ビット並列計算を用いた高度なパターン照合

北海道大学
博士後期課程2年

金田 悠作

17th String Processing and Information Retrieval Symposium (SPIRE2010)

Fast Bit-Parallel Matching for Network and Regular Expressions

Yusaku Kaneta, Shin-ichi Minato, and Hiroki Arimura

Graduate School of Information Science and Technology
Hokkaido University, Japan

Background

- Regular expression matching problem
  - Fundamental problem in string processing
- Its new applications have emerged such as
  - NIDS (Network Intrusion Detection Systems)
  - ESP (Event Stream Processing)
- There are demands for large-scale pattern matching systems
  - However, these systems have to cope with ...

2010/11/29 第2回 離散構造処理系シンポジウム：ビット並列計算を用いた高度なパターン照合 - 金田 悠作（北海道大学） 3

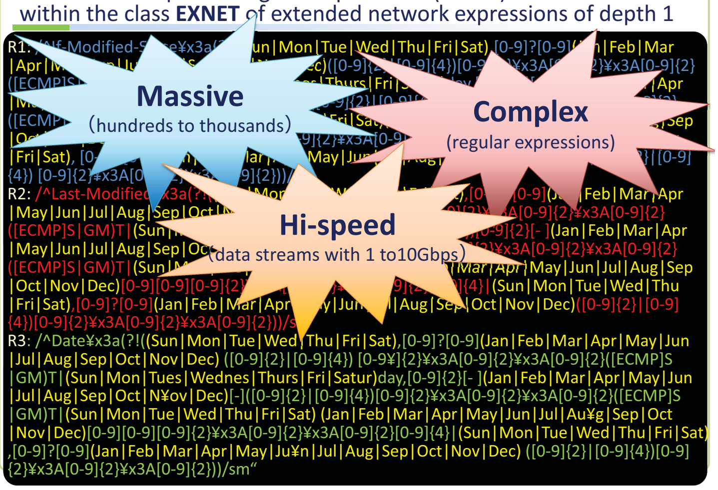
Example of real regular expressions in NIDS applications

Three Perl compatible regular expressions (PCREs) in SNORT 2.8 within the class **EXNET** of extended network expressions of depth 1

```
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“
```

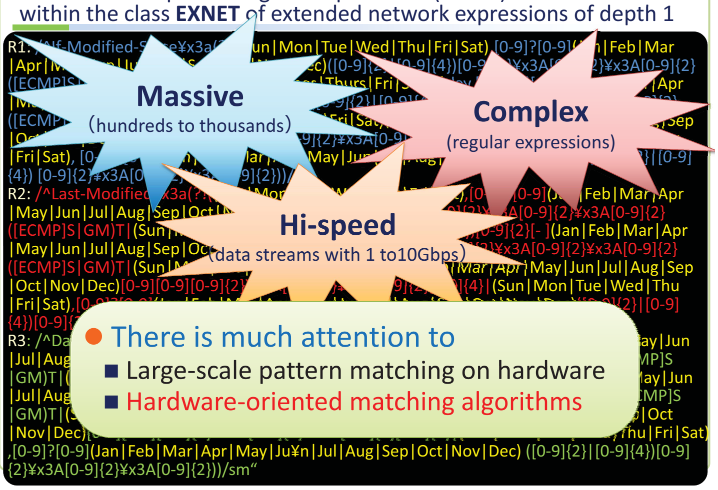
Example of real regular expressions in NIDS applications

Three Perl compatible regular expressions (PCREs) in SNORT 2.8 within the class **EXNET** of extended network expressions of depth 1



Example of real regular expressions in NIDS applications

Three Perl compatible regular expressions (PCREs) in SNORT 2.8 within the class **EXNET** of extended network expressions of depth 1



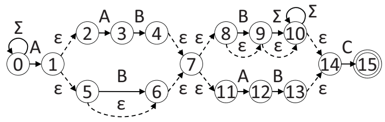
Background

- Large-scale pattern matching systems have to cope with requirements
  - Massive (hundreds to thousands)
  - Complex patterns (regular expressions)
  - High-speed data streams (with 1 to 10 Gbps)
- Hardware-oriented matching algorithms
  - DFA-based, NFA-based, CAM-based ...
  - None of these satisfies all of the above three requirements
- **In this research, we focus on bit-parallel methods for efficient regular expression matching**

