(P)** $\rightarrow$ **V(T)** from **P** into **T** satisfying the conditions **(E1)** and **(E2)**
  - An occurrence of **P** in **T** is the image of the root of **P**
  - The problem is to find all occurrences of P in T
- Unorderd tree homeomorphism problem (UTH)
  - is defined similarly, where many-to-one mapping satisfying **(E1)** and **(E3)** is used.

φ preserves:
(E1) the node labels
(E2) the parent-child relation
(E3) the ancestor-descendant relation



Related work

- **Many studies for tree matching with one-to-one mappings**
  - **[Kilpelainen, Mannila, SIAM J'95]**:
    The unordered tree matching and inclusion problems
  - Corresponds to the subgraph isomorphism problem
- **Few studies for tree matching with many-to-one mappings**
  - **[Yamamoto, Takenouchi, WADS'09]** **UPTM problem**
    - **$O(nr \cdot leaves(P) \cdot depth(P)/w) = O(nm^3/w)$** time
    - **$O(n \cdot leaves(P) \cdot depth(P)/w) = O(nm^2/w)$** space
  - **[Gotz, Koch, Martens, DBPL'07]** **UTH problem**
    - **$O(nm \cdot depth(P)) = O(nm^2)$** time
    - **$O(depth(T) \cdot branch(T)) = O(n^2)$** space

m: the size of P, **n**: the size of T, **h**: the height of T, **w**: the word length, and
r: the maximum number of the same label on paths in P

Our results

- **New decomposition formula** for unordered pseudo-tree matching problem (UPTM)
- **Bit-parallel algorithm** for UPTM that runs in
  - **O(nmlog(w)/w)** time
  - **O(hm/w + mlog(w)/w)** space
  - **O(mlog(w))** preprocessing time
- Key: Fast bit-parallel computation of Tree aggregation in **O(log m)** time
  - Improves a naïve implementation in **O(m)** time
- Modified algorithm for **UTH** with the same complexity

m: the size of P, **n**: the size of T, **h**: the height of T, **w**: the word length

Summary

Algorithm for UPTM	Time	Space (in words)
BP-MatchUPTM (this work)	$O(nm\log(w)/w)$	$O(hm/w + m\log(w)/w)$
[Yamamoto, Takenouchi, WADS'09]	$O(nm^3/w)$	$O(nm^2/w)$

- Our algorithm improves the algorithm by [YT'09] (by $O(m^2/\log(w))$)

Algorithm for UTH	Time	Space (in words)
BP-MatchUTH (this work)	$O(nm\log(w)/w)$	$O(hm/w + m\log(w)/w)$
[Gotz, Koch, Martens, DBPL'07]	$O(nm^2)$	$O(hn)$

- Our algorithm improves the algorithm by [Gotz et al.'07]
- This is the first bit-parallel algorithm for UTH (by $O(mw/\log(w))$)

m: the size of P, **n**: the size of T, **h**: the height of T, **w**: the word length

[Yamamoto, Takenouchi, WADS'09] H. Yamamoto and D. Takenouch, Bit-parallel tree pattern matching algorithms for unordered labeled trees, In *Proc*. WADS'09, 554-565, 2009.

