Hierarchical-Alphabet Automata

*Guest lecture for the **"Current Trends in Artificial Intelligence"** course*

Introduction

- **Who am I?**
    - PhD student at the VUB AI lab
    - Under supervision of Prof. Dr. Johan Loeckx
    - Research interests in applied AI and cybersecurity

- **What is this guest lecture about?**
    - Finite state machine variants + their applications
    - Our hierarchical extension on finite state machines
    - Involvement in projects of the AI lab *Applied Research* team

Topic of this guest lecture

- **Preliminary knowledge**
  - Basics of languages and automata
  - Directed acyclic graphs and ordered sets
  - Applications, advantages and disadvantages

- **My PhD research**
  - Introduction to **Hierarchical-Alphabet Automata (HAA)**
  - Applied research projects within the AI Lab (using HAA)

Before we begin...

- **Slides are as self-contained as possible:**
  - Lots of information on slides, but we'll skip where necessary
  - If you miss guest lecture, you can still read the slides by yourself

- **Follow along with the slides:** patricks.phd/current-trends/slides.pdf

Terminology

- **Alphabet:** finite set of symbols
- **Words:** concatenations of zero or more symbols

- **Factors:** <u>contiguous</u> subsequences of a word
  - **Prefix:** a factor at the beginning of a word
  - **Suffix:** a factor at the end of a word

- **Language:** subset of the infinite set of possible words we can create using an alphabet of symbols

Examples

- **Alphabet:** *{ a, b, c, ..., z }*
- **Words:** *{ "", "a", "aa", "abc", "cars", ... }*

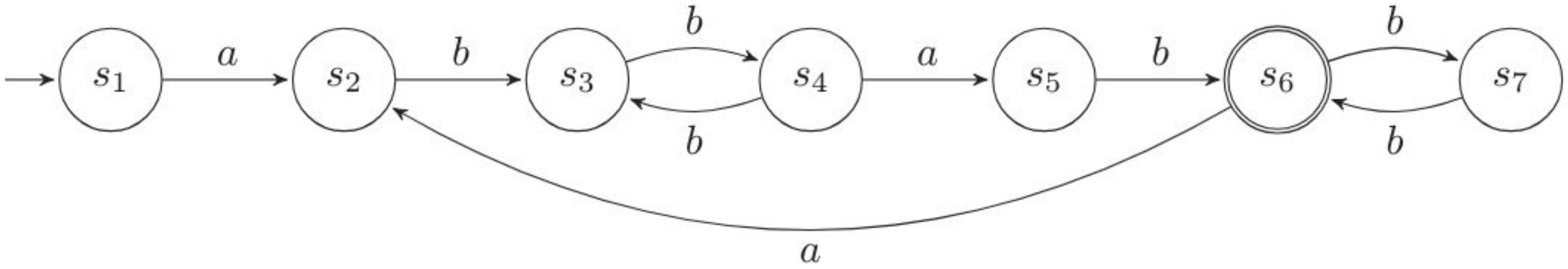
- **Factors:** <u>actor</u> is a factor of <u>factory</u>
  - **Prefix:** <u>fact</u> is a factor and prefix of <u>factory</u>
  - **Suffix:** <u>tory</u> is a factor and suffix of <u>factory</u>

- **Language:** a*b*, the set of factors of a word, ...

Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

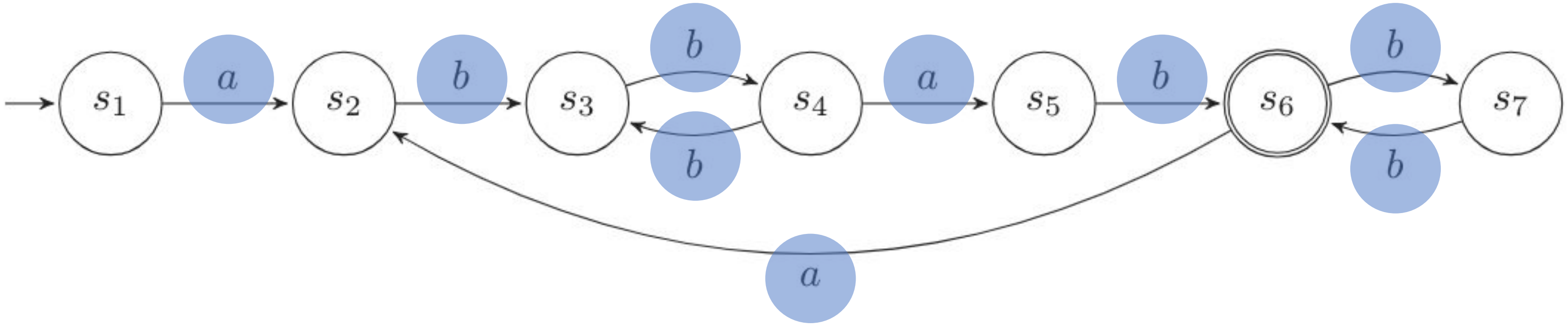
- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \rightarrow S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**



Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

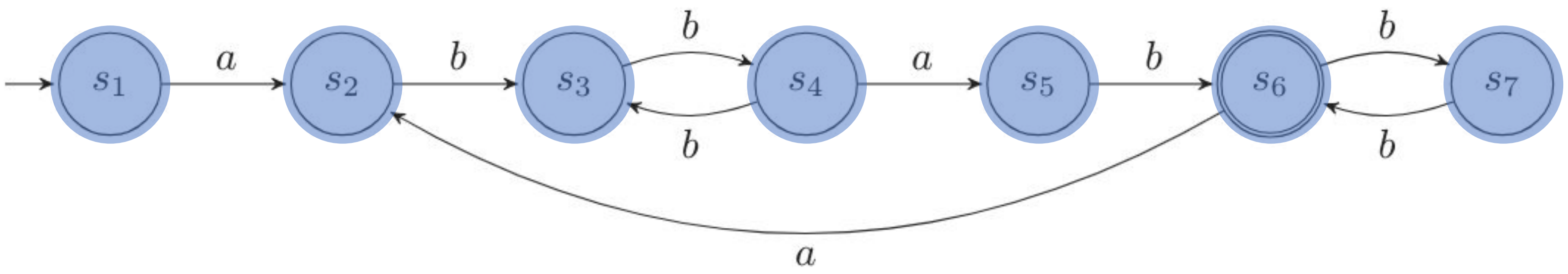
- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \to S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**



Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

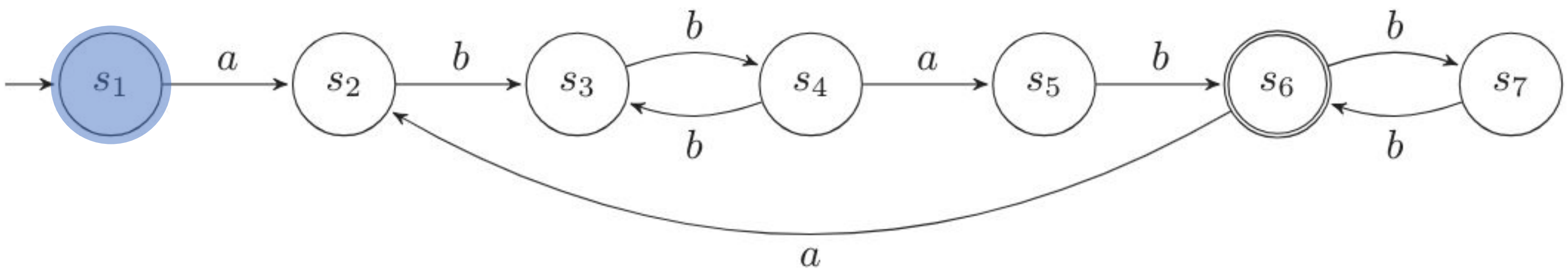
- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \rightarrow S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**



Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \to S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**

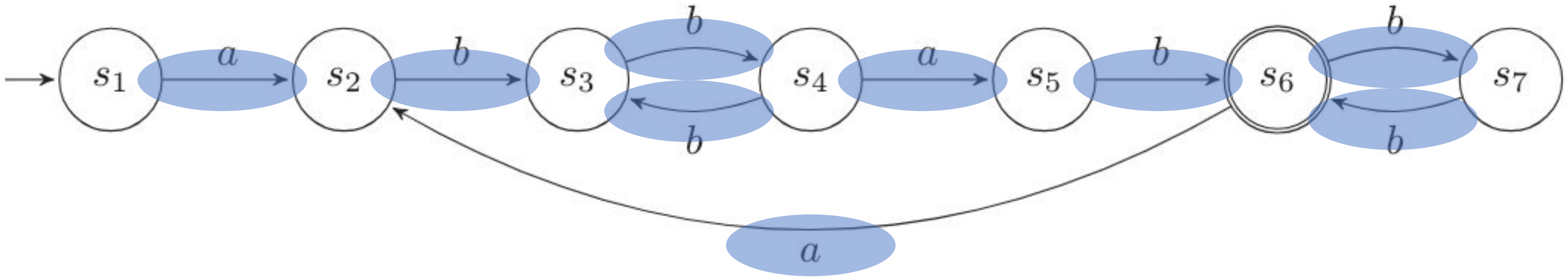


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

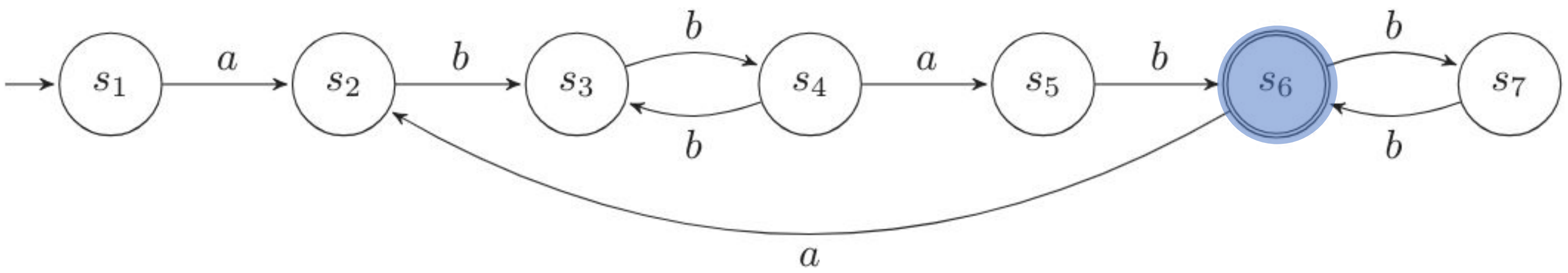
- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \to S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**



Finite State Machine (FSM)

A **deterministic FSM** is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

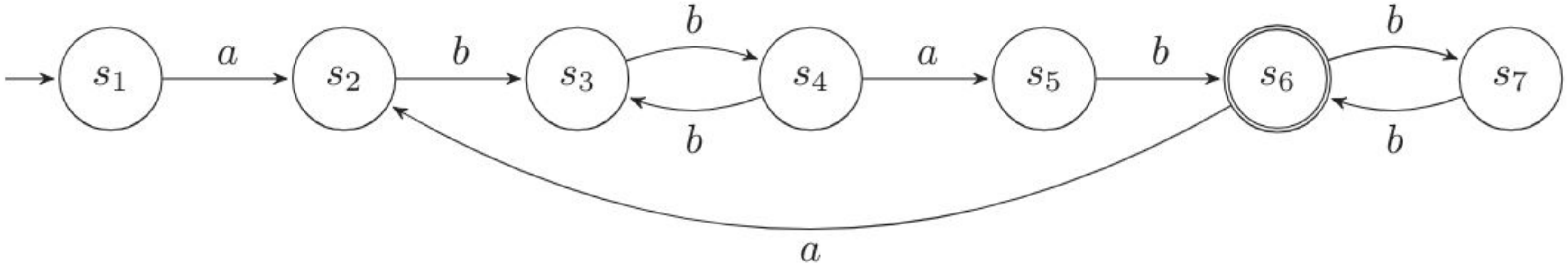
- Σ is a finite set of symbols (the **input alphabet**)
- S is a finite set of **states**
- $s_0 \in S$ is the **initial state**
- $\delta : S \times \Sigma \to S$ is the **state-transition function**
- $F \subseteq S$ the set of **final or accepting states**



Acceptance and rejection

- **An FSM recognises a language:**
  - It *accepts* words that are part of this language
  - It *rejects* words that are not part of this language

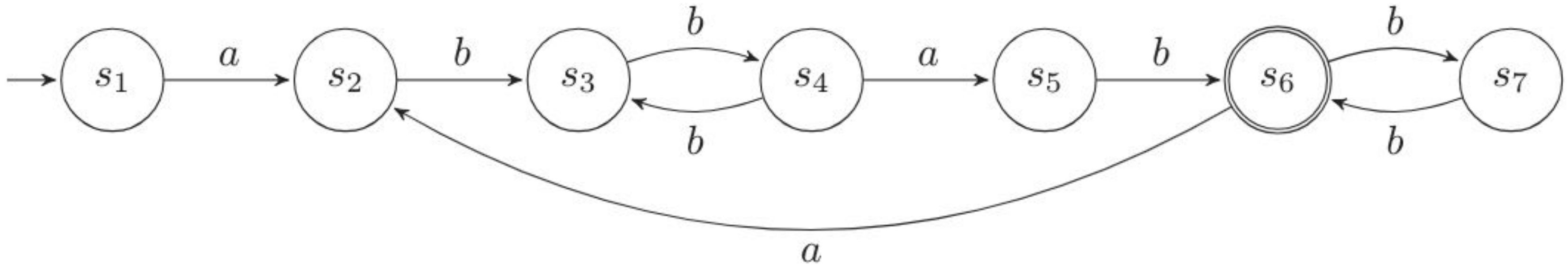
- **Start at the initial state + first symbol of the input:**
  - Repeatedly follow transition function with current state and symbol
    - ***Accepts*** *word if you end up in accepting state*
    - ***Rejects*** *word if you end up in normal state, or if there is no transition*



ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Example

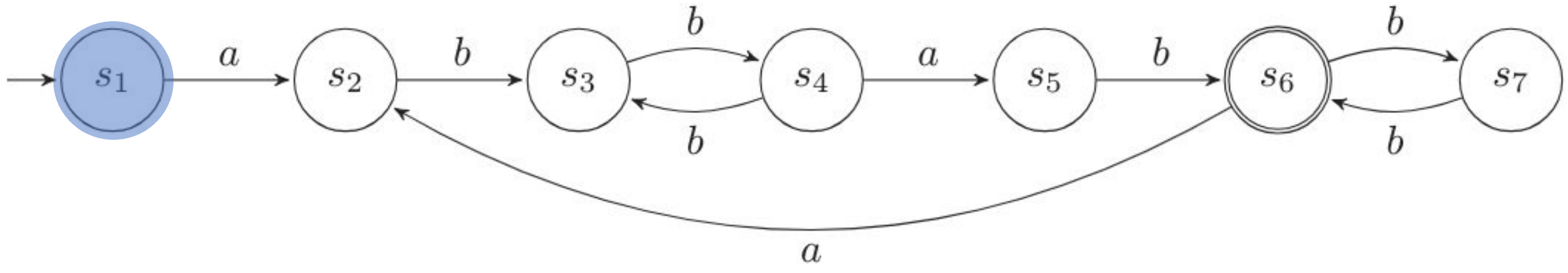
- FSM that recognises the language of **alternating even and odd number of occurrences of ‘*b*’ separated by a single ‘*a*’**



Example

- **Input the word:**

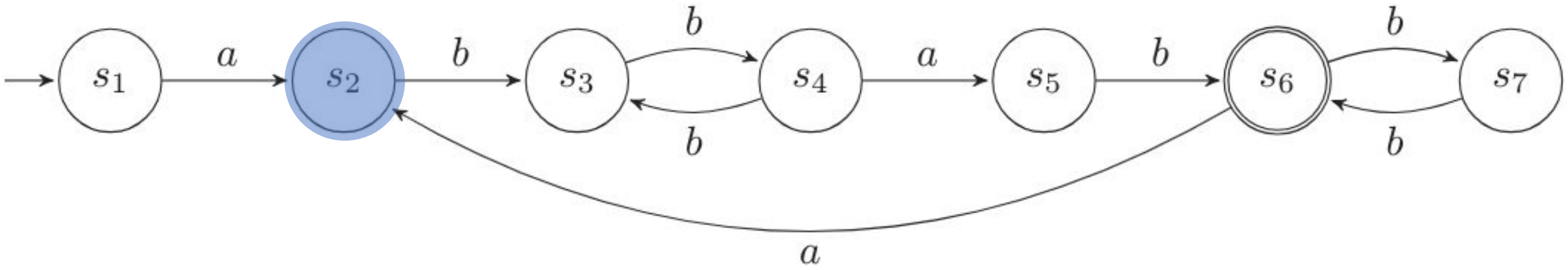
abbabbb



Example

- **Input the word:**

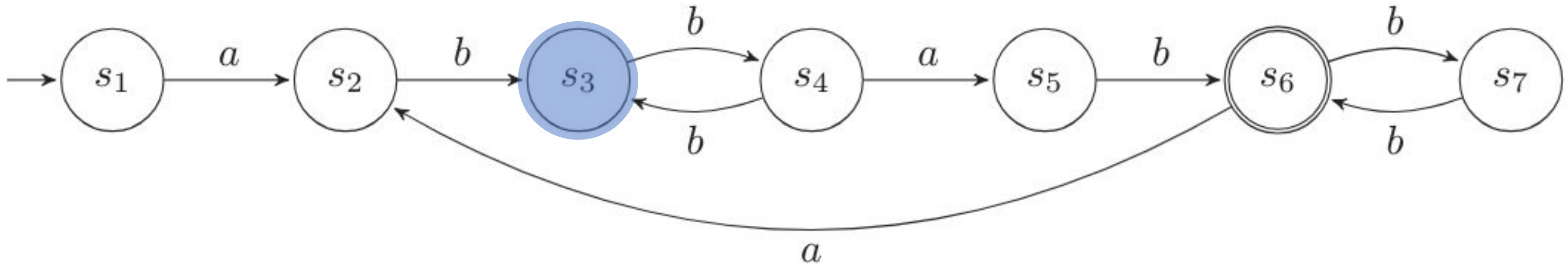
abbabbb



Example

- **Input the word:**

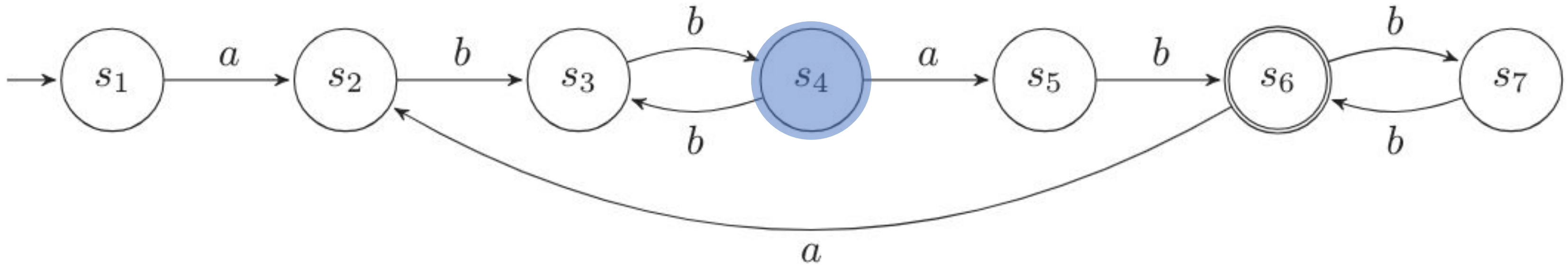
*a**b**babbb*



Example

- **Input the word:**

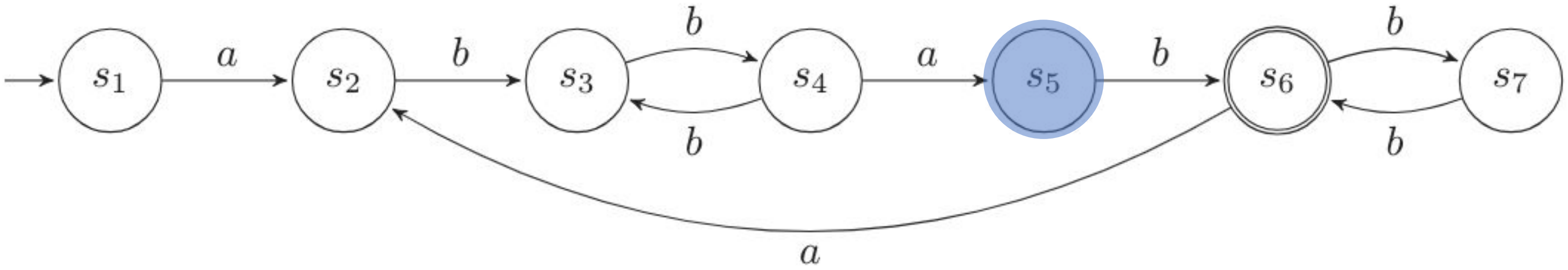
$ab\boldsymbol{b}abbb$



Example

- **Input the word:**

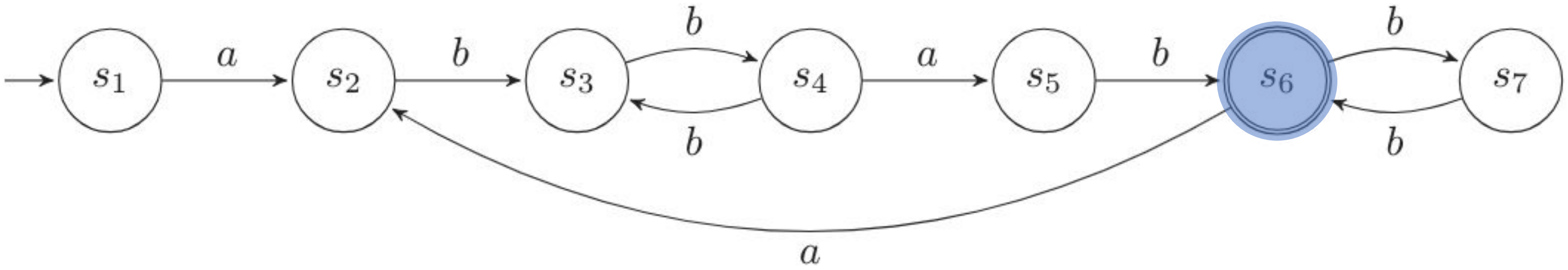
*abb***a***bbb*



Example

- **Input the word:**

*abba***b***bb*

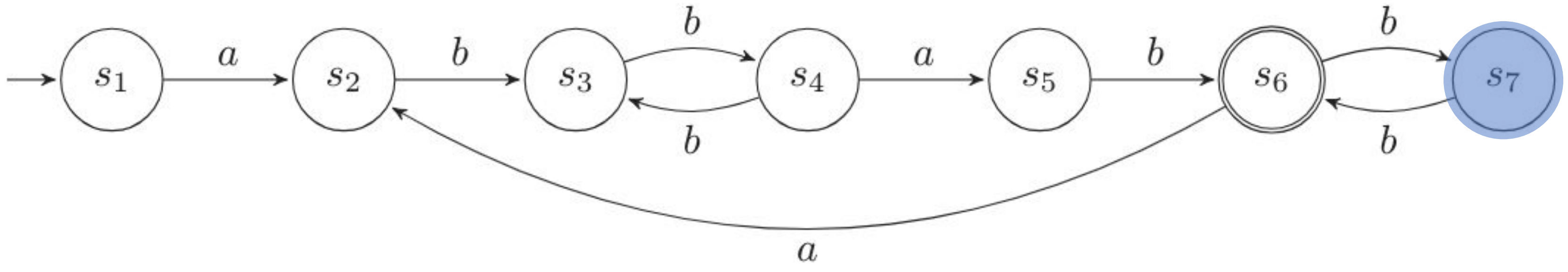


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Example

- **Input the word:**

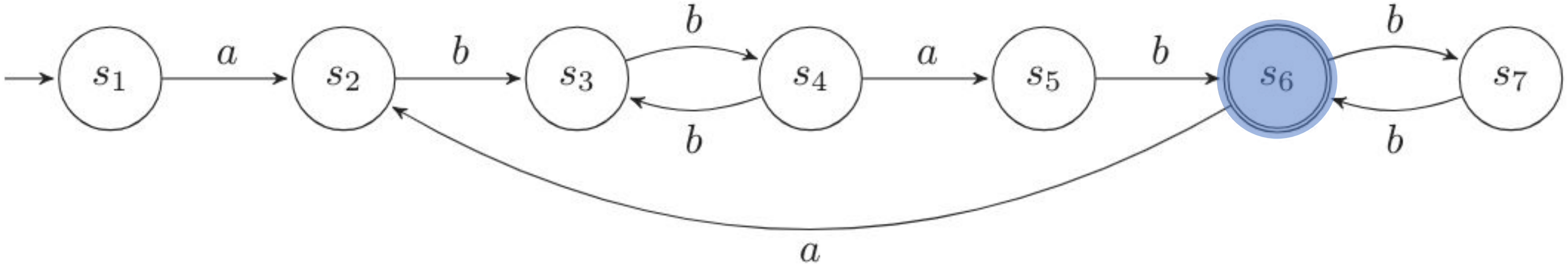
*abbab**b**b*



Example

- **Input the word:**

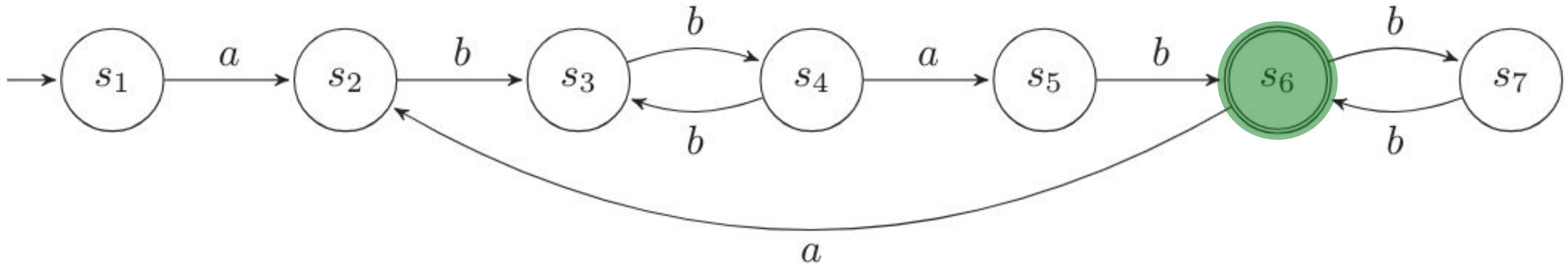
*abbabb***b**



Example

- **Input the word:**

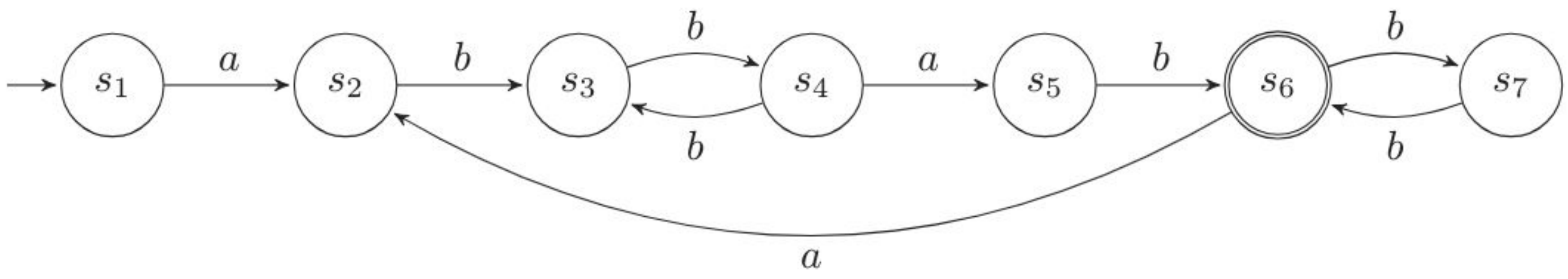
abbabbb ✅



Example

- **What if we would have:**

“abbabb”?