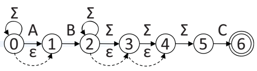
Classes of regular expressions

- We study the regular expression matching problem for the following subclasses of regular expressions
  - **Regular expressions (REG)**
  - **Network expressions (NET)**: The subclass of REG without Kleene-star [Myers, '96]
  - **Extended network expressions (EXNET)**: **(our main target class)** Network expressions over the class **EXT** of extended strings
    - which are concatenations of letter α, class of letters [ab...], optional letter a?, bounded repeat a{lo, hi}, and unbounded repeats a* and a+.
  - These subclasses are widely used in real world applications such as NIDS and ESP

Extended network expressions
R = A(AB|B?)(B?.*|AB)C



Bit-Parallel method



Extended string R = A?B+.{1,3}C

SHIFT-AND approach

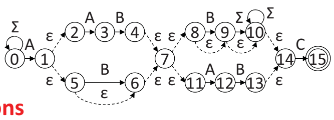
- **SHIFT-AND method** [Baeza-Yates & Gonnet, '92]
  - Bit-parallel matching algorithm for the class **STR** of strings
  - Boolean (&, |) operations only
  - Simple and efficient
- **Extended SHIFT-AND method** [Navarro & Raffinot, '01]
  - A nice extension to the class **EXT** of extended strings
  - Boolean (&, |, ~, ⊕) and arithmetic (-) operations
  - Still simple and efficient
- **Challenge: Expressiveness of matching patterns**
- **Open problem: Can we extend Extended SHIFT-AND method to more general subclasses of REG?**

Related work

- **Bit-parallel Thompson** [Wu & Manber, '92]
  - Bit-parallel matching algorithm for the class **REG**
  - Boolean (&, |) operations
  - Table-lookup
- **Myers's algorithm** [Myers, '92]
  - Matching algorithm for the class **REG**
  - Table-lookup
  - Module decomposition of Thompson NFA
- **Bille's algorithm** [Bille, ICALP2006]
  - Bit-parallel matching algorithm for the class **REG**
  - Boolean (&, |) and arithmetic (-) operations
  - Module decomposition of Thompson NFA

Our results

- We developed **Extended² SHIFT-AND method**
- **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**
- Experimental results on hardware implementation of our algorithm
- **Keys**
  - **Bi-monotonicity lemma** for Thompson NFA
  - Bit-parallel operations **Scatter**, **Gather**, and **Propagate** for simulating ε-moves

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

Comparison: Time & Space complexities

- Our Extended² SHIFT-AND method is the first extension of SHIFT-AND and Extended SHIFT-AND methods to the following classes:
  - Extended network expressions (**EXNET**)
  - Regular expressions (**REG**)

Algorithm	Class	Time	Space (in words)
SHIFT-AND [Baeza-Yates & Gonnet, '92]	**STR**	**O(nm/w)**	**O(m/w)**
Extended SHIFT-AND [Navarro & Raffinot, '01]	**EXT**	**O(nm/w)**	**O(m/w)**
Extended² SHIFT-AND (in this talk)	**EXNET**	**O(ndm/w)**	**O(dm/w)**
	REG	**O(ndlog(m)m/w)**	**O(dlog(m)m/w)**

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

Hardware implementation

- **Merit of our Extended² SHIFT-AND method**
  - **Simple and Efficient**: Our algorithm is particularly efficient for expression of small depth and regular structure. Therefore, it is suitable to applications such as NIDS and ESP.
  - **Hardware friendly**: Our algorithm is suitable to modern parallel hardwares, such as FPGA and GPGPU since it uses only simple Boolean and arithmetic operations (+, –) avoiding the heavy use of table-lookup.

FPGA = Field Programmable Gate Array
GPGPU = General Purpose GPU

- **Implementation**
  - We implemented our algorithm on FPGA.
  - The update formulae 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 runtime, The NFA equivalent to a given expression is simulated by the circuitry.