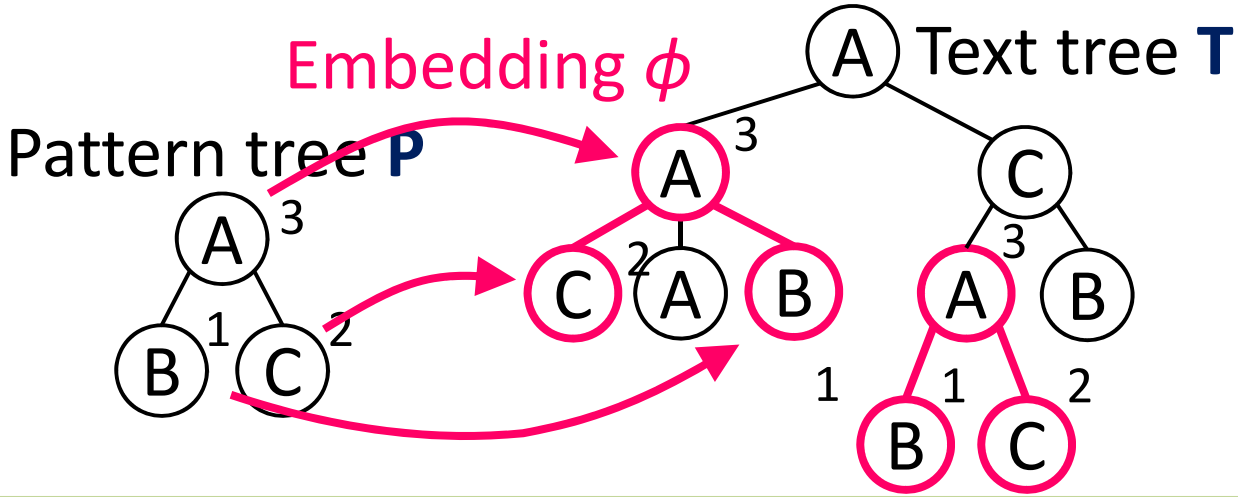
[Gotz, Koch, Martens, DBPL'07] M. Gotz, C. Koch, and W. Martens, Efficient algorithms for tree homeomorphism problem, In *Proc*. DBPL'07, 17-31, 2007.

Algorithm
for the UPTM problem

Our algorithm MatchUPTM

Consists of two components:

1. **New decomposition formula** for bottom-up computataion
2. **Bit-parallel implementation** of five set operations: **Constant, Union , Member , LabelMatch$_P$,** and **TreeAggr$_P$**
   - **Especiallly, O(log m)** time bit-parallel implementation of **TreeAggr operation**



Decomposition formula for UPTM

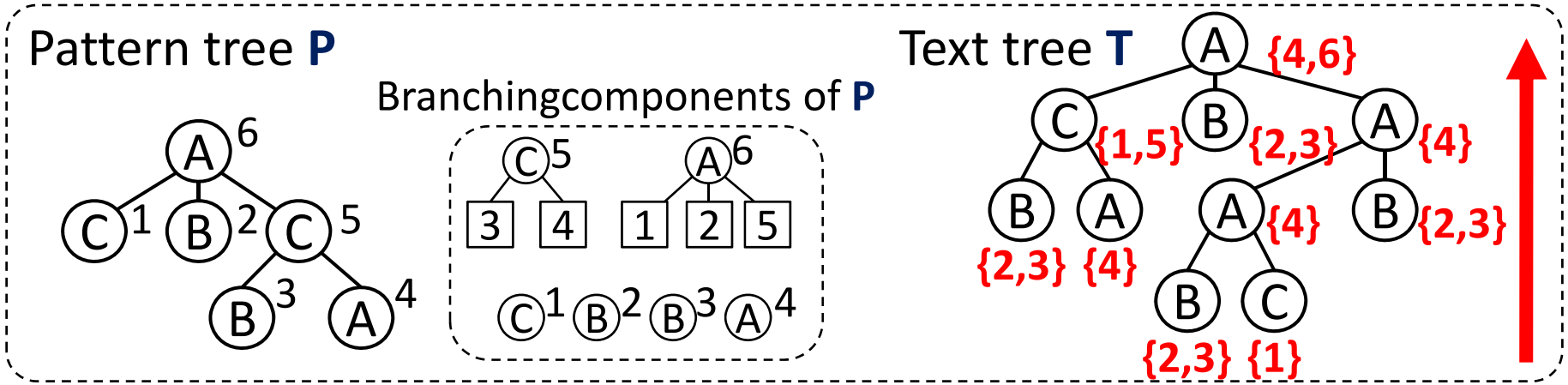
- The embedding set $\mathbf{Emb^{P,T}(v)}$ of text node $\mathbf{v \in V(T)}$
  - is the set of pattern node $\mathbf{x \in V(P)}$ such that P(x), the subtree of **P** rooted at **x**, occurs in **T** at node **v**

Lemma 1 (decomposition formula): For any $\mathbf{x \in V(P)}$, $\mathbf{v \in V(T)}$, $\mathbf{x \in Emb^{P,T}(v)}$

