Regular Languages and Applications

Yo-Sub Han

Department of Computer Science

Yonsei University
Regular Languages

- An old and well-known topic in CS
- Kleene Theorem in 1959
- FA (finite-state automaton) constructions: Thompson automata, position automata in 1960s
- Pattern Matching Problem in 1970s
- ............. in 1980s
- REVISIT State Complexity, Prime Decomposition, Pattern Matching since mid 1990s
Regular Languages

- An old and well-known topic in CS
- Kleene Theorem in 1959
- FA (finite-state automaton) constructions: Thompson automata, position automata in 1960s
- Pattern Matching Problem in 1970s
- ... in 1980s
- REVISIT State Complexity, Prime Decomposition, Pattern Matching since mid 1990s
  - XML, Bioinformatics - New Applications
Overview

- Basic Notions
- Position Construction and XML DTD
- Regular-Expression Pattern Matching
- State Complexity
- Future Directions and Conclusions
Regular Expressions

Regular expressions are a very convenient form that represents (infinite) sets of strings called regular sets.

Given a finite alphabet $\Sigma$, a regular expression over $\Sigma$ is defined recursively as follows:

1. $\emptyset$, the empty-set symbol, is a regular expression.
2. $\lambda$, the empty-string symbol, is a regular expression.
3. $a \in \Sigma$ is a regular expression.
4. $E + F$ (union), where $E$ and $F$ are regular expressions, is a regular expression.
5. $E \cdot F$ (catenation), where $E$ and $F$ are regular expressions, is a regular expression.
6. $E^*$ (Kleene star), where $E$ is a regular expression, is a regular expression.
Finite-state Automata (FAs)

A finite-state automaton $\mathcal{A}$ is specified by a tuple $(Q, \Sigma, \delta, s, F)$;

- $Q$ a finite set of states
- $\Sigma$ a finite alphabet
- $\delta(p, a) = q$ a set of transition rules
- $s \in Q$ the start state
- $F \subseteq Q$ a set of final states
Finite-state Automata - example

$s = q_1$

$F = \{q_9, q_{10}\}$
Finite-state Automata - example

\[ T = aaabbabbb \]

\[ s = q_1 \]

\[ F = \{ q_9, q_{10} \} \]
Finite-state Automata - example

\[ T = \check{aaabbabbb} \]

\[ Q' = \{ q_2 \} \]
Finite-state Automata - example

\[ T = \text{aaabbabbabb} \]
\[ Q' = \{ q_2 \} \]

\[ s = q_1 \]
\[ F = \{ q_9, q_{10} \} \]
Finite-state Automata - example

\[ T = aaabbabbb \]

\[ s = q_1 \]
\[ F = \{ q_9, q_{10} \} \]
Finite-state Automata - example

\[ T = aaabbabbbb \]
\[ Q' = \{q_3\} \]
Finite-state Automata - example

\[ T = aaabbabb \]
\[ Q' = \{q_4, q_6\} \]
Finite-state Automata - example

$T = aaabbabbb$

$Q' = \{q_7\}$

$s = q_1$

$F = \{q_9, q_{10}\}$
Finite-state Automata - example

\[ T = aaabbabbbb \]

\[ Q' = \{ q_8, q_9 \} \]
Finite-state Automata - example

\[ T = aaabbbabbb \]
\[ Q' = \{q_9\} \]
Finite-state Automata - example

\[ T = aaabbabbb \checkmark \text{ accepted!!} \]

\[ Q' = \{ q_9 \} \subseteq F \]

\[ s = q_1 \]

\[ F = \{ q_9, q_{10} \} \]
Finite-state Automata - example

\[ L = L(aa^*b(ab^*bab^* + ba(bb^* + ba))) \]

\[ s = q_1 \]
\[ F = \{ q_9, q_{10} \} \]
REs into Finite-state Automata

The well-known Thompson construction by Ken Thompson in 1968.

\[ E = \lambda \]

\[ E = \emptyset \]

\[ E = a \]

\[ E_1 + E_2 \]

\[ E_1 \cdot E_2 \]

\[ E^* \]
REs into Finite-state Automata

The well-known Thompson construction by Ken Thompson in 1968.

- $E = \lambda$
- $E = \emptyset$
- $E = a$
- $E_1 + E_2$
- $E_1 \cdot E_2$
- $E^*$

Easy to understand and build-up
Too many $\lambda$ transitions
Position Automata - another automaton construction

- Proposed by Glushkov and McNaughton and Yamada in 1960 independently.

- The construction is based on the positions of characters of a given regular expression.
Position Automata - an example

\[ E = (a + b)^*c(a + b) \quad E' = (1 + 2)^*3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^*c(a + b) \quad E' = (1 + 2)^*3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^{\ast}c(a + b) \quad E' = (1 + 2)^{\ast}3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^* c(a + b) \quad E' = (1 + 2)^* 3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^*c(a + b) \quad E' = (1 + 2)^*3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^*c(a + b) \quad E' = (1 + 2)^*3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)c(a + b) \quad \quad E' = (1 + 2)3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^*c(a + b) \quad E' = (1 + 2)^*3(4 + 5) \]
Position Automata - an example

\[ E = (a + b)^* c (a + b) \quad E' = (1 + 2)^* 3 (4 + 5) \]
Position Automata

- The construction looks nice!
- All in-transitions of a state have the same label.
- The number of states = $|E| + 1$

Less states than the Thompson automata and, thus usually faster!

$E = (a + b)^* c(a + b)$
Where do position automata lead us?
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is **one-unambiguous** if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^*N$$
One-Unambiguous Regular Languages

Proposed by Brüggemann-Klein and Wood.