Hardware implementation

- **Experimantal settings**
  - Our circuit written in Verilog HDL is compiled and simulated on Xilinx Virtex-5 FPGA LX330 (51,840 slices and 1 MB block RAMs) using Xilinx ISE Design Suite and Synopsys VCS.
- **Expertimental results**
  - 128 patterns of length 32 can be installed on the FPGA
  - Our algorithm implemnted on the FPGA (0.6 GHz clock) achieves the high throughput of 0.5 Gbps.

Algorithm	Class	#pat	#op	#add	Throughput
Extended² SHIFT-AND (in this talk)	EXNET	128	20	9	0.5 Gbps

#pat, **#op**, and **#add** are the number of input expressions, 32-bit Boolean-operaions, and 32-bit integer additions, respectively

Algorithms

Our bit-parallel algorithm

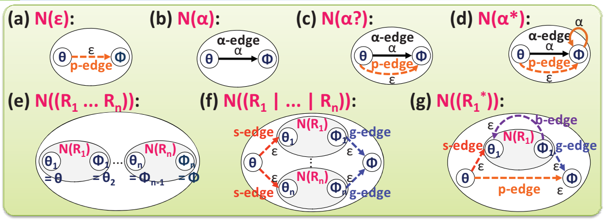
- **Outline of algorithm**

Algorithm Extended² SHIFT-AND

Preprocessing
- Transform an input regular expression **R** to the equivalent NFA N_R (Thompson NFA, TNFA).
- Construct a set of bitmasks for TNFA.

Runtime:
- Simulate TNFA N_R on an input text by using the bitmasks based on bit-parallel method.

Thompson NFA [Thompson, CACM1968]



- A NFA equivalent to a regular expression **R**
  - The source θ and the sink ϕ
- **Depths d(S)** and **d(x)** of subexpressions **S** and states **x**
  - The number of nesting of Union "|"
- TNFA for EXNET has three types of ε-edges
  - Scatter edges, Gather edges, and Propagate edges
- TNFA for REG has one more type of ε-edges
  - Back edges

Simulation of Thompson NFA

- We use bit-parallel method
  - Encode a state set of TNFA in a bitmask **D**
- Procedure RunTNFA:
  - Simulates α-moves $Move_N(D, \alpha)$ in the same way as SHIFT-AND approaches [Baeza-Yates & Gonnet, '92][Navarro & Raffinot, '01]
  - Simulates ε-closure $EpsClo_N(D)$ by combining bit-parallel operations **Scatter**, **Gather**, and **Propagate** [This talk]

```
Procedure RunTNFA(T = t1 … tn: input text)
  D ← InitN;
  D ← EpsCloN(D);
  for t ← t1, …, tn do
    if D & AcceptN ≠ 0 then
      output match i-1;
    D ← MoveN(D, t);
    D ← EpsCloN(D);
  end for
  return D;
```

$Move_N(D, \alpha) \equiv D \leftarrow (((D \ll 1)\ \&\ CHR[\alpha])\ |\ 1)\ |\ (D\ \&\ REP[t]);$

$Init_N \equiv 0^{m-1}1;$

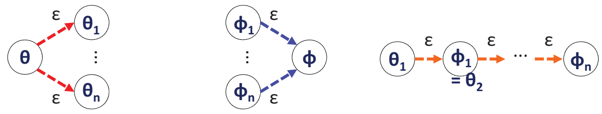
$Accept_N \equiv 10^{m-1};$

In the following, we assume that a constant alphabet Σ, where |Σ| = O(1)

Our bit-parallel algorithm

- **Key:** Computation of ε-closure **EpsClo$_N$(D)**
- Consists of two components:

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

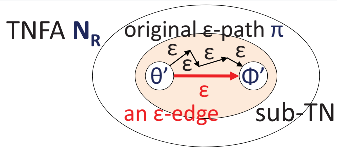


Scatter **Gather** **Propagate**

Bimonotonicity lemma

- Let **N$_R$** = (V, E, θ, ϕ) be a TNFA with source θ and sink ϕ

Procedure Bypassing
1. Visit every **sub-TNFA N'** of **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 **N$_R$** by bypassing