$\Leftrightarrow$ (i) Label matching: $\mathbf{label_P(x) = label_T(v)}$ and

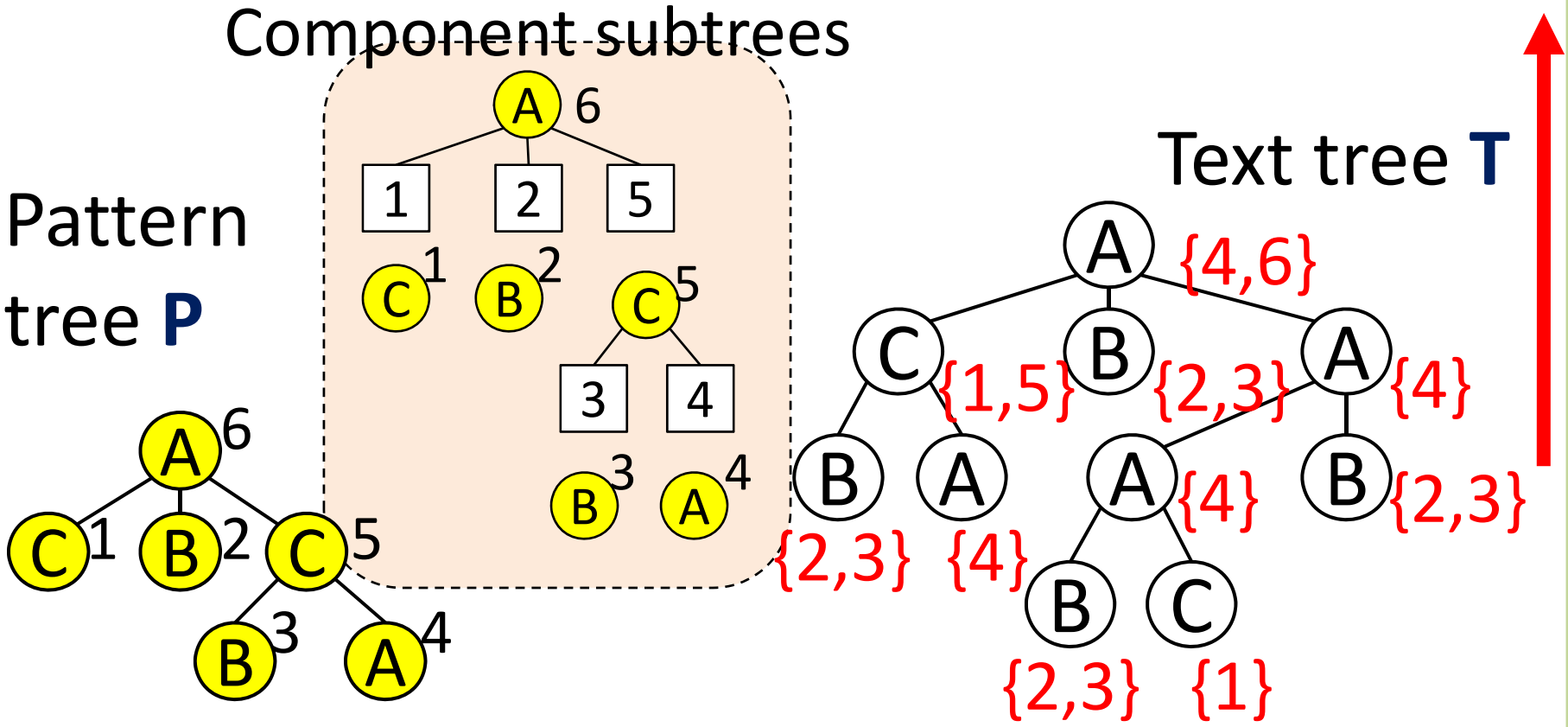
(ii) Tree aggregation: $\mathbf{children(x) \subseteq \bigcup_{1 \le j \le \alpha(v)} Emb^{P,T}(v[j])}$

- From Lemma1, we can develop a bottom-up algorithm for UPTM in $\mathbf{O(nm)}$ time and $\mathbf{O(hm)}$ space, where **h** is the height of **T**