A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^*N$$

\[
\text{[S E O U L U L N]}
\]
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.
- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.
- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^*N$$

$S E O U L U L N$
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^*N$$

$S$ $E$ $O$ $U$ $L$ $L$ $L$ $L$ $N$
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

\[ E = SEO(UL)^*N \]

\[ S E O \boxed{U} L U L N \]
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is **one-unambiguous** if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

\[
E = SEO(UL)^*N
\]
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.
- A regular language $L$ is **one-unambiguous** if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.
- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^*N$$

|S|E|O|U|L|U|L|N|
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.
- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.
- Given an one-unambiguous regular expression $E$ and an input string $w$, we can read $w$ using one lookahead with respect to $E$.

$$E = SEO(UL)^* N$$

$S E O U L U L N$
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language \( L \) is one-unambiguous if there is a regular expression \( E \) such that \( L = L(E) \) and the position automaton of \( E \) is deterministic.

- Given an one-unambiguous regular expression \( E \) and an input string \( w \), we can read \( w \) using one lookahead with respect to \( E \).

\[
E = SEO(UL)^*N
\]
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

- Not all regular expressions are one-unambiguous.

$$E = SEO(UL)^*UNI$$

- Not all regular languages are one-unambiguous. There are some regular languages that cannot be defined by an one-ambiguous regular languages. e.g. $L((a + b)^*a(a + b)^k), k \geq 1$
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.
One-Unambiguous Regular Languages

Proposed by Brüggemann-Klein and Wood.

A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

```xml
<?xml version="1.0"?>
<!DOCTYPE BOOK [ 
  <!ELEMENT p (#PCDATA)> 
  <!ELEMENT BOOK (OPENER, SUBTITLE?, INTRODUCTION?, (SECTION | PART)+)> 
  <!ELEMENT OPENER (TITLE_TEXT)*> 
  <!ELEMENT TITLE_TEXT (#PCDATA)> 
  <!ELEMENT SUBTITLE (#PCDATA)> 
  <!ELEMENT INTRODUCTION (HEADER, p+)+> 
  <!ELEMENT PART (HEADER, CHAPTER+)> 
  <!ELEMENT SECTION (HEADER, p+)> 
  <!ELEMENT HEADER (#PCDATA)> 
  <!ELEMENT CHAPTER (CHAPTER_NUMBER, CHAPTER_TEXT)> 
  <!ELEMENT CHAPTER_NUMBER (#PCDATA)> 
  <!ELEMENT CHAPTER_TEXT (p)+> ]>
```
One-Unambiguous Regular Languages

- Proposed by Brüggemann-Klein and Wood.

- A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

<?xml version="1.0"?>
<!DOCTYPE BOOK [
  <!ELEMENT p (#PCDATA)>
  <!ELEMENT BOOK (OPENER, SUBTITLE?, INTRODUCTION?, (SECTION | PART)+)>>
  <!ELEMENT OPENER (TITLE_TEXT)*>>

BOOK ::= 
  OPENER · (SUBTITLE+λ) · (INTRODUCTION+λ) · (SECTION + PART)(SECTION + PART)*
One-Unambiguous Regular Languages

Proposed by Brüggemann-Klein and Wood.

A regular language $L$ is one-unambiguous if there is a regular expression $E$ such that $L = L(E)$ and the position automaton of $E$ is deterministic.

<?xml version="1.0"?>
<!DOCTYPE BOOK [ 
  <!ELEMENT p (#PCDATA)> 
  <!ELEMENT BOOK (OPENER,SUBTITLE?,INTRODUCTION?,(SECTION | PART)+)> 
  <!ELEMENT OPENER (TITLE_TEXT)*> 
  <!ELEMENT SUBTITLE (#PCDATA)> 
  <!ELEMENT INTRODUCTION (HEADER, p+)+> 
  <!ELEMENT PART (HEADER, CHAPTER+)> 
  <!ELEMENT SECTION (HEADER, p+)> 
  <!ELEMENT HEADER (#PCDATA)> 
  <!ELEMENT CHAPTER (CHAPTER_NUMBER, CHAPTER_TEXT)> 
  <!ELEMENT CHAPTER_NUMBER (#PCDATA)> 
  <!ELEMENT CHAPTER_TEXT (p)+> ]>

BOOK ::= 
  OPENER · (SUBTITLE+λ) · (INTRODUCTION+λ) · (SECTION + PART)(SECTION + PART)*

One-unambiguous regular expression!!
One-Unambiguous Regular Languages vs XML DTD

- Regular expressions for content models of DTD are one-unambiguous.
- XML DTDs are LL(1) grammars [Wood’96].
- $LL(k)$ grammars have a proper hierarchy [AU’72].
- $k$-unambiguous regular languages??
One-Unambiguous Regular Languages vs XML DTD

- Regular expressions for content models of DTD are one-unambiguous
- XML DTDs are LL(1) grammars [Wood’96]
- \(LL(k)\) grammars have a proper hierarchy [AU’72]
- \(k\)-unambiguous regular languages??
  
  We have \(k\)-lookahead for processing an input string.

```plaintext
XML INSTANCE
```

current state

6-lookahead
$k$-lookahead Regular Languages

Two ways for defining $k$-lookahead regular languages.