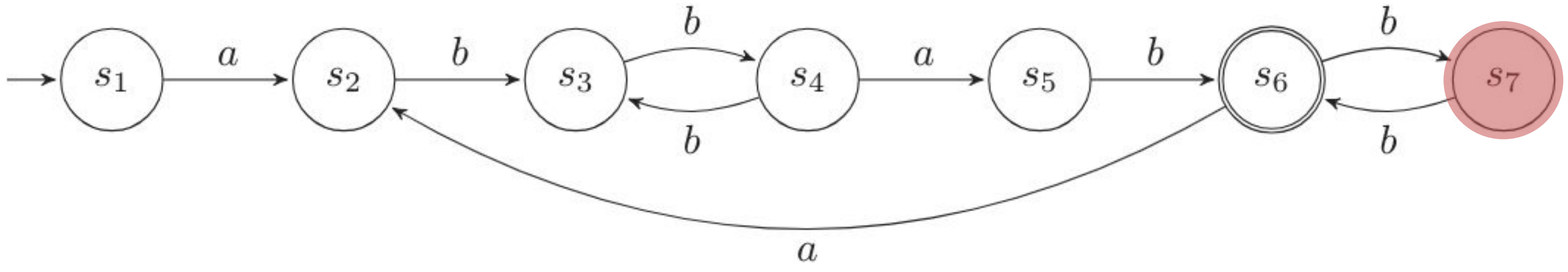


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Example

- **What if we would have:**

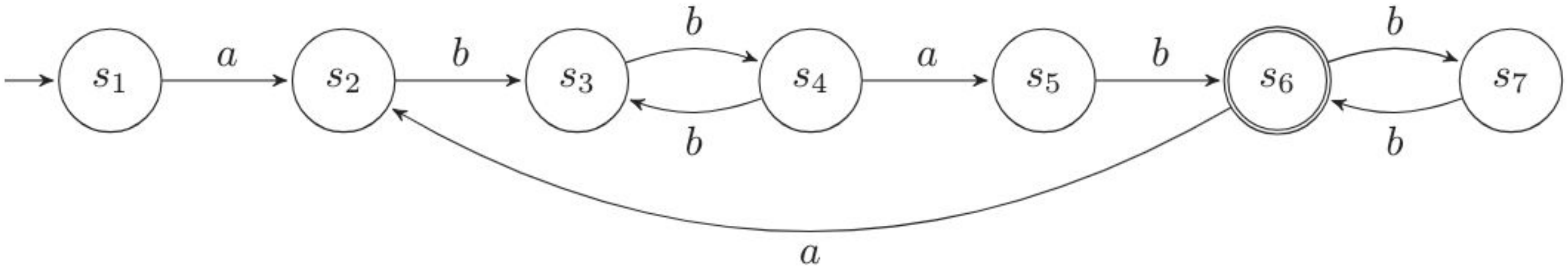
abbab**b** ❌



Example

- **What if we would have:**

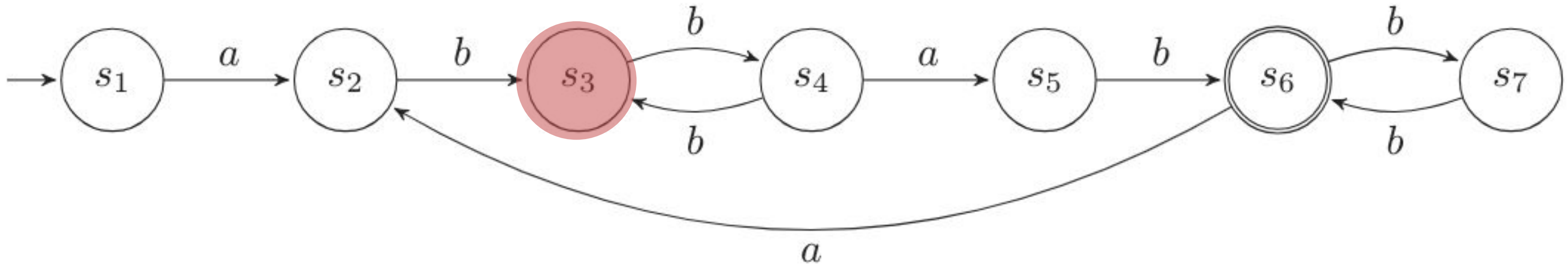
"ababbb"?



Example

- **What if we would have:**

ababbb ❌



Applications of FSMs

- **Anomaly detection:**
  - Learning a language from sequences of system calls

- **What problems could this specific application have?**
  - Scalability issues for programs with non-trivial behaviour
    - ***"State-explosion problem"*** *(more on this later...)*

```
S0;
while (..) {
      S1;
      if (...) S2;
      else S3;
      if (S4) ... ;
      else S2;
      S5;
}
S3;
S4;
```

$S_0S_1S_2$	$S_1S_2S_4$	$S_2S_4S_5$	$S_3S_4S_5$	$S_4S_5S_1$	$S_2S_5S_1$	$S_5S_1S_2$
$S_0S_1S_3$	$S_1S_3S_4$	$S_2S_4S_2$	$S_3S_4S_2$	$S_4S_5S_3$	$S_2S_5S_3$	$S_5S_1S_3$
$S_0S_3S_4$				$S_4S_2S_5$		$S_5S_3S_4$

Figure 1. An example program and associated trigrams. S0,...,S5 **denote system calls.**

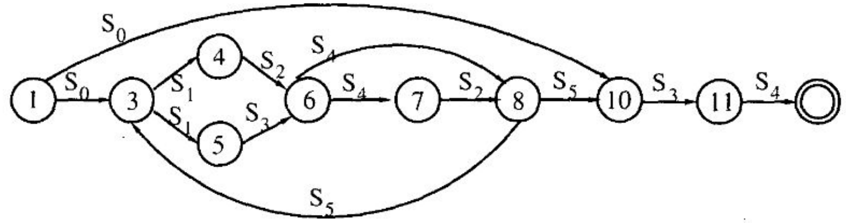
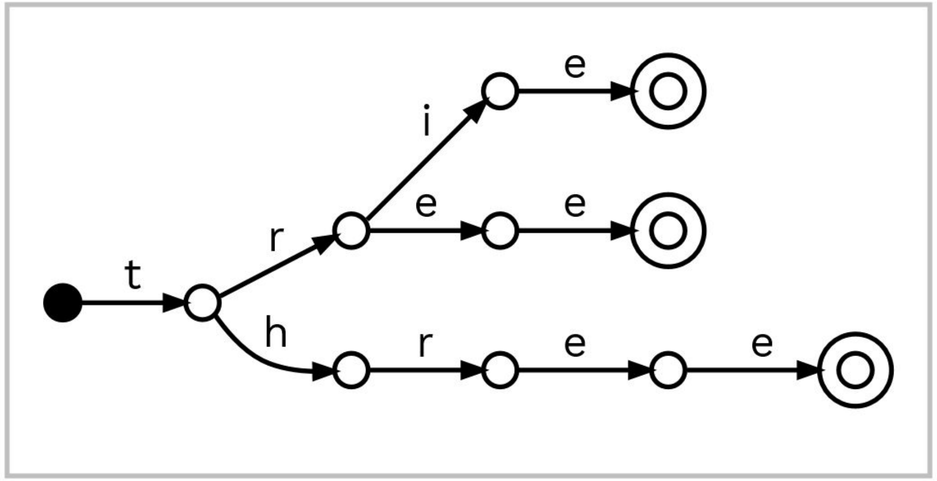


Figure 2. Automaton learnt by our algorithm for Example 1

Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. (2000, May). A fast automaton-based method for detecting anomalous program behaviors. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001 (pp. 144-155). IEEE.

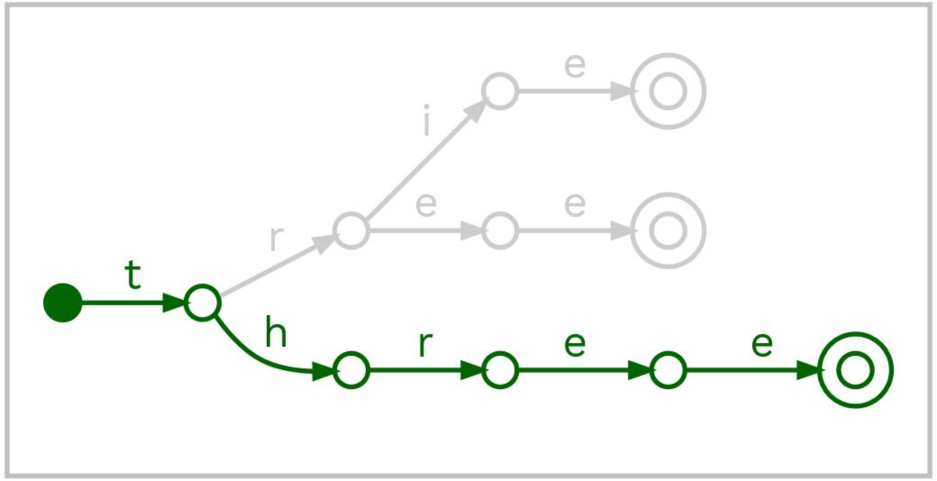
Building FSMs – *tries*

- **Trie:** FSM as a rooted tree associated with a set of words
  - Paths from initial to accepting states represent words from its set
  - **Example:** trie for the set { *three*, *tree*, *trie* }



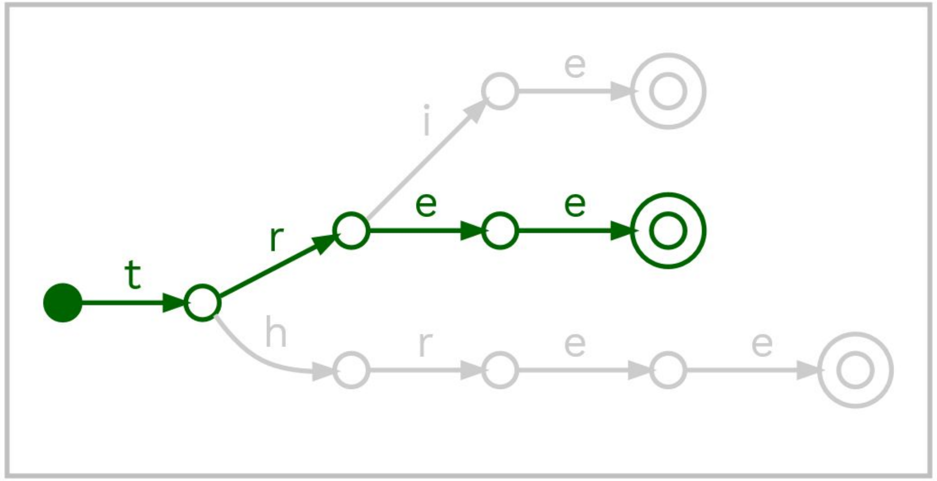
Building FSMs – *tries*

- **Trie:** rooted tree associated with a set of words
  - Paths from initial to accepting states represent words from its set
  - **Example:** trie for the set { ***three***, *tree*, *trie* } ✅



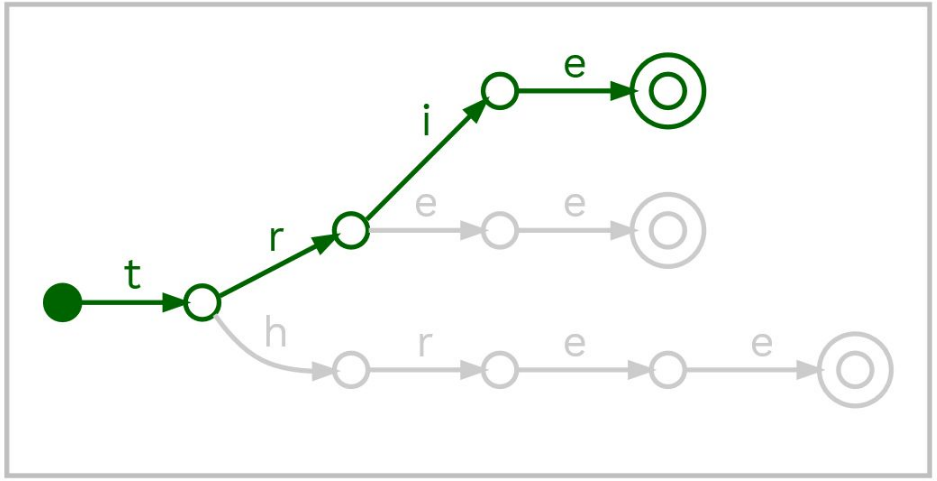
Building FSMs – *tries*

- **Trie:** rooted tree associated with a set of words
  - Paths from initial to accepting states represent words from its set
  - **Example:** trie for the set { *three*, ***tree***, *trie* } ✅



Building FSMs – *tries*

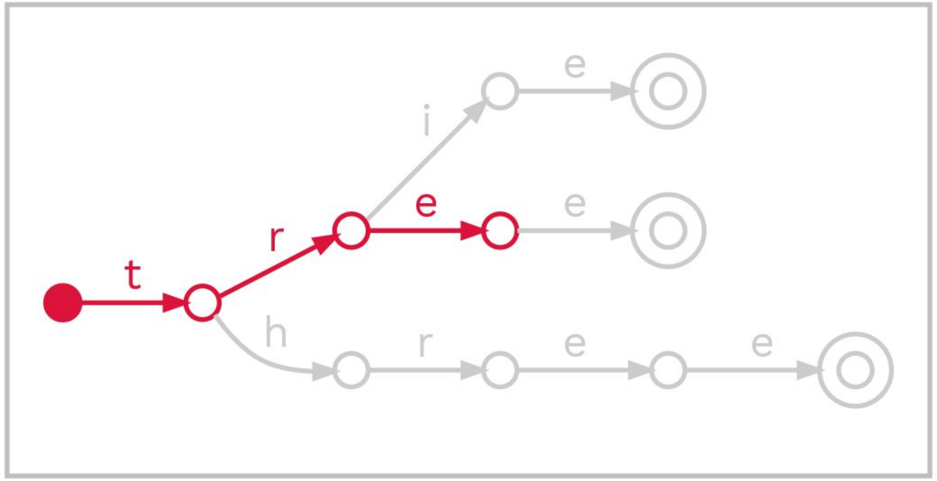
- **Trie:** rooted tree associated with a set of words
  - Paths from initial to accepting states represent words from its set
  - **Example:** trie for the set { *three*, *tree*, ***trie*** } ✅