Bimonotonicity lemma

- Let x, y be any states in TNFA N$_R$ for **Expand(R)**
- For any states x, y, define d(x) $\le_1$ d(y) iff d(x) - d(y) ≤ 1

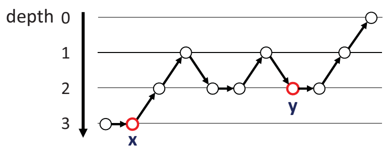
Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path π' = (x$_1$ = x, ... , x$_n$ = y) from **x** to **y** in N$_R$ such that d(x$_1$) $\ge_1$ ··· $\ge_1$ d(x$_k$) and d(x$_k$) $\le_1$ ··· $\le_1$ d(x$_n$) (Eq1)

Errata: There is **a mistake** in the definition of the bimonotonicity in Page 379 of the conference proceedings. **The direction of inequality $\le_1$ is opposite to the correct one :-)**
Please correct it as in the above (Eq1).

Bimonotonicity lemma

- Let x, y be any states in TNFA N$_R$ for **Expand(R)**
- For any states x, y, define d(x) $\le_1$ d(y) iff d(x) - d(y) ≤ 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 π' = (x$_1$ = x, ... , x$_n$ = y) from **x** to **y** in N$_R$ such that d(x$_1$) $\ge_1$ ··· $\ge_1$ d(x$_k$) and d(x$_k$) $\le_1$ ··· $\le_1$ d(x$_n$)

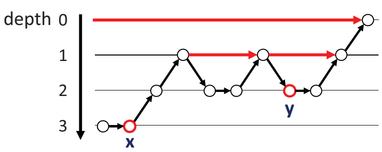


Step 0.
Suppose we have a TNFA **N$_R$** with an ε-path π from state **x** to state **y**

Bimonotonicity lemma

- Let x, y be any states in TNFA N$_R$ for **Expand(R)**
- For any states x, y, define d(x) $\le_1$ d(y) iff d(x) - d(y) ≤ 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 π' = (x$_1$ = x, ... , x$_n$ = y) from **x** to **y** in N$_R$ such that d(x$_1$) $\ge_1$ ··· $\ge_1$ d(x$_k$) and d(x$_k$) $\le_1$ ··· $\le_1$ d(x$_n$)

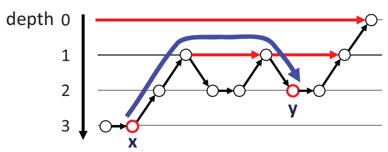


Step 1.
Applying bypassing transformation to the TNFA **N$_R$**

Bimonotonicity lemma

- Let x, y be any states in TNFA N$_R$ for **Expand(R)**
- For any states x, y, define d(x) $\le_1$ d(y) iff d(x) - d(y) ≤ 1

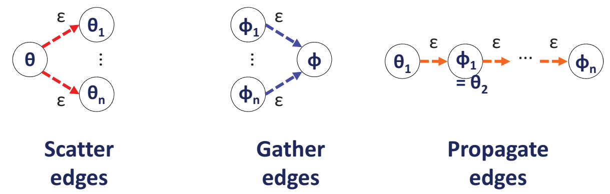
Lemma 1. (Bi-monotonicity lemma)
If there is an ε-path π from **x** to **y**, then there also exists some bi-monotone ε-path π' = (x$_1$ = x, ... , x$_n$ = y) from **x** to **y** in N$_R$ such that d(x$_1$) $\ge_1$ ··· $\ge_1$ d(x$_k$) and d(x$_k$) $\le_1$ ··· $\le_1$ d(x$_n$)



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

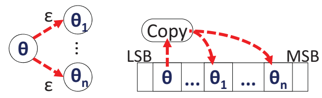
Bit-parallel implementation

- **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 operation

- **Basic idea**: carry propagation of integer subtraction
  (The origin of using carry propagation is due to [Navarro & Raffinot, '01])