- The first is based on a lookahead of at most $k \geq 1$ symbols to determine the next, at most one, matching position in a given regular expression: deterministic $k$-lookahead regular expressions

- The second is similar except that when we use a lookahead of $k$ symbols, we must match the next $k$ positions uniquely: $k$-block-deterministic regular expressions
Deterministic $k$-lookahead regular languages

at state $q_i$

$\cdots a_i a_{i+1} a_{i+2} \cdots a_i a_k a_{k+1}$

$k$-lookahead

after reading $a_{i+1}$

at state $q_i+1$

$\cdots a_i a_{i+1} a_{i+2} \cdots a_i a_k a_{k+1}$

$k$-lookahead
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$$E = (a + b)^* a$$
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$E = (a + b)^* a$

$\cdots aaa\#$
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

\[ E = (a + b)^* a \]

\[
\begin{array}{c}
0 \quad a \\
\downarrow \quad a \\
1 \quad a \\
\downarrow \quad a \\
2 \quad b \\
\downarrow \quad b \\
3 \quad a \\
\end{array}
\]

\[
\begin{array}{c}
\cdots [a a a \#] 
\end{array}
\]
Deterministic $k$-lookahead regular languages

A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

\[ E = (a + b)^* a \]

\[ \cdots [aaa\#] \]
A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$$E = (a + b)^*a$$

$\cdots[aaa\#]$
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$$E = (a + b)^* a$$

$\cdots a[a][a][\#]$
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$$E = (a + b)^* a$$

$$\cdots a\underline{a}a\#$$
Deterministic $k$-lookahead regular languages

- A regular language $L$ is deterministic $k$-lookahead if there is a deterministic $k$-lookahead regular expression for $L$.

- A regular expression is deterministic $k$-lookahead if its position automaton is deterministic $k$-lookahead.

$$E = (a + b)^* a$$

$E$ is deterministic 2-lookahead.

$$\cdots aaaa\#$$
Deterministic $k$-lookahead regular languages

**Thm.** $L((a+b)^*a(a+b)^k)$, for $k \geq 0$, is deterministic $(k+1)$-lookahead.
Deterministic $k$-lookahead regular languages

**Thm.** $L((a+b)^*a(a+b)^k)$, for $k \geq 0$, is deterministic $(k+1)$-lookahead.
Deterministic $k$-lookahead regular languages

**Thm.** $L((a+b)^* a (a+b)^k)$, for $k \geq 0$, is deterministic $(k+1)$-lookahead.
Deterministic $k$-lookahead regular languages

**Thm.** $L((a+b)^*a(a+b)^k)$, for $k \geq 0$, is deterministic $(k+1)$-lookahead.

- There exists a **hierarchy** for deterministic $k$-lookahead regular languages
\(k\)-block-deterministic regular languages

After reading \(a_i + 1 \cdot \cdot \cdot a_i + k\)

at state \(q_i\)

\(k\)-lookahead

at state \(q_i'\)
We define a regular language $L$ to be $k$-block-deterministic if there exists a $k$-block automaton $A' = (Q, \Sigma, \Gamma, \delta, s, F)$ that satisfies the following conditions:

1. $A'$ is a position automaton over $\Gamma$.
2. $A'$ is a deterministic block automaton.
3. $L = L(A')$.

It is easy to verify that a position automaton $A$ for an 1-deterministic regular language is 1-block-deterministic.
$k$-block-deterministic regular languages

**Thm.** There is a proper hierarchy in $k$-block-deterministic regular languages.

**Sketch of Proof.** A $(k-1)$-block-deterministic regular language is $k$-block-deterministic by definition. Thus, it is enough to show that there is a $k$-block-deterministic regular language that is not $(k-1)$-block-deterministic.
\[ k-3 \text{ states} \]

Diagram showing a transition graph with states labeled as follows:

- **A**: Transition from state $q_1$ to $q_2$ on input $a$, then to $q_3$ on input $b$.
- **A'**: Transition from state $q_1$ to $q_3$ on input $aaa \cdots aab$.

States are connected by arrows indicating transitions on inputs $a$ and $b$. The diagram includes a sequence of states labeled with dotted lines to indicate the $k-3$ states.
Two Ways...

**Thm.** $k$-block-deterministic regular languages are a proper subfamily of deterministic $k$-lookahead regular languages.

XML DTD vs XML Schema

- There’s no vs
- XML Schema are much more flexible and powerful
- Thus, there’re also much more difficult and confusing
XML DTD vs XML Schema

- There’s no vs
- XML Schema are much more flexible and powerful
- Thus, there’re also much more difficult and confusing
XML DTD vs XML Schema

- There’s no vs
- XML Schema are much more flexible and powerful
- Thus, there’re also much more difficult and confusing

XML DTD ➔ 1-lookahead determinism ➔ XML Schema ➔ k-lookahead determinism
XML DTD vs XML Schema

- There’s no vs
- XML Schema are much more flexible and powerful
- Thus, there’re also much more difficult and confusing

XML DTD = 1-lookahead determinism

XML Schema = $k$-lookahead determinism
XML DTD vs XML Schema

- There’s no vs
- XML Schema are much more flexible and powerful
- Thus, there’re also much more difficult and confusing

\[\begin{align*}
\text{XML DTD} & \quad \equiv \quad 1\text{-lookahead determinism} \\
\text{XML Schema} & \quad ? \quad k\text{-lookahead determinism}
\end{align*}\]
Pattern Matching - an application of regular languages

Given a regular expression pattern $P$ and a text $T$, find all substrings of $T$ that are in $L(P)$.

$T = AGCTAATCCCTGAGAGTCCAGTTAGTCCCAT$

$P = T \cdot (AG + C)^* \cdot T$
Pattern Matching - an application of regular languages

Given a regular expression pattern $P$ and a text $T$, find all substrings of $T$ that are in $L(P)$.

$T = AGCTAA\textcolor{red}{TCCCTGAGAGTCCAGT}TAGTCCCAT$

$P = T \cdot (AG + C)^* \cdot T$
Pattern Matching

New Domains: WEB, Bioinformatics, Huge DB, Images or Source Codes
Pattern Matching - related work

Given a text $T$ and a regular expression $E$,

- The recognition problem: We can report all end positions of matching substrings of $T$ in $O(mn)$ time [Aho] or in $O(mn/\log n)$ time [Myers].