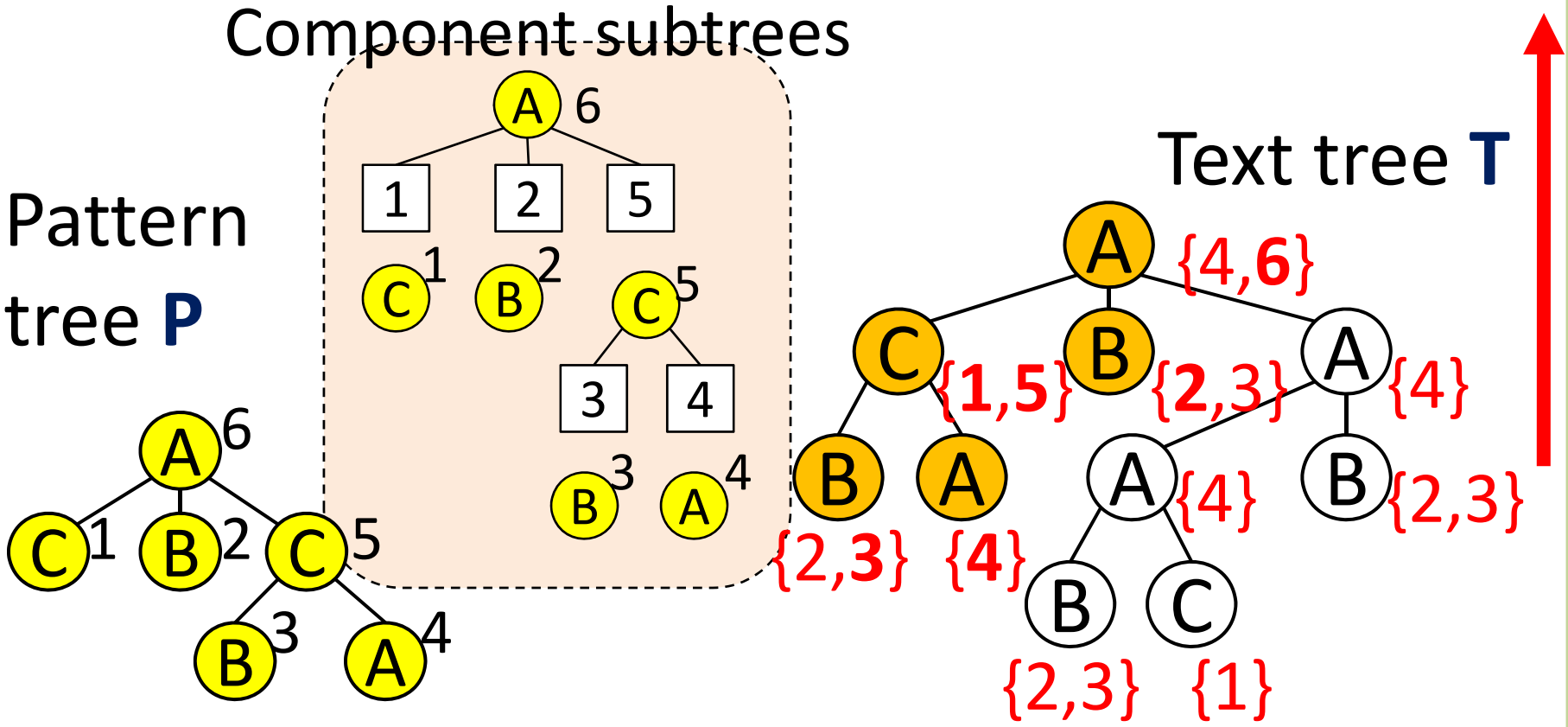
Basic idea: bottom-up algorithm

- Traverses the text tree T from the leaves to the root.
- For each text node **v,** compute the set **S** consisting of the IDs of **matched subtrees** whose roots match to **v**.



Basic idea: bottom-up algorithm

- Traverses the text tree T from the leaves to the root.
- For each text node **v,** compute the set **S** consisting of the IDs of **matched subtrees** whose roots match to **v**.