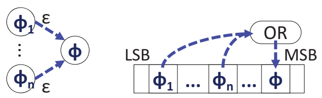
	1	0	0	0	0	0	0	0
−)	0	0	0	0	0	0	0	1
	0	1	1	1	1	1	1	1

Preprocess: for depth **k**, we construct the following bitmasks
- $BLK_S[k]$: the state $j = θ_n + 1$
- $SRC_S[k]$: the source $j = θ$
- $DST_S[k]$: the destinations $j \in \{θ_1, ..., θ_n\}$

Runtime: for 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])



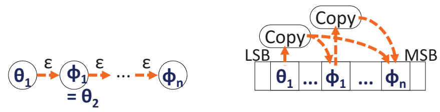
	0	0	1	0	1	0	1	0
+)	0	1	1	1	1	1	1	1
	1	0	1	0	1	0	1	1

Preprocess: for depth **k**, we construct the following bitmasks
- $BLK_G[k]$: the states **j** in interval $[Φ_1..Φ\text{-}1]$
- $SRC_G[k]$: the sources $j \in \{Φ_1, ..., Φ_n\}$
- $DST_G[k]$: the destination $j = Φ$

Runtime: for state set D and depth **k**, we perform
- **Gather(D, k)** ≡ { $D \leftarrow D \mid ((BLK_G[k] + (D \,\&\, SRC_G[k])) \,\&\, DST_G[k]$; }

Propagate operation

- **Basic idea**: Propagate operation of Extended SHIFT-AND for the ε-closure of a set of ε-blocks [Navarro & Raffinot, '01]
  - An ε-block is a maximal consecutive sequence of ε-edges.



Preprocess: for depth **k**, we construct the following bitmasks
- $BLK_P[k]$: the states **j** in ε-block $B = \{θ_1, Φ_1 = θ_2, \cdots, Φ_n\}$
- $SRC_P[k]$: the least significant state $j = \min(B) = θ_1$ in ε-block **B**
- $DST_P[k]$: the most significant state $j = \min(B) = Φ_n$ in ε-block **B**

Runtime: for state set D and depth **k**, we perform
- **Propagate(D, k)** ≡ { $A \leftarrow (D \,\&\, BLK_P[k]) \mid DST_P[k]$;
  $D \leftarrow D \mid (BLK_P[k] \,\&\, ((\sim(A - SRC_P[k])) \oplus A))$; }

Putting them together

- Scatter, Gather, and Propagate can be computable in O(m/w) time using O(m/w) space and O(m) preprocessing.
- The following procedure correctly computes the ε-closure $EpsClo_N(D)$ in O(dm/w) time using O(dm/w) space and O(dm) preprocessing, where d = d(R).

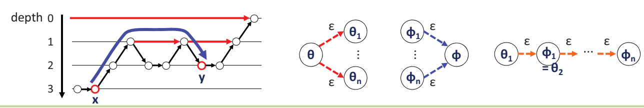
```
Procedure EpsClo_N(D: a state set of TNFA N_R)
for k ← d(R), ..., 1 do
   D ← Propagate(D, k);
   D ← Gather(D, k-1);
end for
D ← Propagate(D, 0);
for k ← d(R), ..., 1 do
   D ← Scatter(D, k-1);
   D ← Propagate(D, k);
end for
return D;
```


Main result for the class EXNET

- Combining $Move_N$ and $EpsClo_N$ with our algorithm Extended² SHIFT-AND, we have:

Theorem 1. Our bit-parallel algorithm for **EXNET** solves the regular expression matching problem for **EXNET** 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 an extension of our 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(m)m/w)** time
- **O(dlog(m)m/w)** space
- **O(dlog(m)m)** preprocessing

m: size of R, **n**: size of T, **d**: depth of T, **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 m)** term with **O(1)**
- If the O(1)-bit-reversal operation is available, then we can also replace the **O(log m)** term with **O(1)**

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**
- We implemented our algorithm on FPGA, and achieves the high throughtput of **0.5 Gbps**
- Future works:
  - Tree and XML matching