- The identification problem: We can report all (start, end) positions of matching substrings of $T$ in $O(mn^2)$ time [Aho].
Pattern Matching - recognition problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba$$

$$T = \begin{array}{cccccccccccc}
a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}$$
Pattern Matching - recognition problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

\[ E = a(a + b)^*ba \]

\[ T = \begin{array}{cccccccccccc}
    a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array} \]
Pattern Matching - recognition problem

Given \( E \) over \( \Sigma \), we prepend \( \Sigma^* \) to \( E \); this allows matching to begin at any position in \( T \).

\[
E = a(a + b)^*ba
\]

\[
T = \begin{array}{cccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}
\]
Pattern Matching - recognition problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba$$

$$T = \begin{array}{cccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}$$

Given $E$ and $T$, we can find all end positions of matching substrings of $T$ in $O(mn)$ time using $O(m)$ space, where $|E| = m$ and $|T| = n$ [Aho].
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba$$

$$T = \begin{array}{cccccccccccc}
a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}$$
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba \quad E^R = ab(a + b)^*a$$

$T = \begin{array}{ccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}$
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a+b)^*ba \quad E^R = ab(a+b)^*a$$

$$T = \begin{array}{cccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}$$
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

\[ E = a(a + b)^*ba \quad E^R = ab(a + b)^*a \]

$T = a\ b\ b\ a\ b\ a\ a\ b\ a\ b\ b\ a\ a$
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

\[
E = a(a + b)^*ba \quad E^R = ab(a + b)^*a
\]

$T =$

\[
\begin{array}{cccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a \\
\end{array}
\]
Pattern Matching - identification problem

Given \( E \) over \( \Sigma \), we prepend \( \Sigma^* \) to \( E \); this allows matching to begin at any position in \( T \).

\[
E = a(a+b)^*ba \quad E^R = ab(a+b)^*a
\]

\[
T = \begin{array}{cccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a
\end{array}
\]
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba \quad E^R = ab(a + b)^*a$$

$T = a\ b\ b\ a\ b\ a\ a\ b\ a\ b\ b\ a\ a$

Running Time = No. of matching end positions $\times O(mn)$
$= O(n) \times O(mn) = O(mn^2)$. 
Pattern Matching - identification problem

Given $E$ over $\Sigma$, we prepend $\Sigma^*$ to $E$; this allows matching to begin at any position in $T$.

$$E = a(a + b)^*ba \quad E^R = a(b(a + b))^*a$$

$$T = \begin{array}{cccccccccccccc}
  a & b & b & a & b & a & a & b & a & b & b & a & a
\end{array}$$

Running Time = No. of matching end positions $\times O(mn)$

$$= O(n) \times O(mn) = O(mn^2).$$

We can solve the identification problem in $O(mn^2)$ worst-case time using $O(m)$ space [Aho].
Prefix and Infix

Given two strings $x$ and $y$ over $\Sigma$, we say

- $x$ is a **prefix** of $y$ if there exists $z \in \Sigma^*$ such that $xz = y$.

- $x$ is an **infix** of $y$ if there exists $u, v \in \Sigma^*$ such that $uxv = y$; we often call $x$ a **substring** of $y$. 
Prefix and Infix

Given two strings $x$ and $y$ over $\Sigma$, we say

- $x$ is a prefix of $y$ if there exists $z \in \Sigma^*$ such that $xz = y$.
- $x$ is an infix of $y$ if there exists $u, v \in \Sigma^*$ such that $uxv = y$; we often call $x$ a substring of $y$.

\[
y = \text{seoul}\]
Prefix and Infix

Given two strings $x$ and $y$ over $\Sigma$, we say

- $x$ is a prefix of $y$ if there exists $z \in \Sigma^*$ such that $xz = y$.

- $x$ is an infix of $y$ if there exists $u, v \in \Sigma^*$ such that $uxv = y$; we often call $x$ a substring of $y$.

$y = \text{seoul}$  
‘seo’ is a prefix of $y$.  

Prefix and Infix

Given two strings \( x \) and \( y \) over \( \Sigma \), we say

- \( x \) is a prefix of \( y \) if there exists \( z \in \Sigma^* \) such that \( xz = y \).

- \( x \) is an infix of \( y \) if there exists \( u, v \in \Sigma^* \) such that \( uxv = y \); we often call \( x \) a substring of \( y \).

\[ y = \text{seoul} \quad \text{‘eou’ is an infix of } y. \]
Prefix and Infix

Given two strings $x$ and $y$ over $\Sigma$, we say

- $x$ is a **prefix** of $y$ if there exists $z \in \Sigma^*$ such that $xz = y$.

- $x$ is an **infix** of $y$ if there exists $u, v \in \Sigma^*$ such that $uxv = y$; we often call $x$ a **substring** of $y$.

We define a pattern $P$ to be

- **prefix-free** if no string in $P$ is a prefix of any other strings in $P$.

- **infix-free** if no string in $P$ is an infix of any other strings in $P$. 
Infix-free Regular-Expression Matching

$\mathcal{L}_{IN} \subsetneq \mathcal{L}_{PRE} \subsetneq \mathcal{L}_{REG}$
Infix-free Regular-Expression Matching

$$\mathcal{L}_{IN} \subseteq \mathcal{L}_{PRE} \subseteq \mathcal{L}_{REG}$$

Given an infix-free regular expression $E$ and a text $T$:

$$y = \text{seoul}$$

‘eou’ is an infix of $y$.

<table>
<thead>
<tr>
<th>$E$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
Infix-free Regular-Expression Matching

$$\mathcal{L}_{IN} \subseteq \mathcal{L}_{PRE} \subseteq \mathcal{L}_{REG}$$

Given an infix-free regular expression $E$ and a text $T$:

$$y = \text{seoul}$$

‘eou’ is an infix of $y$.