Outline of our tree matching algorithm

algorithm MatchUPTM($P[1..m]$: a pattern tree, $T[1..n]$: a text tree):
Global Variables: P and T;
Output: all occurrences of P in T w.r.t. unordered pseudo-tree matching (UPTM);
1: VisitUPTM($root(T)$);

procedure VisitUPTM(v: a text node)
Return Value: $R = Emb^{P,T}(v)$;
2: $S \leftarrow$ Constant($\emptyset$); {See Definition 2}
3: **for** $i = 1, \ldots \alpha(v)$ **do**
4: $S \leftarrow$ Union(S, VisitUPTM($v[i]$));
5: $R \leftarrow$ Constant($[1..m]$);
6: $R \leftarrow$ LabelMatch$_P(R, label_T(v))$; {See Definition 2}
7: $R \leftarrow$ TreeAggr$_P(R, S)$; {See Definition 2}
8: **if** Member($R, root(P)$) **then** {See Definition 2}
9: **output** "*A match is found at a node v.*";
10: **return** R;

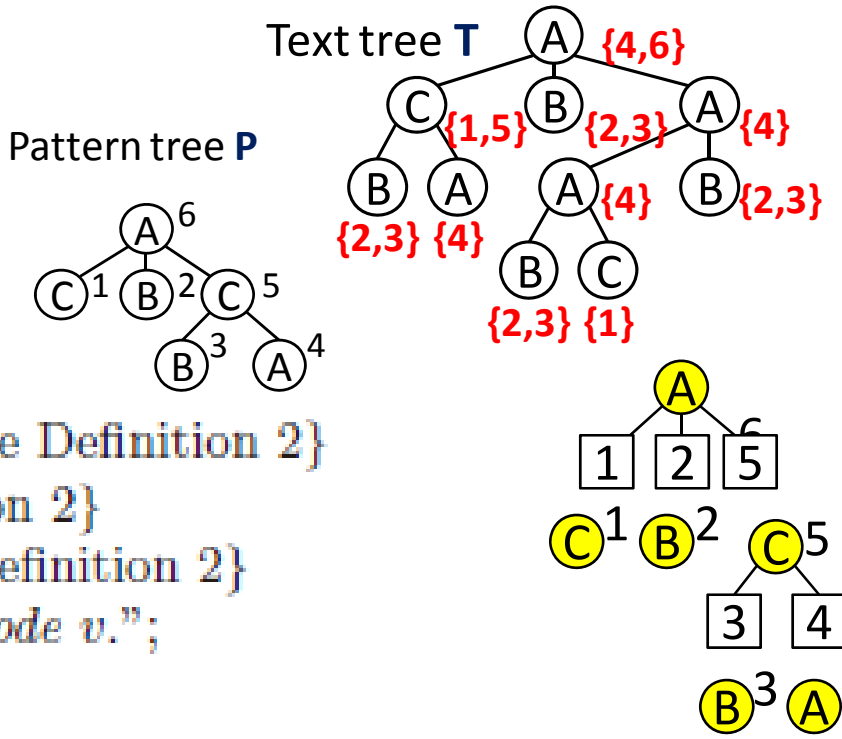


Fig. 2. An algorithm for the unordered pseudo-tree matching problem.

Bit-parallel implementation

- To obtain further speed-up, we use bit-parallelism
  - Encoding an embedding set **Emb(v)** $\subseteq \{1,\ldots,m\}$ for each node **v** by a bitmask **X** $\in \{0,1\}^m$ of length **m**.
  - By implementing the five set operations by using Bit-wise Boolean operations &, |, ~ and integer addition + [BGY'92]
- Key: Bit-parallel implementation of **$\mathbf{TreeAggr_P}$**

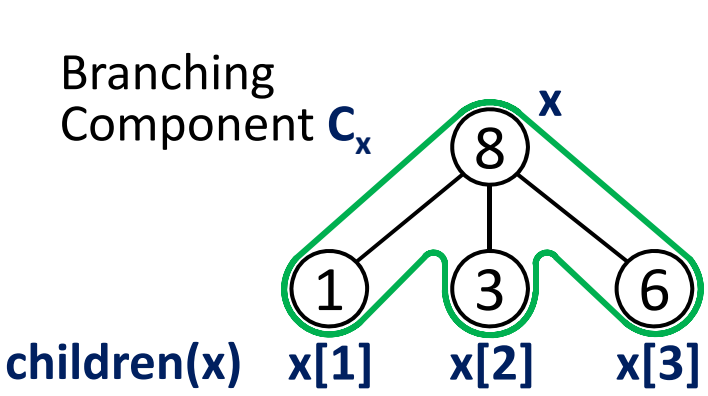
Operation	Original impl.	Bit-parallel impl.	
Constant(S)	**O(m)** time	**O(m/w)** time	Easy
Union(R, S)	**O(m)** time	**O(m/w)** time	
Member(R, x)	**O(m)** time	**O(m/w)** time	
$\mathbf{LabelMatch_P}$(R, α)	**O(m)** time	**O(m/w)** time (From **[BYG92]**)	
$\mathbf{TreeAggr_P}$(R, S)	**O(m)** time	**O(m log(w)/w)** time **(This work)**	**Hard to implement**