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

Thank you

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

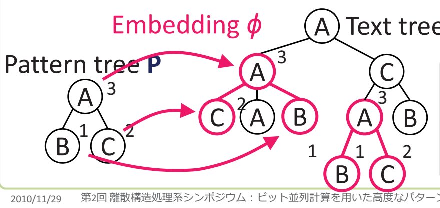
Faster Bit-Parallel Algorithms for Unordered Pseudo-Tree Matching and Tree Homeomorphism

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

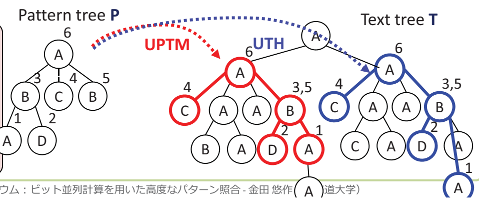
Background: Many-to-one matching

- 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) → 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·leaves(P)·depth(P)/w) = O(nm³/w)** time
    - **O(n·leaves(P)·depth(P)/w) = O(nm²/w)** space
  - **[Gotz, Koch, Martens, DBPL'07] UTH problem**
    - **O(nm·depth(P)) = O(nm²)** time
    - **O(depth(T)·branch(T)) = O(n²)** 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(nmlog(w)/w)	O(hm/w + mlog(w)/w)
[Yamamoto, Takenouchi, WADS'09]	O(nm³/w)	O(nm²/w)

- Our algorithm improves the algorithm by [YT'09] (by O(m²/log(w)))

Algorithm for UTH	Time	Space (in words)
BP-MatchUTH (this work)	O(nmlog(w)/w)	O(hm/w + mlog(w)/w)
[Gotz, Koch, Martens, DBPL'07]	O(nm²)	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

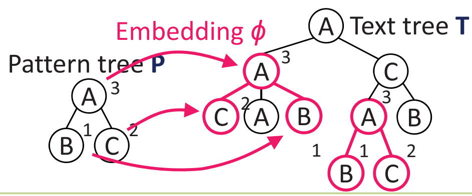
[Yamamoto, Takenouchi, WADS'09] H. Yamamoto and D. Takenouch, Bit-parallel tree pattern matching algorithms for unordered labeled trees, In *Proc.* WADS'09, 554-565, 2009.
[Gotz, Koch, Martens, DBPL'07] M. Gotz, C. Koch, and W. Martens, Efficient algorithms for tree homeomorphism problem, In *Proc.* 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**

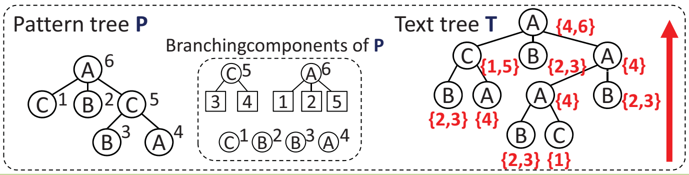


Decomposition formula for UPTM

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

Lemma 1 (decomposition formula): For any $x \in V(P)$, $v \in V(T)$, $x \in Emb^{P,T}(v)$
⇔ (i) Label matching: $label_P(x) = label_T(v)$ and
(ii) Tree aggregation: $children(x) \subseteq \bigcup_{1 \le j \le \alpha(v)} Emb^{P,T}(v[j])$

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



Bit-parallel implementation

- To obtain further speed-up, we use bit-parallelism
  - Encoding an embedding set **Emb(v)** ⊆ {1,...,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 $TreeAggr_P$

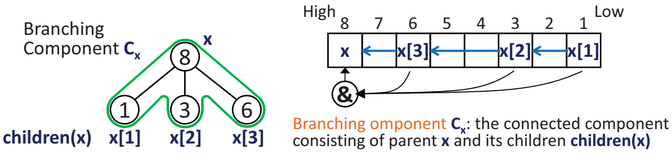
Operation	Original impl.	Bit-parallel impl.	
Constant(S)	O(m) time	O(m/w) time	Easy
Union(R, S)	O(m) time	O(m/w) time	
Member(R, x)	O(m) time	O(m/w) time	
$LabelMatch_P(R, \alpha)$	O(m) time	O(m/w) time (From **[BYG92]**)	
$TreeAggr_P(R, S)$	O(m) time	O(m log(w)/w) time (This work)	Hard to implement