$E$

$T$

1 2 3 4 5 6 7 8 9 10 11 12

the recognition process
Infix-free Regular-Expression Matching

\( \mathcal{L}_{IN} \subset \mathcal{L}_{PRE} \subset \mathcal{L}_{REG} \)

Given an infix-free regular expression \( E \) and a text \( T \):

\[ y = \text{seoul} \]

‘eou’ is an infix of \( y \).

\[ E \]

\[ T \]

1 2 3 4 5 6 7 8 9 10 11 12

\( \Rightarrow \)

the recognition process
Infix-free Regular-Expression Matching

\( \mathcal{L}_{IN} \subsetneq \mathcal{L}_{PRE} \subsetneq \mathcal{L}_{REG} \)

Given an infix-free regular expression \( E \) and a text \( T \):

\( y = \text{seoul} \quad \text{‘eou’ is an infix of } y. \)

Given an infix-free regular expression \( E \) and a text \( T \):

\( E \)

\( T \)

1 2 3 4 5 6 7 8 9 10 11 12

\( E \)

\( E^R \)
Infix-free Regular-Expression Matching

\[ \mathcal{L}_{IN} \subsetneq \mathcal{L}_{PRE} \subsetneq \mathcal{L}_{REG} \]

Given an infix-free regular expression \( E \) and a text \( T \):

\[ y = \text{seoul} \quad \text{‘eou’ is an infix of } y. \]

Because of infix-freeness, each pair of (↓, ↑) from left to right must be a matching substring.
Infix-free Regular-Expression Matching

\[ \mathcal{L}_{IN} \subsetneq \mathcal{L}_{PRE} \subsetneq \mathcal{L}_{REG} \]

Given an infix-free regular expression \( E \) and a text \( T \):

\[ y = \text{seoul} \quad \text{‘eou’ is an infix of } y. \]

Because of **infix-freeness**, each pair of (\( \downarrow, \uparrow \)) from left to right must be a matching substring.

We can find all matching substrings in \( O(mn) \) time [HWW07].

Prefix-Free Regular Languages and Pattern Matching, **Yo-Sub Han**, Yajun Wang and Derick Wood

Prefix-free Regular-Expression Matching

- $L_{IN} \subsetneq L_{PRE} \subsetneq L_{REG}$
- If $E$ is infix-free, we have an $O(mn)$ running time algorithm
- If $E$ is a (normal) regular expression, we have an $O(mn^2)$ running time algorithm
- If $E$ is prefix-free, then there are at most $n$ matching substrings of $T$ that belong to $L(E)$, where $n$ is the size of $T$. 
Prefix-free Regular-Expression Matching

- $L_{IN} \subsetneq L_{PRE} \subsetneq L_{REG}$

- If $E$ is infix-free, we have an $O(mn)$ running time algorithm

- If $E$ is a (normal) regular expression, we have an $O(mn^2)$ running time algorithm

- If $E$ is prefix-free, then there are at most $n$ matching substrings of $T$ that belong to $L(E)$, where $n$ is the size of $T$. 

$$|T| = 13$$
Prefix-free Regular-Expression Matching

- $L_{IN} \subset L_{PRE} \subset L_{REG}$
- If $E$ is infix-free, we have an $O(mn)$ running time algorithm
- If $E$ is a (normal) regular expression, we have an $O(mn^2)$ running time algorithm
- If $E$ is prefix-free, then there are at most $n$ matching substrings of $T$ that belong to $L(E)$, where $n$ is the size of $T$.

```
a b c a c a c b c b b a a
```

$|T| = 13$

$cacbcb$ is a prefix of $cacbcbba$. This contradicts that $L(E)$ is prefix-free.
Prefix-free Regular-Expression Matching

- $L_{IN} \subsetneq L_{PRE} \subsetneq L_{REG}$
- If $E$ is infix-free, we have an $O(mn)$ running time algorithm
- If $E$ is a (normal) regular expression, we have an $O(mn^2)$ running time algorithm
- If $E$ is prefix-free, then there are at most $n$ matching substrings of $T$ that belong to $L(E)$, where $n$ is the size of $T$.
- Can we have an $O(mn)$ time algorithm?

cacbcb is a prefix of cacbcbba. This contradicts that $L(E)$ is prefix-free.
Prefix-free Regular-Expression Matching

- $L_{IN} \subsetneq L_{PRE} \subsetneq L_{REG}$
- If $E$ is infix-free, we have an $O(mn)$ running time algorithm
- If $E$ is a (normal) regular expression, we have an $O(mn^2)$ running time algorithm
- If $E$ is prefix-free, then there are at most $n$ matching substrings of $T$ that belong to $L(E)$, where $n$ is the size of $T$.
- Can we have an $O(mn)$ time algorithm?

\[ a \ a \ c \ c \ b \ c \ b \] is a prefix of \[ a \ c \ a \ c \ b \ c \ b b a \]. This contradicts that $L(E)$ is prefix-free.

\[\text{\textbf{YES!!}}\]
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]
\[ T \]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[ \rightarrow \]

the recognition process
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[ \begin{array}{ccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \end{array} \]

the recognition process
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
\]

\[ E^R \]
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \]

\[ E^R \]
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[ E^R \]

parallel processing starts
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[ E \]

\[ T \]

\[ a \ b \ b \ a \ b \ a \ a \ b \ b \ a \ b \ a \ a \]

Running Time = No. of matching end positions \( \times O(mn) \)

\( = O(n) \times O(mn) = O(mn^2). \)
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

\[
E
\]
\[
T
\]

Running Time = No. of matching end positions

= \( O(n) \times O(mn) = O(mn^2) \).