Building FSMs – *tries*

- **Trie:** rooted tree associated with a set of words
  - Paths from initial to accepting states represent words from its set
  - **Example:** trie for the set { *three, tree, trie* }

tre is not part of the set { *three, tree, trie* } ❌



Construction Algorithm – *tries*

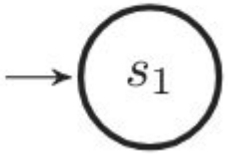
```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

{ *trie*, *tree* }

Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

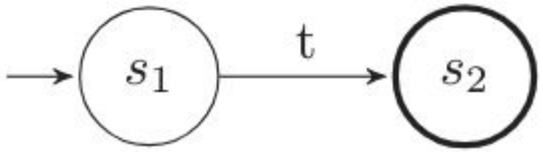
{ *trie*, *tree* }



Construction Algorithm – *tries*

```
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

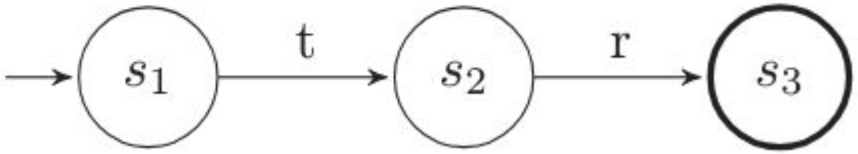
{ ***t**rie*, *tree* }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

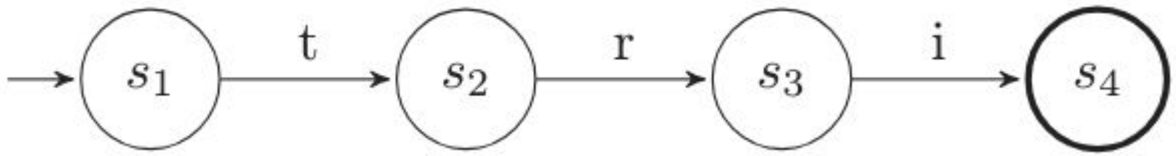
{ *trie*, *tree* }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

{ *trie*, *tree* }

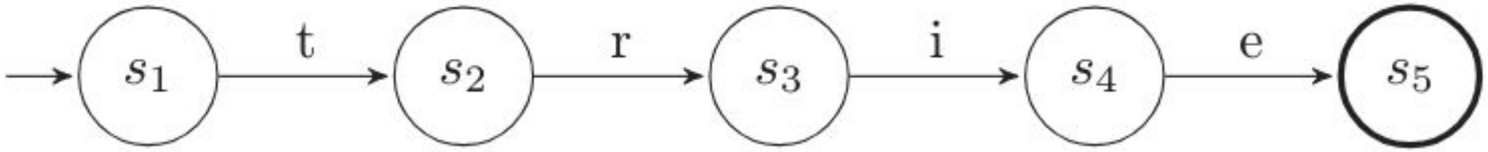


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

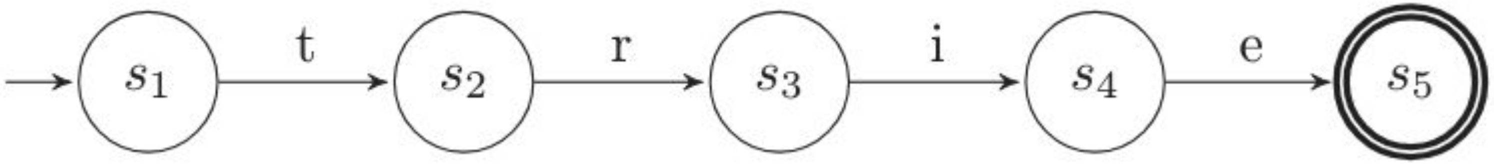
{ *trie*, *tree* }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

{ *trie*, *tree* }

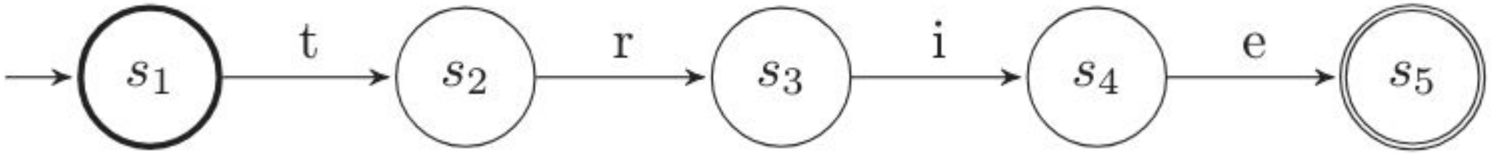


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

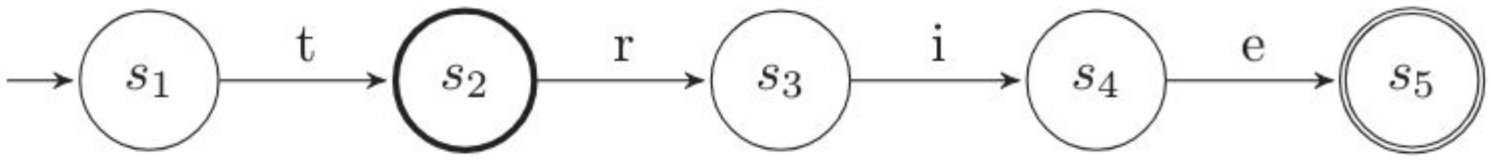
{ *trie*, *tree* }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

{ *trie*, ***t**ree* }

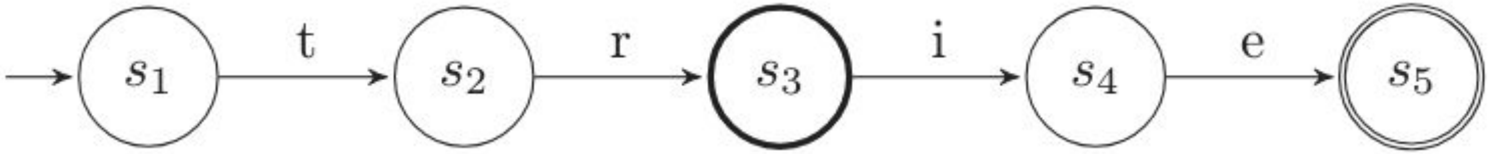


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Construction Algorithm – *tries*

```
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

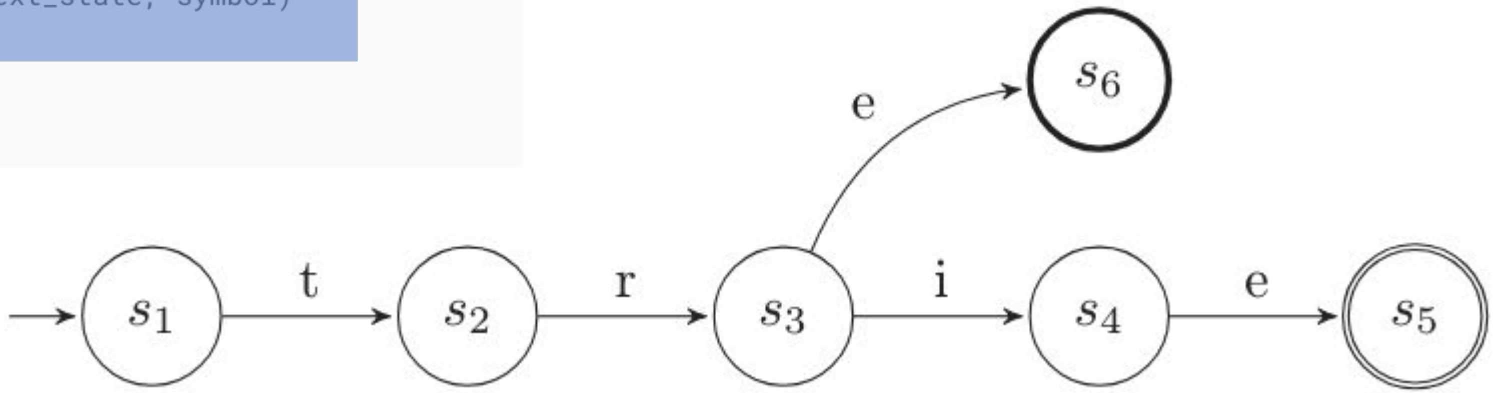
{ *trie*, *tree* }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

{ *trie*, *tree* }

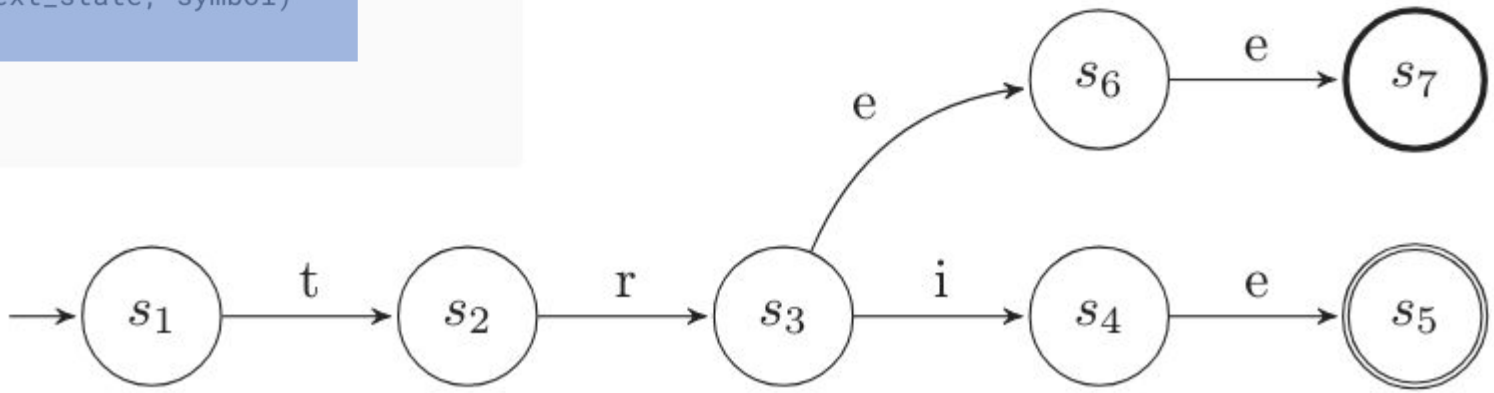


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

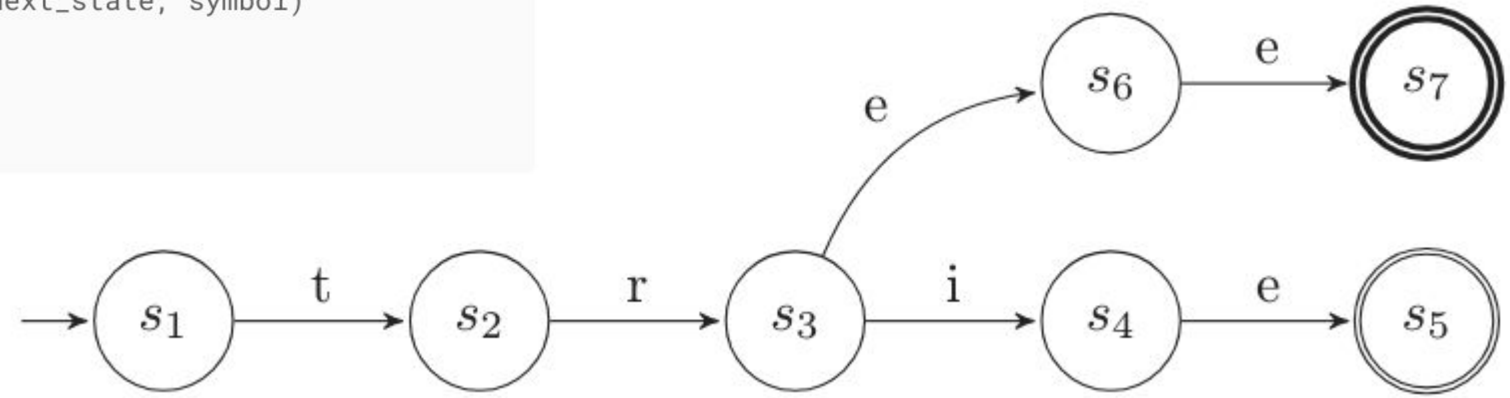
{ *trie*, ***tre*e** }



Construction Algorithm – *tries*

```python
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

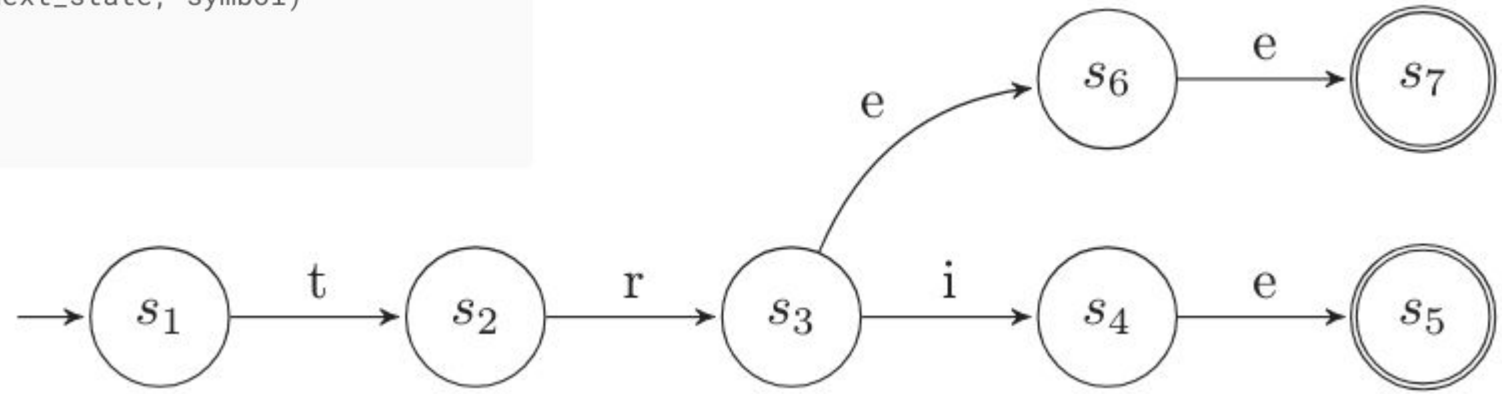
{ *trie*, *tree* }



Construction Algorithm – *tries*

```
def trie(sequences):
    """Constructs a trie from a set of sequences."""
    fsm = Automaton()
    for sequence in sequences:
        current = fsm.initial
        for symbol in sequence:
            if symbol not in fsm.alphabet:
                fsm.add_symbol(symbol)
            if (next_state := fsm.follow(current, symbol)) is None:
                next_state = fsm.add_state()
                fsm.set_transition(current, next_state, symbol)
            current = next_state
        fsm.accepting.add(current)
    return fsm
```