[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

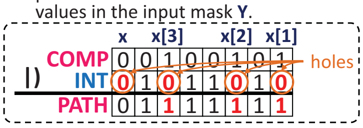
- Computes the parent value as the logical AND of the children values
- **Preprocess**: Build the following bitmasks
  - **DST**: the position of parent **x**
  - **SRC**: the positions of children **children(x)**
  - **SEED**: the lowest position of component C_x
  - **INT**: the interval of C_x except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



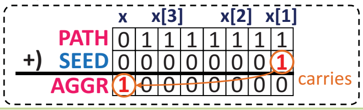
Branching omponent C_x: the connected component consisting of parent **x** and its children **children(x)**

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**.

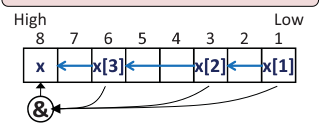


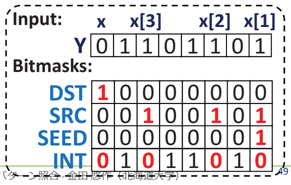
  - **Line3**: Compute the **AGGR** mask. If all the "holes" in **PATH** are filled then the parent value is set



Runtime:

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

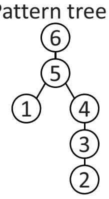




Separator tree-based decomposition

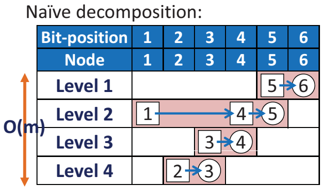
- By using **the separator tree-based decomposition technique**, we can implement
  Tree Aggregation in O(log(m)) time
  using O(mlog m) preprocessing time

Pattern tree **P**

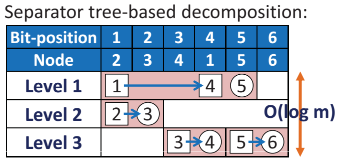


Lemma (Jordan , 1869). Let **S** be a binary tree. Then, there exists a node in S such that **|S(v)| ≤ (2/3)|S|** and **|S(v')| ≤ (2/3)|S|**, where **S(v)** is the subtree of **S** rooted at v and **S(v')** is the tree obtained by pruning **S(v)** from **S**.

Naïve decomposition:

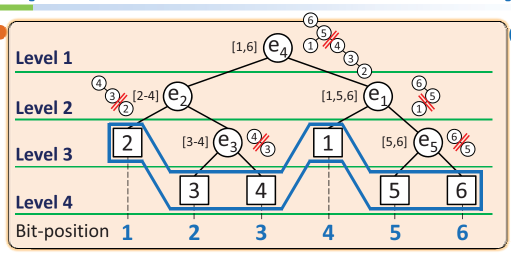


Separator tree-based decomposition:



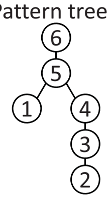
Bit-assignment also differs from naïve decomposition.

Separator tree-based decomposition

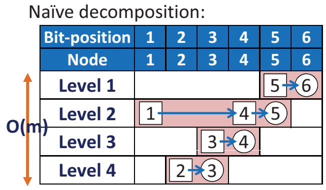


osition

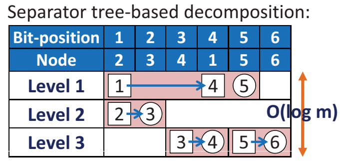
Pattern tree **P**



Naïve decomposition:



Separator tree-based decomposition:



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

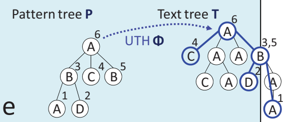
Note: This improves the time complexity **$O(nm^3/w)$** of the previous bit-parallel algorithm by **[Yamamoto &Takenouchi, WADS'09]** with a factor of **$O(m^2/log(w))$**

[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