Parallel processing starts
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

$$E$$

$$T$$

Running Time = No. of matching end positions
$$= O(n) \times O(mn) = O(mn^2)$$.

parallel processing starts
Prefix-free Regular-Expression Matching

Sketch of our algorithm:

Because of prefix-freeness, no two process can have the same state of $E$ at the same time. This implies that a single reverse scan is enough to find corresponding start positions for each end position.
Prefix-free Regular-Expression Matching

Given a prefix-free regular expression $E$ and a text $T$, we can identify all matching substrings of $T$ that belong to $L(E)$ in $O(mn)$ worst-case time - [HWW07].

Prefix-Free Regular Languages and Pattern Matching, Yo-Sub Han, Yajun Wang and Derick Wood
State Complexity

What is the state complexity of a regular language \( L \)?
State Complexity

What is the state complexity of a regular language \( L \)?

**State complexity** is a descriptional complexity of \( L \)

- \( L \) has a unique minimal DFA \( A \)
- We define the state complexity of \( L \) to be the number of states in \( A \)
State Complexity

What is the state complexity of a regular language $L$?

State complexity is a descriptional complexity of $L$

- $L$ has a unique minimal DFA $A$
- We define the state complexity of $L$ to be the number of states in $A$

We can estimate needed resource.
State Complexity Problem

Given two (arbitrary) regular languages $L_1$ and $L_2$, what is the state complexity of $L_1 \cap L_2$?
State Complexity Problem

Given two (arbitrary) regular languages $L_1$ and $L_2$, what is the state complexity of $L_1 \cap L_2$?

- Upper bound

![Diagram](image)
State Complexity Problem

Given two (arbitrary) regular languages $L_1$ and $L_2$, what is the state complexity of $L_1 \cap L_2$?

- Upper bound

\[ m_1 \cap m_2 \begin{array}{c}
\cap
\end{array} \quad f(m_1, m_2) \leq \text{at most} \]
State Complexity Problem

Given two (arbitrary) regular languages $L_1$ and $L_2$, what is the state complexity of $L_1 \cap L_2$?

- Upper bound

- Lower bound

\[ m_1 \cap m_2 \leq f(m_1, m_2) \]
State Complexity Problem

Given two (arbitrary) regular languages $L_1$ and $L_2$, what is the state complexity of $L_1 \cap L_2$?

- **Upper bound**

  \[
  m_1 \cap m_2 \leq f(m_1, m_2)
  \]

- **Lower bound**

  Present two (general) $L_1$ and $L_2$ such that the state complexity of $L_1 \cap L_2$ always reaches the upper bound.
State Complexity Problem

Given two (arbitrary) regular languages \( L_1 \) and \( L_2 \), what is the state complexity of \( L_1 \cap L_2 \)?

- Upper bound

\[
\text{at most } f(m_1, m_2)
\]

- Lower bound

Present two (general) \( L_1 \) and \( L_2 \) such that the state complexity of \( L_1 \cap L_2 \) always reaches the upper bound.

- Tight bound: \( \text{UB} = \text{LB} \), the state complexity of the intersection of two regular languages is \( f(m_1, m_2) \)
State Complexity - Motivation

In recent years, there have been many new applications of FAs, such as in natural language and speech processing, software engineering, and image generation and encoding that need a large number of states.

the Bell Labs multilingual TTS system: 26.6MB for German, 30.0MB for French and 39.0MB for Chinese.
State Complexity - motivation

New Helper: FA manipulation software systems such as *Grail+, Auto-mate* and *FireLite*
State Complexity - motivation

New Helper: FA manipulation software systems such as Grail+, Automate and FireLite

We calculate the upper bound.
State Complexity - motivation

New Helper: FA manipulation software systems such as *Grail+, Auto-mate* and *FireLite*

We calculate the upper bound.

We guess a lower bound and verify it, and repeat this step until we find a matching lower bound.
State Complexity - motivation

New Helper: FA manipulation software systems such as *Grail+*, *Auto-mate* and *FireLite*

We calculate the upper bound.

We guess a lower bound and verify it, and repeat this step until a matching lower bound.
# State Complexity

<table>
<thead>
<tr>
<th>operation</th>
<th>finite languages</th>
<th>regular languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1 \cup L_2$</td>
<td>$O(mn)$</td>
<td>$mn$</td>
</tr>
<tr>
<td>$L_1 \cap L_2$</td>
<td>$O(mn)$</td>
<td>$mn$</td>
</tr>
<tr>
<td>$\Sigma^* \setminus L_1$</td>
<td>$m$</td>
<td>$m$</td>
</tr>
<tr>
<td>$L_1 \cdot L_2$</td>
<td>$(m - n + 3)2^{n-2} - 1$</td>
<td>$(2m - 1)2^{n-1}$</td>
</tr>
<tr>
<td>$L_1^*$</td>
<td>$2^{m-3} + 2^{m-4}$, for $m \geq 4$</td>
<td>$2^{m-1} + 2^{m-2}$</td>
</tr>
<tr>
<td>$L_1^R$</td>
<td>$3 \cdot 2^{p-1} - 1$ if $m = 2p$, $2^p - 1$ if $m = 2p - 1$</td>
<td>$2^m$</td>
</tr>
</tbody>
</table>
Union of Finite Languages

Given two minimal DFAs $A$ and $B$ for non-empty finite languages $L_1$ and $L_2$, we can construct a DFA for $L(A) \cup L(B)$ based on the Cartesian product of states as follows:
Let $A = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $B = (Q_2, \Sigma, \delta_2, s_2, F_2)$.