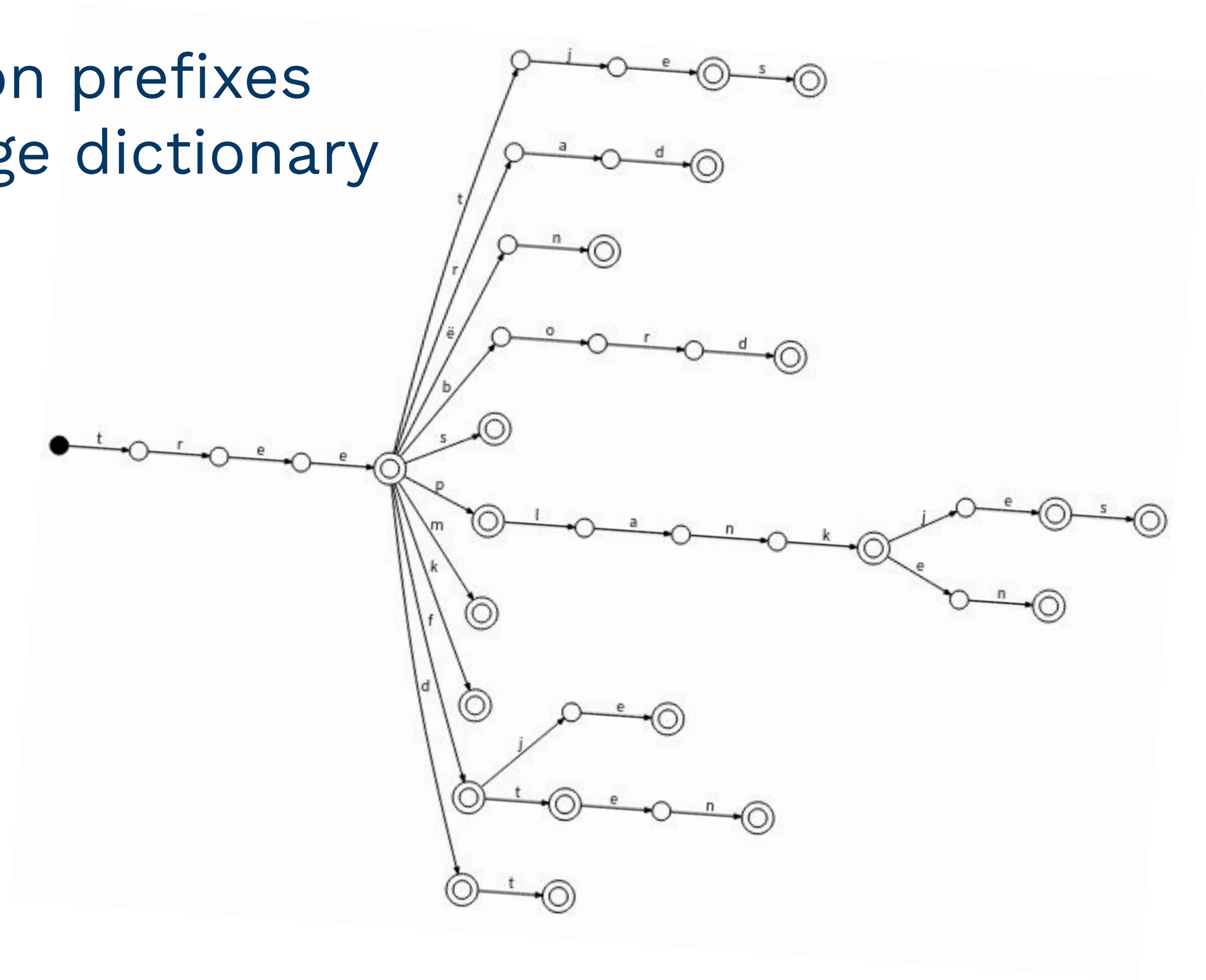
{ *trie*, *tree* } ✅



ARTIFICIAL INTELLIGENCE RESEARCH GROUP

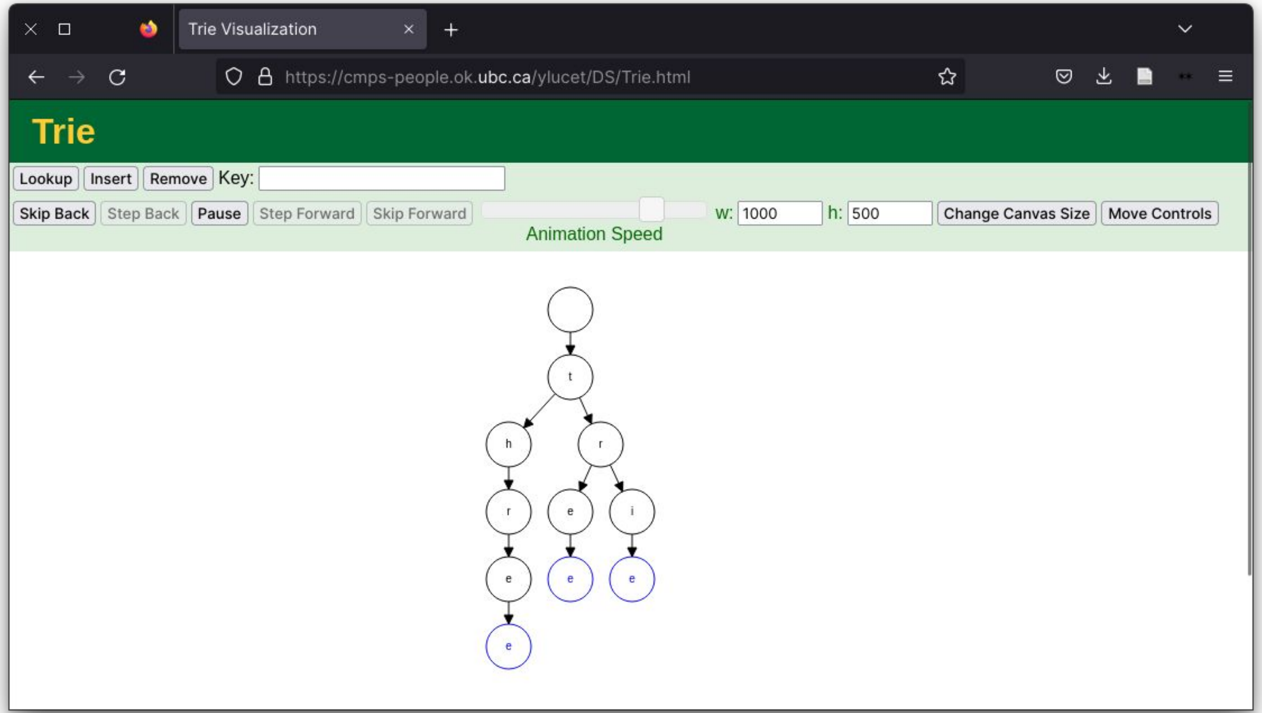
Applications – *tries*

- **Efficient data structure:**
  - Storing words based on common prefixes
  - May require less storage for large dictionary
  - Allows for efficient search



ARTIFICIAL INTELLIGENCE RESEARCH GROUP

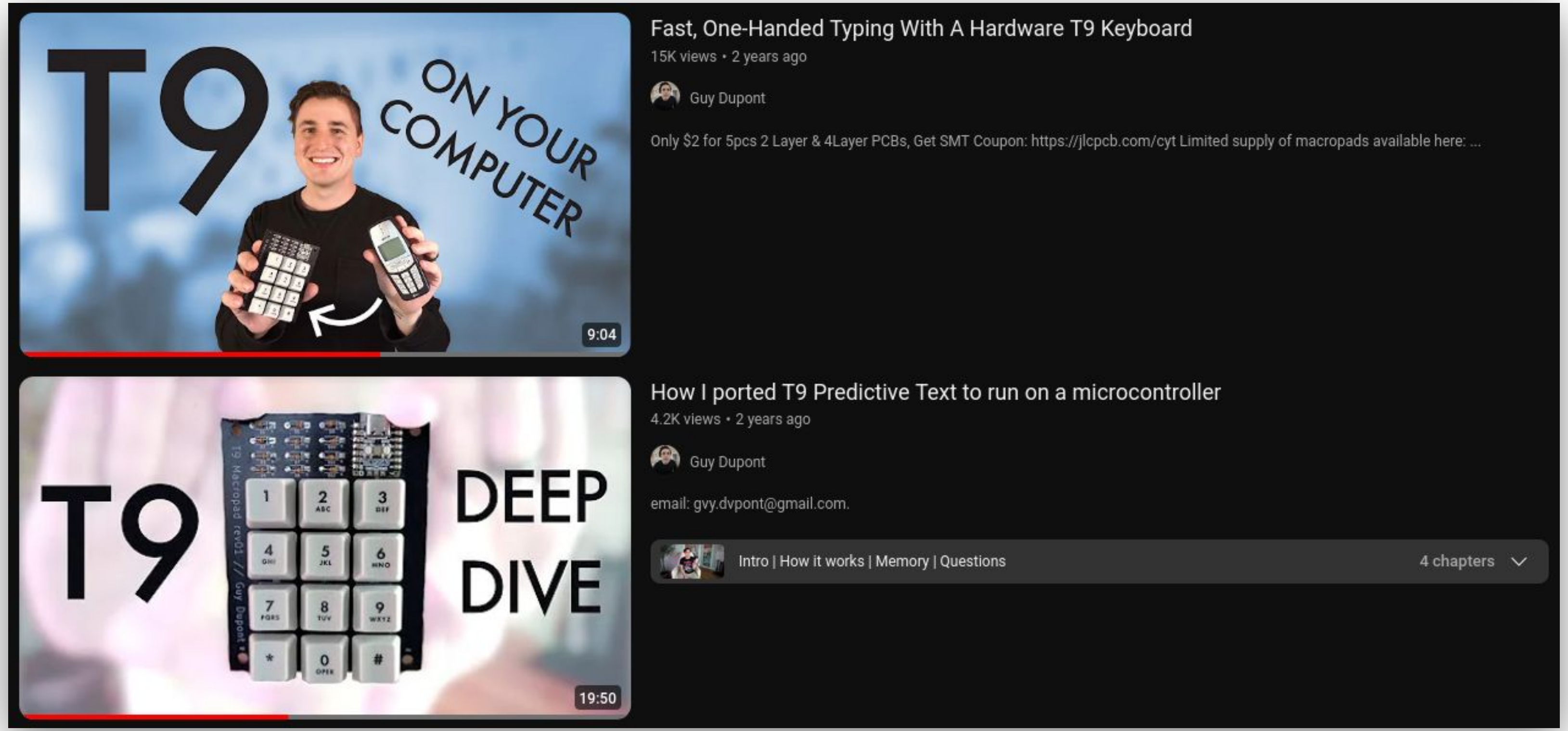
Demonstration



https://cmps-people.ok.ubc.ca/ylucet/DS/Trie.html

ARTIFICIAL INTELLIGENCE RESEARCH GROUP

What is it used for?



https://youtu.be/6cbBSEbwLUI

Building FSMs – *factor automaton*

- **Factor automaton of a word *x*:**
  - The minimal deterministic automaton that recognises the factors of *x*

- **Remember:**
  - **Factors:** <u>contiguous</u> subsequences of a word

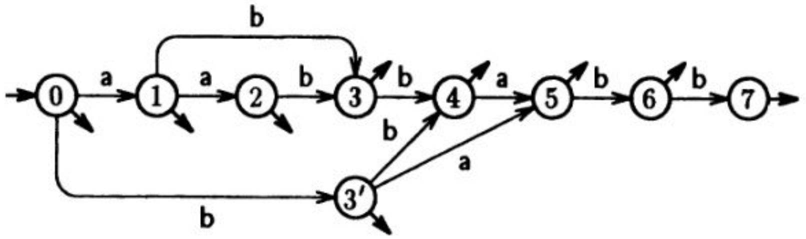


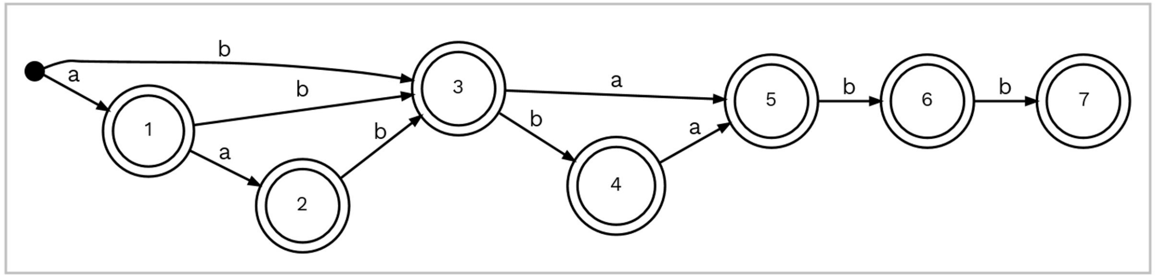
Fig. 7.11. Minimal deterministic automaton recognizing the factors of aabbabb.

Crochemore, M., & Hancart, C. (1997). Automata for matching patterns. Handbook of Formal Languages: Volume 2. Linear Modeling: Background and Application, 399-462.

ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Building FSMs – *factor oracle*

- **Factor oracle:**
  - Acyclic automaton built on a set of words, recognises **at least** all factors of every word of the set



- **Applications:**
  - Intended for multi-pattern matching

Allauzen, C., Crochemore, M., & Raffinot, M. (1999). Factor oracle: A new structure for pattern matching. In SOFSEM'99: Theory and Practice of Informatics: 26th Conference on Current Trends in Theory and Practice of Informatics Milovy, Czech Republic, November 27—December 4, 1999 Proceedings 26 (pp. 295-310). Springer Berlin Heidelberg.