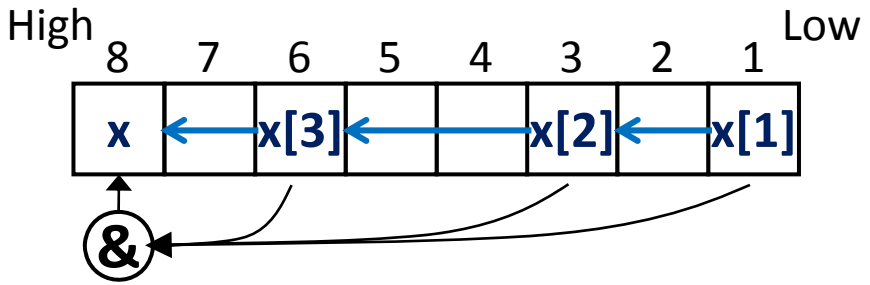
[BYG'92] R. Baeza-Yates and G. H. Gonnet, CACM, 35(10), 74-82, 1992.

m: the size of P, **n**: the size of T, **w**: the word length

Bit-parallel tree aggregation

- Computes the parent value as the logical AND of the children values
- **Preprocess**: Build the following bitmasks
  - **DST**: the position of parent **x**
  - **SRC**: the positions of children **children(x)**
  - **SEED**: the lowest position of component C_x
  - **INT**: the interval of C_x except for **x** and **children(x)**
- **Runtime**: Simulate tree aggregation by bit-operations





Branching omponent C_x: the connected component consisting of parent **x** and its children **children(x)**

Bit-parallel tree aggregation

- **Basic idea: Using the carry propergation by integer addition**
- **Line2**: Compute the **PATH** mask. We fill the "holes" at the children positions in **INT** with the children values in the input mask **Y**.

	x	x[3]			x[2]		x[1]	
COMP	0	0	1	0	0	1	0	1
|) INT	**0**	1	**0**	1	1	**0**	1	**0**
PATH	0	1	**1**	1	1	**1**	1	**1**

holes

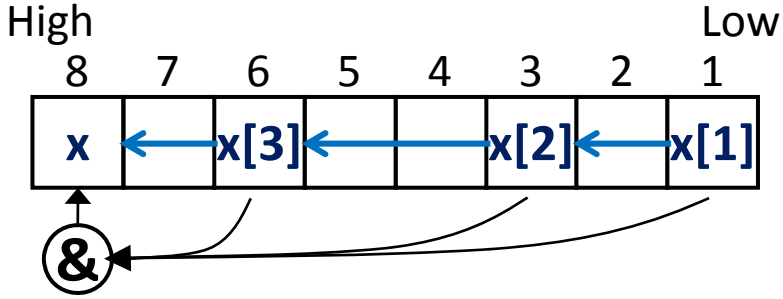
- **Line3**: Compute the **AGGR** mask. If all the "holes" in **PATH** are filled then the parent value is set

	x	x[3]			x[2]		x[1]	
PATH	0	1	1	1	1	1	1	1
+) SEED	0	0	0	0	0	0	0	**1**
AGGR	**1**	0	0	0	0	0	0	0

carries

Runtime:

1. COMP ← Y & SRC;
2. PATH ← COMP | INT ;
3. AGGR ← PATH + SEED;
4. RESULT ← AGGR & DST;



Input:

	x	x[3]			x[2]		x[1]	
Y	0	1	1	0	1	1	0	1

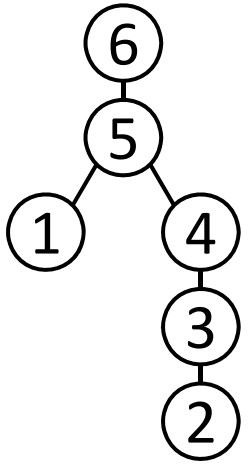
Bitmasks:

DST	**1**	0	0	0	0	0	0	0
SRC	0	0	**1**	0	0	**1**	0	**1**
SEED	0	0	0	0	0	0	0	**1**
INT	**0**	1	**0**	1	1	**0**	1	**0**

Separator tree-based decomposition

- By using **the separator tree-based decomposition technique**, we can implement
  Tree Aggregation in O(log(m)) time
  using O(mlog m) preprocessing time

Pattern tree **P**



Lemma (Jordan , 1869). Let **S** be a binary tree. Then, there exists a node in S such that **|S(v)| ≤ (2/3)|S|** and **|S(v')| ≤ (2/3)|S|**, where **S(v)** is the subtree of **S** rooted at **v** and **S(v')** is the tree obtained by pruning **S(v)** from **S**.

Naïve decomposition:

Bit-position	1	2	3	4	5	6
Node	1	2	3	4	5	6
Level 1					5→6	
Level 2	1→			→4→	5	
Level 3			3→	4		
Level 4		2→	3			

O(m)

Separator tree-based decomposition:

Bit-position	1	2	3	4	5	6
Node	2	3	4	1	5	6
Level 1	1→			→4	5	
Level 2	2→	3				
Level 3			3→	4	5→	6

O(log m)

Bit-assignment also differs from naïve decomposition.

Main result for the UPTM problem

- By appying the module decomposition techniques of **[Myers '92]** and **[Bille '06]**, we have:

Theorem 1. (complexity of the UPTM problem)
The algorithm **BP-MatchUPTM** solves the unordered pseudo-tree matching problem in

- **O(nmlog(w)/w)** time, using
- **O(hm/w + mlog(w)/w)** space and
- **O(mlog(w))** preprocessing time

m: the size of P, **n**: the size of T, **h**: the height of T, **w**: the word length

Note: This improves the time complexity $O(nm^3/w)$ of the previous bit-parallel algorithm by **[Yamamoto &Takenouchi, WADS'09]** with a factor of $O(m^2/\log(w))$

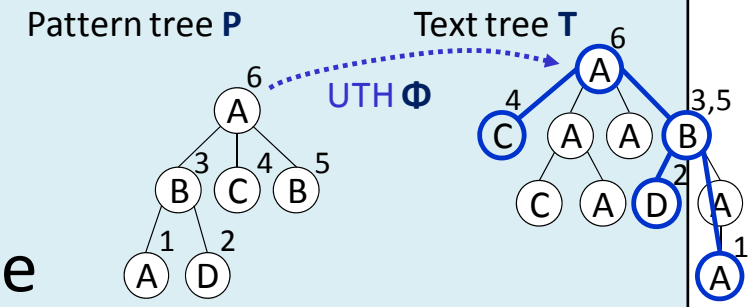
[Bille'06] P. Bille, New algorithms for regular expression matching, In *Proc*. ICALP'06, 643-654, 2006.
[Myers'92] E. W. Myers, A four-russian algorithm for regular expression pattern matching, JACM, 39(2), 430-448, 1992.

Main result for the UTH problem

- Modified Bit-parallel algorithm **BP-MatchUTH**:
  - Based on a similar decomposition formula
  - The code is same as VisitUPTM except line 9

Theorem 2. (complexity of the UTH problem)
The algorithm **BP-MatchUTH** solves the unordered tree homeomorphism problem in

- **O(nmlog(w)/w)** time
- **O(hm/w + mlog(w)/w)** space
- **O(mlog(w))** preprocessing time



m: the size of P, **n**: the size of T, **h**: the height of T, **w**: the word length

Note: This seems **the first bit-parallel algorithm for UTH problem** as far as we know, and It slightly improves the time complexity **$O(nm^2)$** of the algorithm by **[Gotz, Koch, Martens, DBPL'07]** with **a factor of O(mw/log(w))**

Conclusion

- Tree matching with many-to-one mapping
  - **UPTM**: unordered pseudo-tree matching
  - **UTH**: unordered tree homeomorphism
- Bit-parallel algorithms for **UPTM** and **UTH** that run in
  - **O(nmlog(w)/w)** time
  - **O(hm/w + mlog(w)/w)** space
  - **O(mlog(w))** preprocessing
- Future works
  - Extension of this technique for tree matching and inclusion with one-to-one mappings (seems difficult)
  - Applications to practical subclasses of XPath and XQuery languages

Thank you