$M = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F)$, where for all $p \in Q_1$ and $q \in Q_2$ and $a \in \Sigma$,
\[
\delta((p, q), a) = (\delta(p, a), \delta(q, a))
\]
and $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

$M$ is deterministic.
The $m-1$th state in $A$ is the final state whose out-transitions go to the sink state, the $m$th state.
The $m-1$th state in $A$ is the final state whose out-transitions go to the sink state, the $m$th state.

For a state $(i, j)$ in $M$, $L_{i,j}(M) = L_i(A) \cup L_j(B)$. 

Union of Finite Languages - Cartesian Product of States
The $m-1$th state in $A$ is the final state whose out-transitions go to the sink state, the $m$th state.

For a state $(i, j)$ in $M$, 
$L_{i,j}(M) = L_i(A) \cup L_j(B)$.

all states are unreachable from state $(1,1)$ since $A$ and $B$ are non-returning.
Union of Finite Languages - Cartesian Product of States

The $m-1$th state in $A$ is the final state whose out-transitions go to the sink state, the $m$th state.

For a state $(i, j)$ in $M$, $L_{i,j}(M) = L_i(A) \cup L_j(B)$.

all states equivalent since $L_{m-1,n-1} = L_{m-1,n}$

all states are unreachable from state $(1,1)$ since $A$ and $B$ are non-returning.
Union of Finite Languages - Cartesian Product of States

The \( m-1 \)th state in \( A \) is the final state whose out-transitions go to the sink state, the \( m \)th state.

For a state \( (i,j) \) in \( M \),
\[ L_{i,j}(M) = L_i(A) \cup L_j(B). \]

**Lemma 1.** \( mn - (m + n - 2) - 2 = mn - (m + n) \) states are sufficient for \( L(A) \cup L(B) \).
Lemma 1. $mn - (m + n)$ states are sufficient for $L(A) \cup L(B)$.

The next question is whether or not the bound is reachable in general.
Lemma 1. $mn - (m + n)$ states are sufficient for $L(A) \cup L(B)$.

The next question is whether or not the bound is reachable in general.

The answer is YES and NO.
Lemma 2. The upper bound $mn - (m + n)$ cannot be reached with a fixed alphabet when $m$ and $n$ are arbitrarily large.

Proof.
Let $A$ have $\{p_0, p_1, \ldots, p_{m-1}\}$ and $B$ have $\{q_0, q_1, \ldots, q_{n-1}\}$.
We order the states such that if $p_j$ is reachable from $p_i$, then $i < j$.
Let $i \in \{1, \ldots, m-1\}$. Any string that reaches $p_i$ from $p_0$ can go through only the states $p_1, \ldots, p_{i-1}$ in between and cannot visit the same state twice.
Hence, there are at most
\[
 t + t^2 + \cdots + t^i = \frac{t(t^i - 1)}{t - 1} = \text{def } D(i)
\]
strings that can reach $p_i$ from $p_0$. 
Lemma 2. The upper bound \( mn - (m + n) \) cannot be reached with a fixed alphabet when \( m \) and \( n \) are arbitrarily large.

Proof.
Let \( A \) have \( \{p_0, p_1, \ldots, p_{m-1}\} \) and \( B \) have \( \{q_0, q_1, \ldots, q_{n-1}\} \).
We order the states such that if \( p_j \) is reachable from \( p_i \), then \( i < j \).
Let \( i \in \{1, \ldots, m-1\} \). Any string that reaches \( p_i \) from \( p_0 \) can go through only the states \( p_1, \ldots, p_{i-1} \) in between and cannot visit the same state twice.
Hence, there are at most
\[
t + t^2 + \cdots + t^i = \frac{t(t^i - 1)}{t - 1} = \text{def } D(i)
\]
strings that can reach \( p_i \) from \( p_0 \).
Since \( M_\cup \) is deterministic, for any fixed \( i \) for \( 1 \leq i < m - 1 \), at most \( D(i) \) of the pair-states \((p_i, q_j)\) are reachable from \((p_0, q_0)\) in \( M_\cup \).

Thus, if \( n - 2 > D(i) \), then some pair-states with \( p_i \) as the first component are not reachable. Therefore, the bound \( mn - (m + n) \) is not reachable.
Union of Finite Languages

- **Lemma 2.** The upper bound $mn - (m + n)$ cannot be reached with a fixed alphabet when $m$ and $n$ are arbitrarily large.

What if the size of an alphabet is NOT fixed?
Lemma 2. The upper bound $mn - (m + n)$ cannot be reached with a fixed alphabet when $m$ and $n$ are arbitrarily large.

What if the size of an alphabet is NOT fixed?

Lemma 3. The upper bound $mn - (m + n)$ is reachable if the size of the alphabet can depend on $m$ and $n$. 
Lemma 3. The upper bound \( mn - (m + n) \) is reachable if the size of the alphabet can depend on \( m \) and \( n \).

We prove the lemma by presenting two finite languages whose union reaches the bound.

Let \( \Sigma = \{b, c\} \cup \{a_{i,j} \mid 1 \leq i \leq m - 2, 1 \leq j \leq n - 2 \text{ and } (i,j) \neq (m-2,n-2)\} \)

Let \( A = (Q_1, \Sigma, \delta_1, p_0, \{p_{m-2}\}) \), where \( Q_1 = \{p_0, p_1, \ldots, p_{m-1}\} \) and \( \delta_1 \) is defined as follows:

- \( \delta_1(p_i, b) = p_{i+1} \), for \( 0 \leq i \leq m - 2 \).
- \( \delta_1(p_0, a_{i,j}) = p_i \), for \( 1 \leq i \leq m - 2 \) and \( 1 \leq j \leq n - 2 \), \( (i,j) \neq (m-2,n-2) \).