Factor Automata vs. Factor Oracles

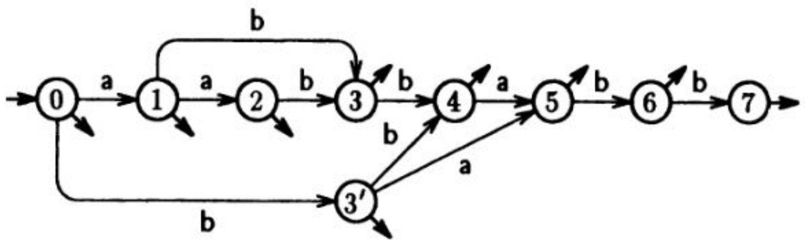
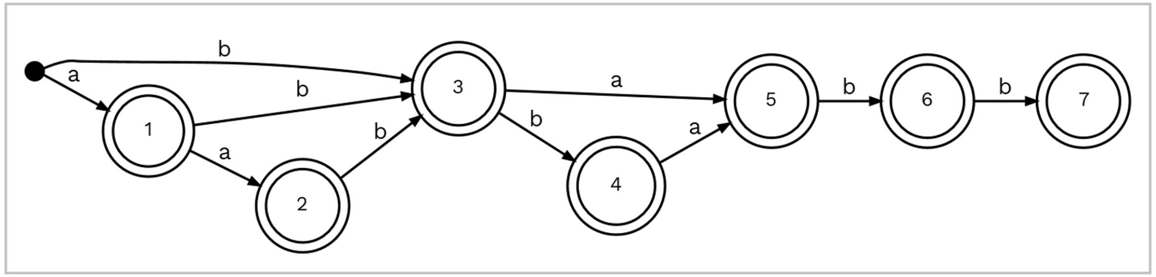


Fig. 7.11. Minimal deterministic automaton recognizing the factors of aabbabb.



- **Factor automaton of a word *x*:**
  - The minimal deterministic automaton that recognises the factors of *x*

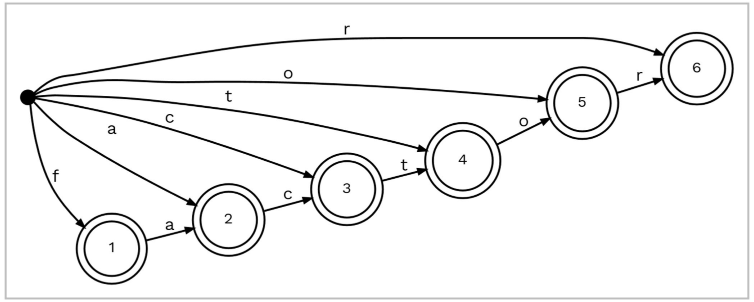
- **Factor oracle:**
  - Acyclic automaton built on a set of words, recognises **at least** all factors of every word of the set

    ("abab" is not a factor!)

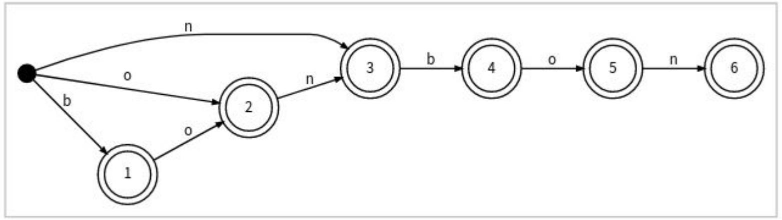
Building FSMs – *factor oracle*

- **States and transitions:**
  - Has exactly *m + 1* states
  - Between *m* and *2m - 1* transitions

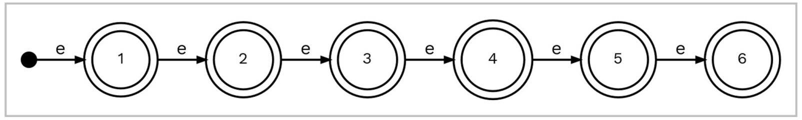
- **Examples:**



"factor"



"bonbon"



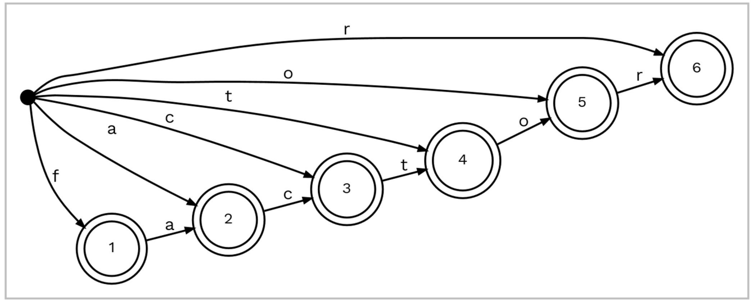
"eeeeee"

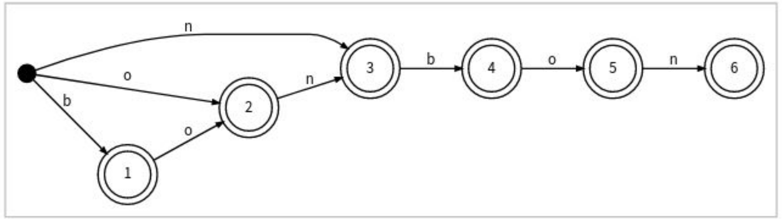
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

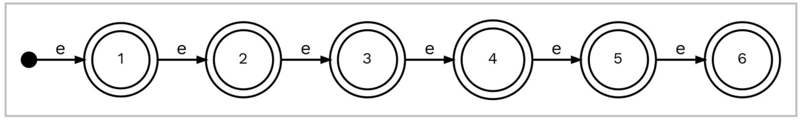
Building FSMs – *factor oracle*

- **States and transitions:**
  - Has exactly *m + 1* states
  - Between *m* and *2m - 1* transitions

- **Examples:**







“factor”
Worst case

“bonbon”
Average case

“eeeeee”
Best case

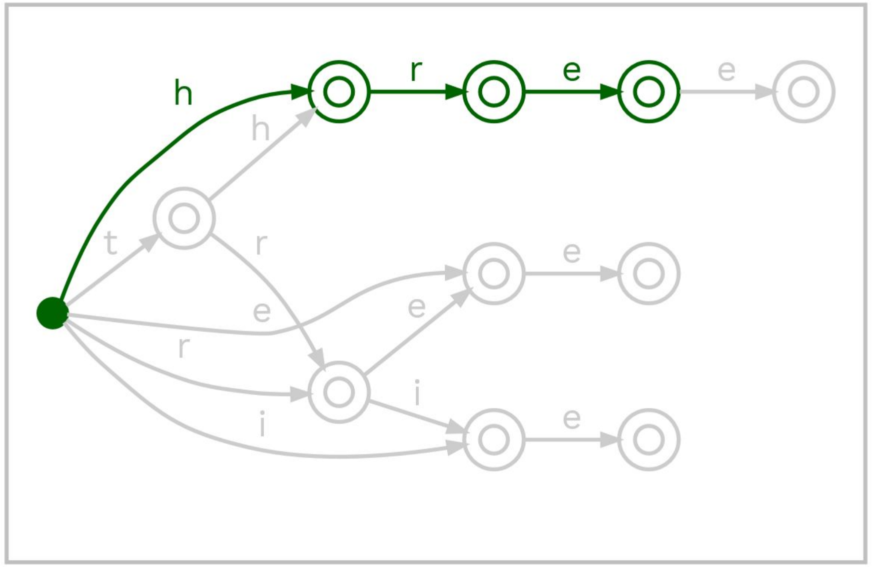
Why?

ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Building FSMs – *factor oracle*

- **Accepts at least all factors:**
  - Factor oracle for { *three*, *trie*, *tree* }

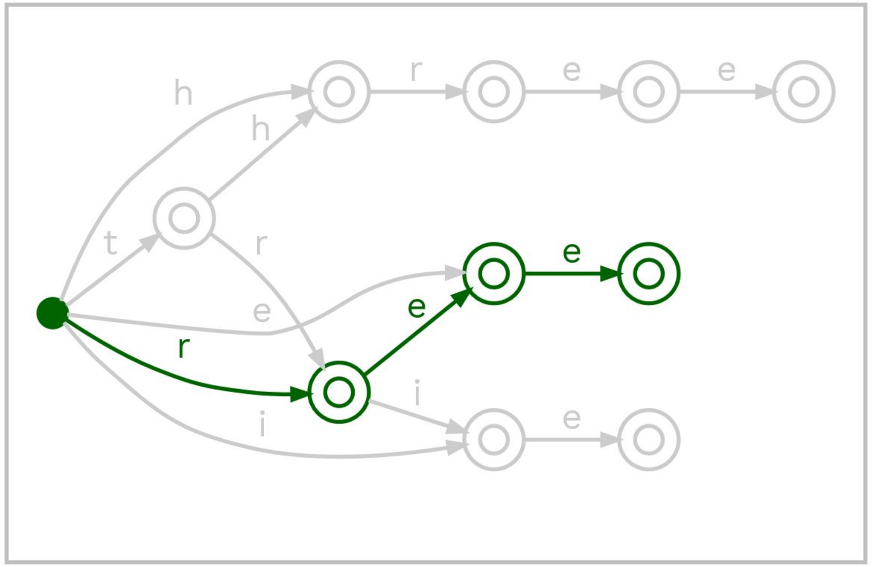
- **Examples:**
  - Accepts "*hre*", factor of t**hre**e



Building FSMs – *factor oracle*

- **Accepts at least all factors:**
  - Factor oracle for { *three, trie, tree* }

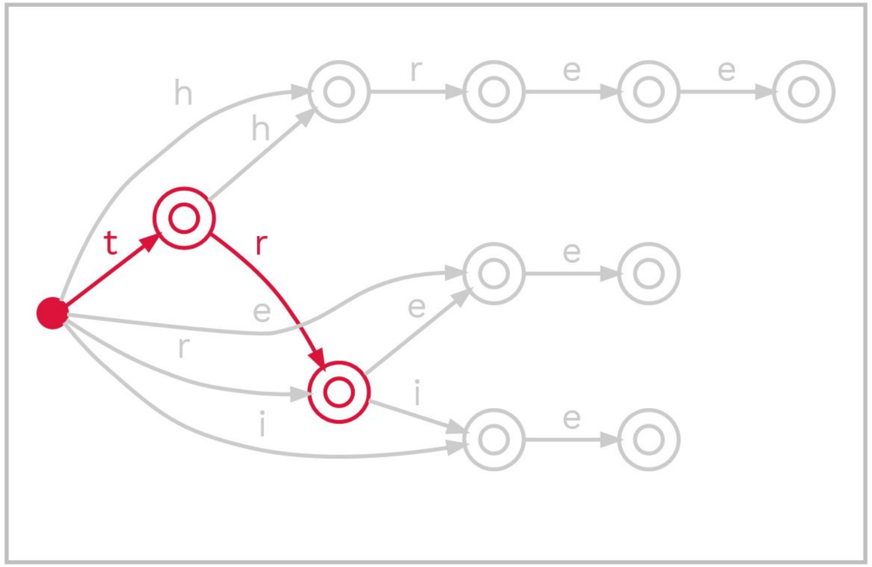
- **Examples:**
  - Accepts “*ree*”, factor of both th**ree** and t**ree**



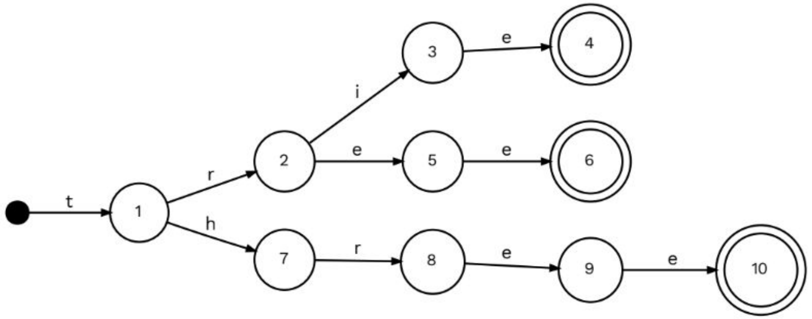
Building FSMs – *factor oracle*

- **Accepts at least all factors:**
  - Factor oracle for { *three*, *trie*, *tree* }

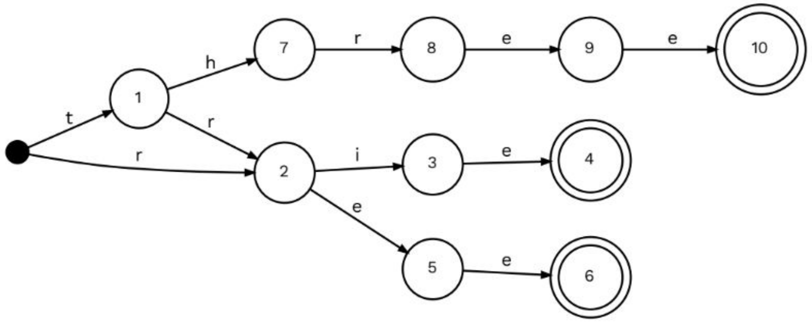
- **Examples:**
  - Does not accept “*trh*”, **not a factor** (and thus no transition labelled ‘h’)



Construction Algorithm – *factor oracle*

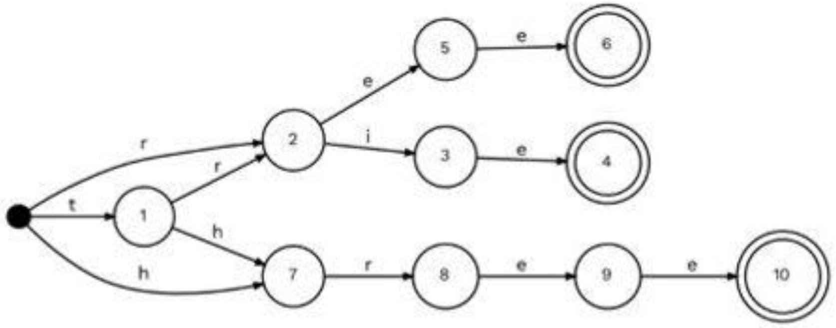


(a) The trie for $P = \{\texttt{trie}, \texttt{tree}, \texttt{three}\}$, taken from *Fig. 2.10*. In the first step of the algorithm, we generate the trie which we use as a starting point to generate the factor oracle.

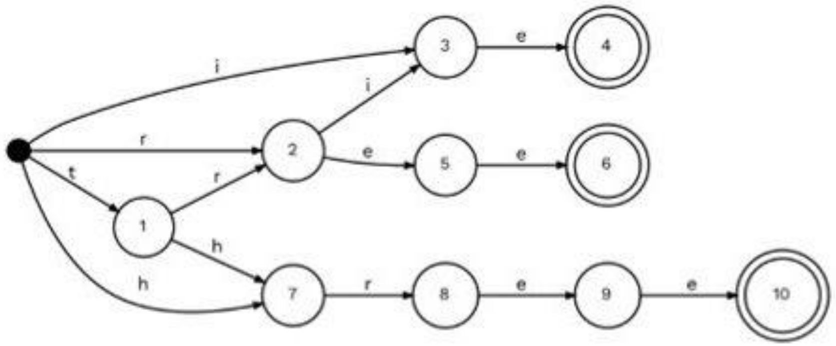


(b) In this step of the construction algorithm, a new transition **from the initial state to state 2** is made, labelled with r. There are now 11 transitions in total.