Let \( B = (Q_2, \Sigma, \delta_2, q_0, \{q_{n-2}\}) \), where \( Q_2 = \{q_0, q_1, \ldots, q_{n-1}\} \) and \( \delta_2 \) is defined as follows:

- \( \delta_2(q_i, c) = q_{i+1} \), for \( 0 \leq i \leq n - 2 \).
- \( \delta_2(q_0, a_{i,j}) = q_j \), for \( 1 \leq j \leq n - 2 \) and \( 1 \leq i \leq m - 2 \), \( (i,j) \neq (m-2,n-2) \).
Lemma 3. The upper bound $mn - (m + n)$ is reachable if the size of the alphabet can depend on $m$ and $n$.

An example of two minimal DFAs for finite languages whose sizes are 6 and 5, respectively, where state 5 above and state 4 below are sink states.
Union of Finite Languages

Lemma 3. The upper bound $mn - (m + n)$ is reachable if the size of the alphabet can depend on $m$ and $n$.

Let $L = L(A) \cup L(B)$. We show that there exists a set $R$ consisting of $mn - (m + n)$ strings over $\Sigma$ that are pairwise inequivalent modulo the right invariant congruence of $L$.

Let $R = R_1 \cup R_2 \cup R_3$, where

$R_1 = \{b^i \mid 0 \leq i \leq m - 1\}$.

$R_2 = \{c^j \mid 1 \leq j \leq n - 3\}$. (Note that $R_2$ does not include strings $c^0$, $c^{n-2}$ and $c^{n-1}$.)

$R_3 = \{a_{i,j} \mid 1 \leq i \leq m - 2 \text{ and } 1 \leq j \leq n - 2 \text{ and } (i,j) \neq (m-2,n-2)\}$.

It is easy to verify that all strings in $R$ are pairwise inequivalent. (The complete proof is given in the proceedings.)

Then, $|R| = mn - (m + n)$. 
Union of Finite Languages

**Theorem 1.** Given two minimal DFAs $A$ and $B$ for finite languages, $mn - (m + n)$ states are necessary and sufficient in the worst-case for the minimal DFA of $L(A) \cup L(B)$, where $m = |A|$ and $n = |B|$. 
Union of Finite Languages

Lemma 2 shows that the upper bound is unreachable if $|\Sigma|$ is fixed whereas Lemma 3 shows that the upper bound is reachable if $|\Sigma|$ depends on $m$ and $n$.

Then, what is the state complexity of union with a fixed sized alphabet?
Lemma 4. There exist DFAs $A$ and $B$, with $m$ and $n$ states respectively, that recognize finite languages over $\Sigma$ such that the minimal DFA for $L(A) \cup L(B)$ requires $c(\min\{m, n\})^2$ states.

Proof.
Let $s \geq 1$ be arbitrary and $r = \lceil \log s \rceil$. We define the finite language

$L_1 = \{w_1w_2 \mid |w_1| = 2r, w_2 = \text{odd}(w_1) \in \{a, b\}^*, \text{even}(w_1) \in \{c, d\}^\ast \}$.

$L_1$ can be recognized by a DFA $A$ with at most $10s$ states.
Union of Finite Languages

\[ L_1 = \{ w_1 w_2 \mid |w_1| = 2r, w_2 = \text{odd}(w_1) \in \{a, b\}^*, \text{even}(w_1) \in \{c, d\}^* \}. \]

A DFA \( A \) that recognizes \( L_1 \) when \( r = 3 \). We omit the sink state and its in-transitions.
Union of Finite Languages

Symmetrically, we define

\[ L_2 = \{ w_1w_2 \mid |w_1| = 2r, \text{odd}(w_1) \in \{a, b\}^*, w_2 = \text{even}(w_1) \in \{c, d\}^* \}. \]

The language \( L_2 \) consists of strings \( uv \), where \( |u| = 2r \), odd characters of \( u \) are in \( \{a, b\} \), even characters of \( u \) are in \( \{c, d\} \) and \( \text{even}(u) \) coincides with \( v \).

By a similar argument, \( L_2 \) can be recognized by a DFA \( B \) with at most 10s states.
Union of Finite Languages

Now let \( L = L_1 \cup L_2 \).

Let \( u_1 \) and \( u_2 \) be distinct strings of length \( 2r \) such that odd\((u_i)\) \( \in \{a, b\}^* \) and even\((u_i)\) \( \in \{c, d\}^* \) for \( i = 1, 2 \).

- If \( \text{odd}(u_1) \neq \text{odd}(u_2) \): \( u_1 \cdot \text{odd}(u_1) \in L_1 \subseteq L \) but \( u_2 \cdot \text{odd}(u_1) \notin L \). Hence, \( u_1 \) and \( u_2 \) are not equivalent modulo the right invariant congruence of \( L \).
- If \( \text{even}(u_1) \neq \text{even}(u_2) \): \( u_1 \cdot \text{even}(u_1) \in L_2 \subseteq L \) but \( u_2 \cdot \text{even}(u_1) \notin L \).

The above implies that the right invariant congruence of \( L \) has at least \( 2^r \cdot 2^r \geq s^2 \) different classes. Therefore, if \( m = n = 10s \) is the size of the minimal DFAs for the finite languages \( L_1 \) and \( L_2 \), then we know that the minimal DFA for \( L = L_1 \cup L_2 \) needs at least

\[
\frac{1}{100} n^2 \text{ states.}
\]
RECAP

- Structural properties of the $k$-lookahead determinism that might lead to an efficient XML Schema parser
- Fast regular-expression pattern matching algorithms
- State Complexity
Future Directions and Conclusions

Hierarchy of \( k \)-lookahead determinism

XML Schema parser

pattern matching + indexing

regular-expression pattern matching system for source codes

state complexity

pure theory

practical application
THANK YOU
ANY QUESTIONS??