Construction Algorithm – *factor oracle*



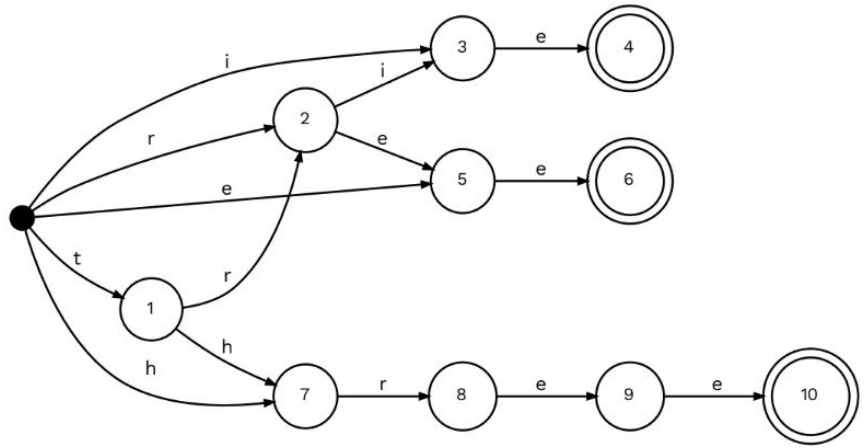
(c) In this step of the construction algorithm, a new transition **from the initial state to state 7** is made, labelled with h. There are now 12 transitions in total.



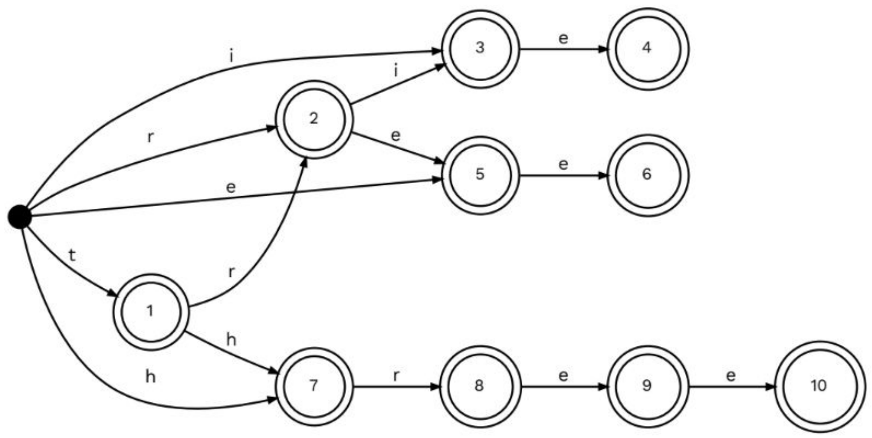
(d) In this step of the construction algorithm, a new transition **from the initial state to state 3** is made, labelled with i. There are now 13 transitions in total.

ARTIFICIAL
INTELLIGENCE
RESEARCH GROUP

Construction Algorithm – *factor oracle*



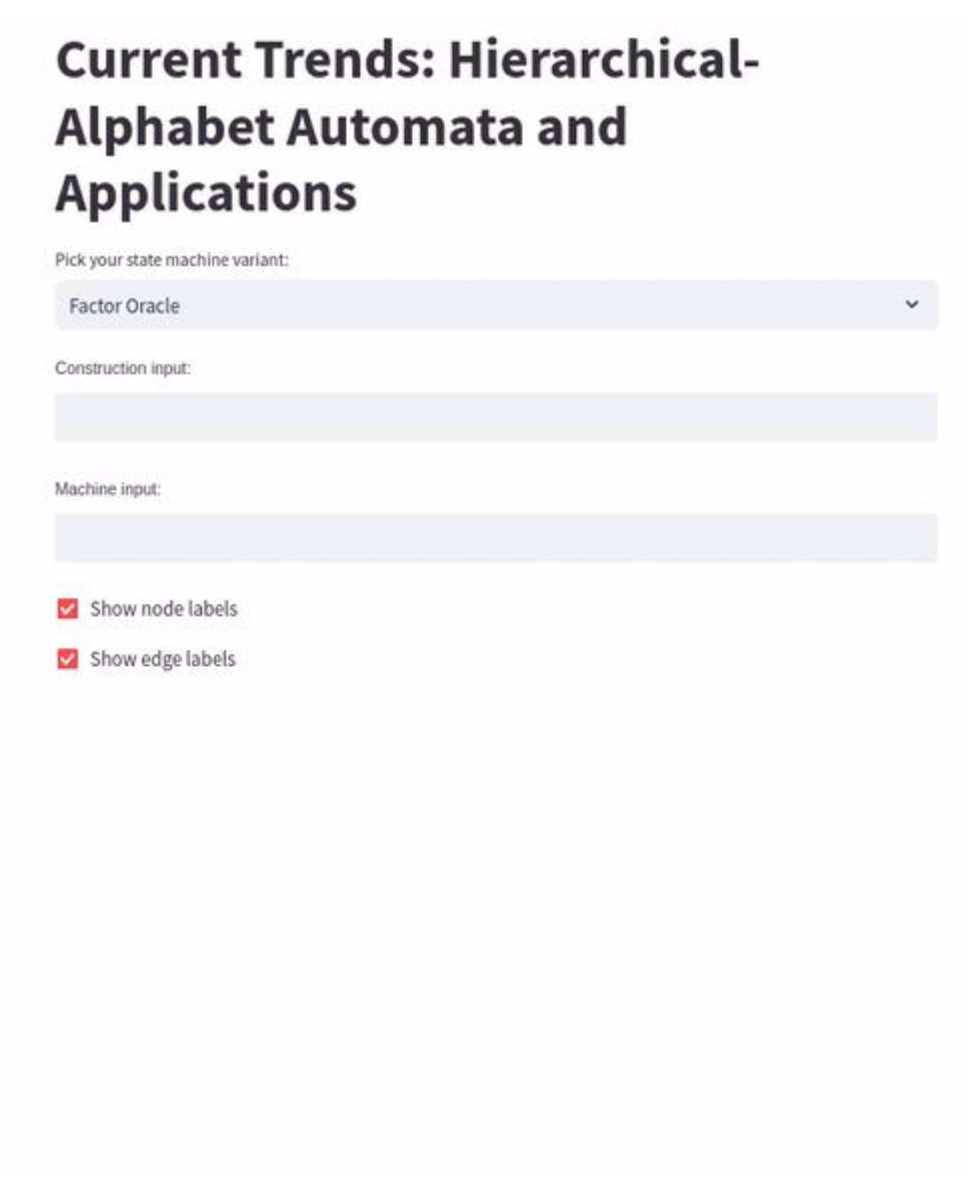
(e) In this step of the construction algorithm, a new transition **from the initial state to state 5** is made, labelled with e. There are now 14 transitions in total.



(f) In the final step of the construction algorithm, **all states are marked as final states**. The algorithm transformed the original trie to the factor oracle of $P = \{\texttt{trie},\ \texttt{tree},\ \texttt{three}\}$.

ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Demonstration



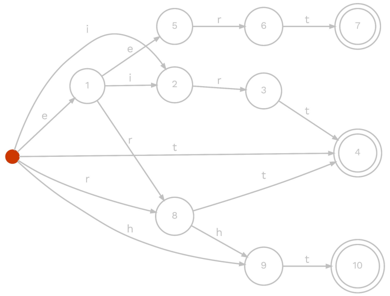
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

What is it used for?

- **Pattern matching:**
  - Backwards Oracle Matching (BOM)
  - Set Backwards Oracle Matching (SBOM)

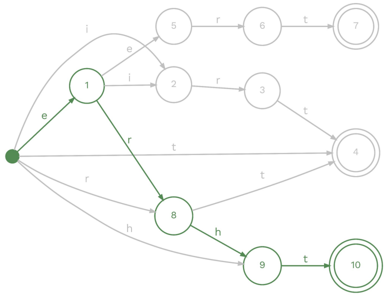
two␣or␣three␣trees

(a) We attempted to input ␣ in the factor oracle, but the oracle rejects directly. We shift the window after the ␣ and continue our search.



two␣or␣three␣trees

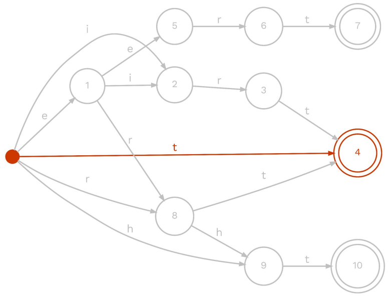
(c) We successfully input *e*, *r*, *h*, and *t* in the oracle. We reach accepting state 10, which is associated with the word *three*. We check for an occurrence of *three*, and shift the window by 1.



...

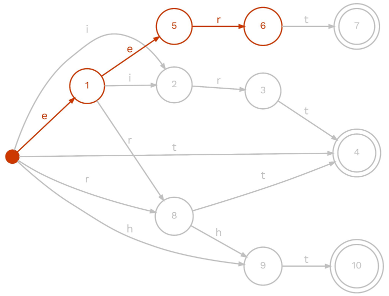
two␣or␣three␣trees

(b) We successfully input *t* in the factor oracle, but the oracle fails on ␣. We shift the window after the ␣ and continue our search.



two␣or␣three␣trees

(d) We successfully input *e*, *e*, and *r* in the oracle, but the oracle fail on the character *h*. We shift the window after the *h* and continue our search.



ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Applications of FSMs

- **Anomaly detection:**
  - Learning a language from sequences of system calls
- **FOs for anomaly detection?**
  - Accept factors of allowed system call sequences
- **Problems:**
  - Evolving behaviour
  - Difficulties with broad behaviour
    - *State-explosion problem?*

```
1.  S0;
2.  while (..) {
3.      S1;
4.      if (...) S2;
5.      else S3;
6.      if (S4) ... ;
7.      else S2;
8.      S5;
9.  }
10. S3;
11. S4;
```

$S_0S_1S_2$	$S_1S_2S_4$	$S_2S_4S_5$	$S_3S_4S_5$	$S_4S_5S_1$	$S_2S_5S_1$	$S_5S_1S_2$
$S_0S_1S_3$	$S_1S_3S_4$	$S_2S_4S_2$	$S_3S_4S_2$	$S_4S_5S_3$	$S_2S_5S_3$	$S_5S_1S_3$
$S_0S_3S_4$				$S_4S_2S_5$		$S_5S_3S_4$

Figure 1. An example program and associated trigrams. S0,...,S5 **denote system calls.**

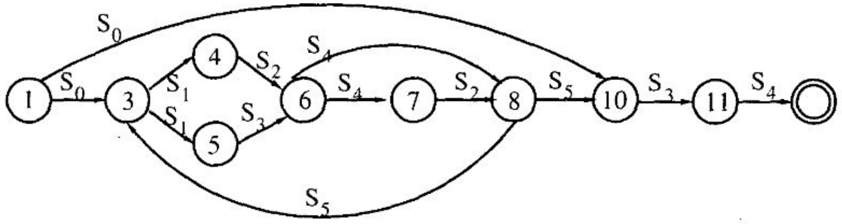


Figure 2. Automaton learnt by our algorithm for Example 1

Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. (2000, May). A fast automaton-based method for detecting anomalous program behaviors. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001 (pp. 144-155). IEEE.

Example – *Even and Odd*

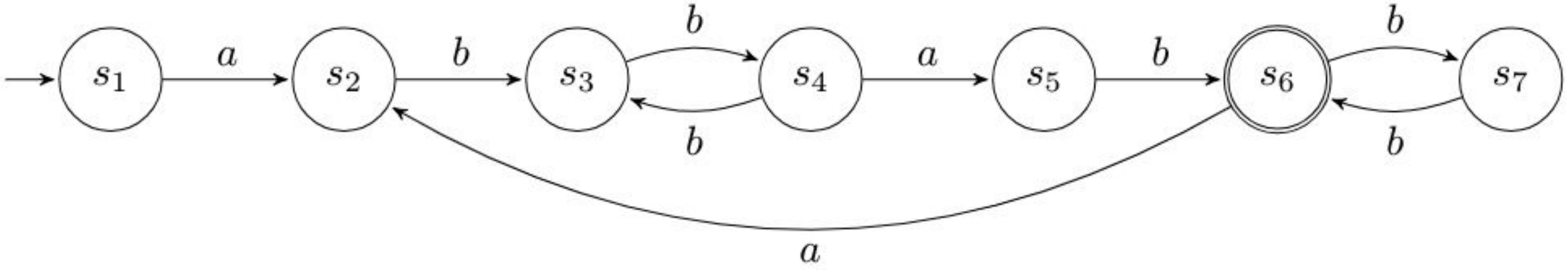
Even-odd binary words

Create a machine that accepts words starting with a, followed by alternating even and odd number of repetitions of b separated by one a, where the word always ends with an odd number of b:

$$(a(bb)^+ab(bb)*)^+$$

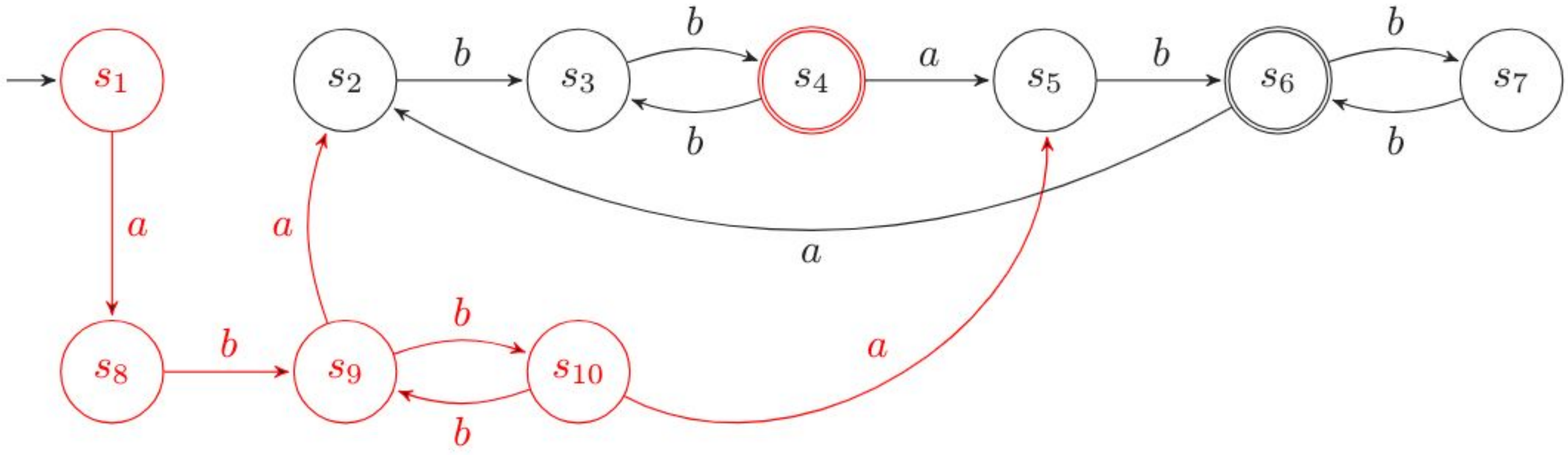
abbab ✅
abbabbb ✅
abbbbababb ✅
...

Regular FSM



Example – *Even and Odd*

- **Evolving behaviour:**
  - What if we can now also start with an odd number of occurrences of *b*?



Example – *Even and Odd*

- **Evolving behaviour:**
  - Now, what if we alternate between alternating from even to odd to even, and repeating even to even and odd to odd?

- **Problem:**
  - Building on previous machine requires us to make *too many changes*
  - There is no modularity to exploit, and machine gets too complex

Research Goals

- **Solving previously-mentioned issues:**
  - Capturing broad behaviour with compact FSMs
  - Easily change model when behaviour changes or evolves

- **Our hypothesis:**
  - We can exploit hierarchical relationships of symbols to create compact, less complex, modular machines

- **How do we do this?**
  - **Hierarchical-alphabet automata (HAA):** introducing hierarchy in machines *and* alphabet

Hierarchical-Alphabet Automata

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

Partial Order

An **order** (or partial order) on P is a binary relation $\leq$ on P such that, for all $x, y, z \in P$,

1. $x \preceq x$ (*reflexivity*)
2. $x \preceq y$ and $y \preceq x$ imply $x = y$ (*antisymmetry*)
3. $x \preceq y$ and $y \preceq z$ imply $x \preceq z$ (*transitivity*)

Partially Ordered Set (poset)

A poset $P = (X, \preceq)$ is a pair consisting of a set X and the partial order $\preceq$ on X

Hierarchical-Alphabet Automata

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

Directed Acyclic Graph (DAG)

A **directed acyclic graph (DAG)** is a directed graph with no directed cycles.

ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Hierarchical-Alphabet Automata

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

Example

A poset $P = (M, \preceq)$ consisting of a set $M = \{m_1, m_2, m_3\}$ with m_1 as the greatest element. We can represent this poset P as a DAG with $m_i \in M$ as the vertices, m_1 as the source node, and the covering relation $(m_i \gtrdot m_j)$ as directed edges.

Covering Relation

Let X be a set with a partial order $\preceq$. Let $\prec$ be the relation on X such that $x \prec y \iff x \preceq y \land x \neq y$ with $x, y \in X$. Then y **covers** x if $x < y$ and there is no element $z \in X$ such that $x < z < y$.

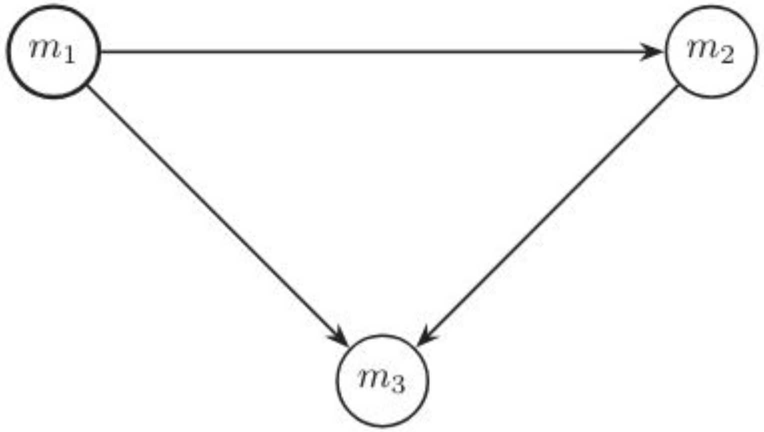
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Hierarchical-Alphabet Automata

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

Example

A poset $P = (M, \preceq)$ consisting of a set $M = \{m_1, m_2, m_3\}$ with m_1 as the greatest element. We can represent this poset P as a DAG with $m_i \in M$ as the vertices, m_1 as the source node, and the covering relation $(m_i \gtrdot m_j)$ as directed edges.

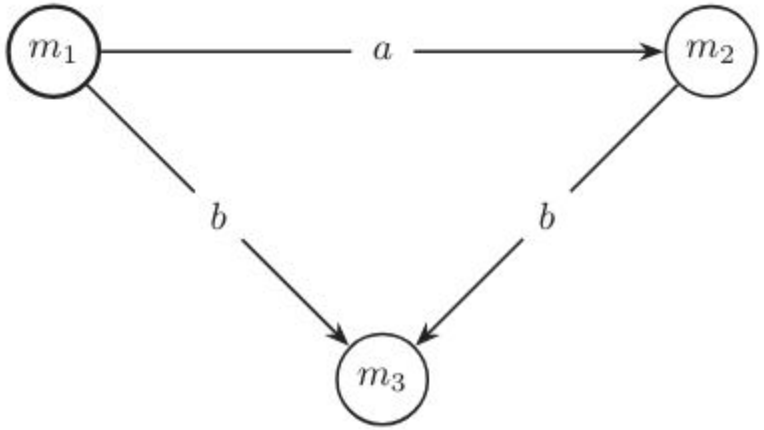


Hierarchical-Alphabet Automata

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

We also introduce an **edge labelling** to the edges of the DAG:

For every $m_i \in M$, there is at most one outgoing edge labelled with $\sigma \in \Sigma_{m_i}$.



Hierarchical-Alphabet Automata

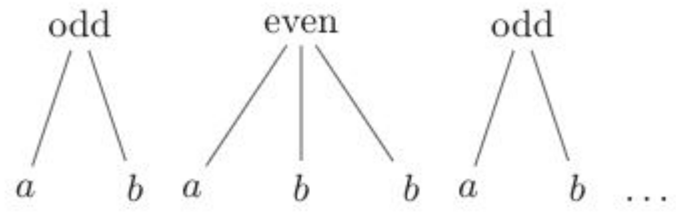
- **Input of the HAA:**
  - Sequence of **ordered trees**
    - **Leafs** are our observations
    - **Internal nodes** represent groupings of these observations

- ***Ordered trees:***
  - *Rotted tree where the order of the subtrees of a node are significant*

Rooted tree

A **rooted tree** is a finite set S of one or more nodes such that (a) there is one specially designated node, called the root of the tree, and (b) the remaining nodes are partitioned into $m \geq 0$ disjoint sets $S_1, \ldots, S_m$, and each of the sets in turn is a rooted tree. The trees $S_1, \ldots, S_m$ are called the subtrees of the root.

- Number of subtrees of a node = the ***degree*** of the node
  - Node of degree zero is called a ***leaf***
  - Node of positive degree is called an ***internal node***
  - Node of degree at least 2 is called a ***branching node***
- The ***level*** of a node is the number of separations from root



Hierarchical-Alphabet Automata

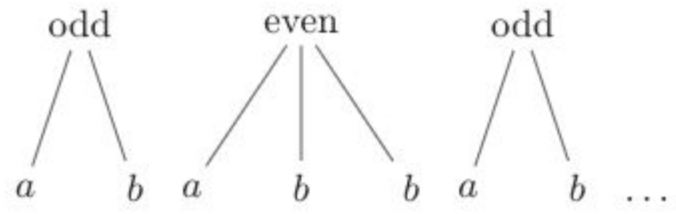
- **Input of the HAA:**
  - Sequence of **ordered trees**
    - **Leafs** are our observations
    - **Internal nodes** represent groupings of these observations

- ***Ordered trees:***
  - *Rooted tree where the order of the subtrees of a node are significant*

Rooted tree

A **rooted tree** is a finite set S of one or more nodes such that (a) there is one specially designated node, called the root of the tree, and (b) the remaining nodes are partitioned into $m \geq 0$ disjoint sets $S_1, \ldots, S_m$, and each of the sets in turn is a rooted tree. The trees $S_1, \ldots, S_m$ are called the subtrees of the root.

- Number of subtrees of a node = the ***degree*** of the node
  - Node of degree zero is called a ***leaf***
  - Node of positive degree is called an ***internal node***
  - Node of degree at least 2 is called a ***branching node***
- The ***level*** of a node is the number of separations from root

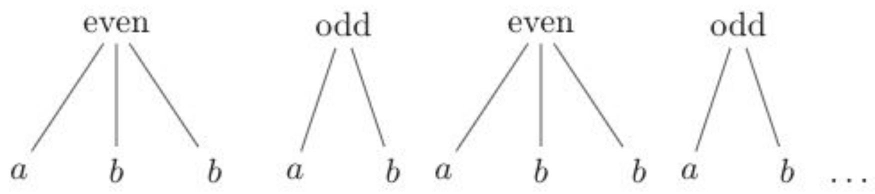


Example – *Even and Odd*

Even-odd binary words

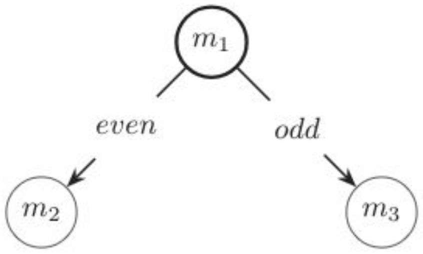
Create a machine that accepts words starting with a, followed by alternating even and odd number of repetitions of b separated by one a, where the word always ends with an odd number of b:

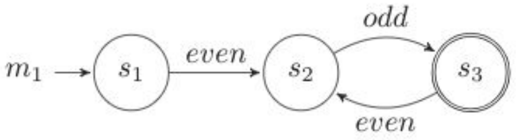
$$(a(bb)^{+}ab(bb)*)^{+}$$

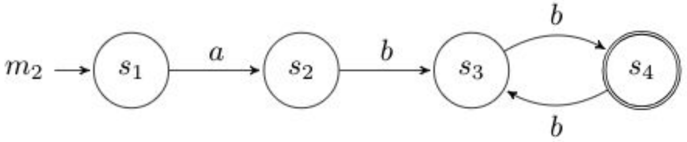


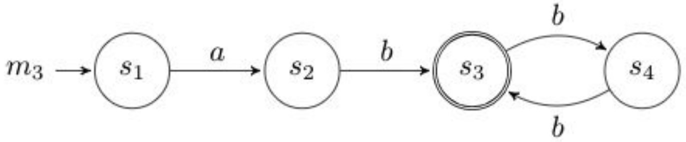
Hierarchical-Alphabet Automaton

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





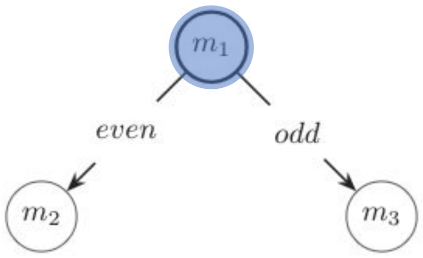


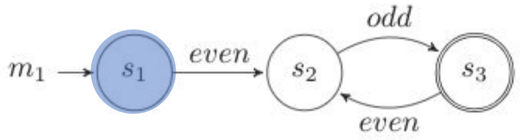


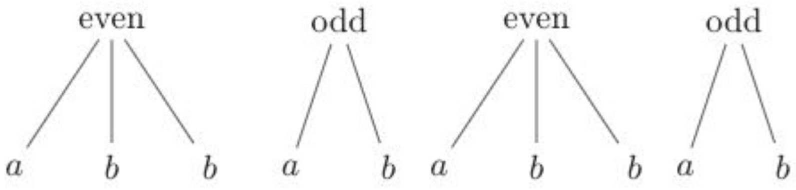
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

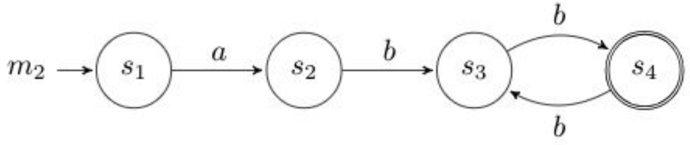
Hierarchical-Alphabet Automaton

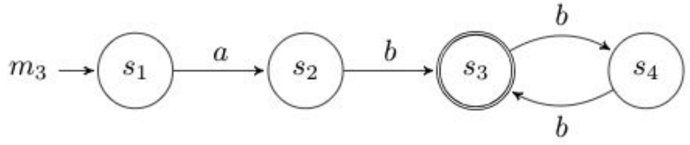
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





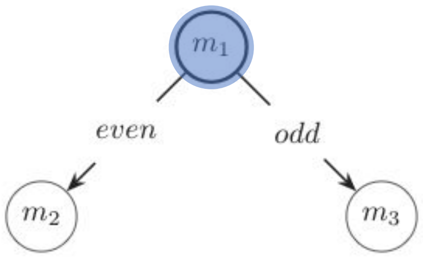


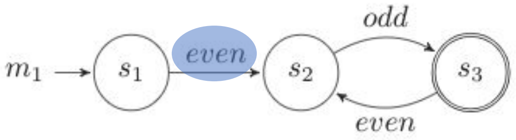


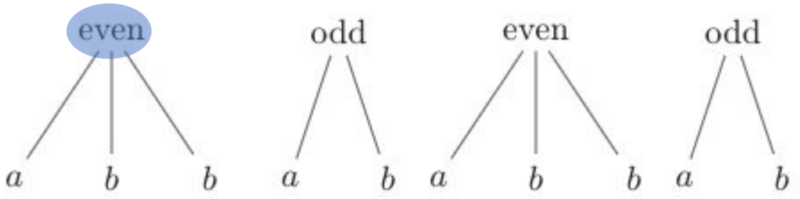


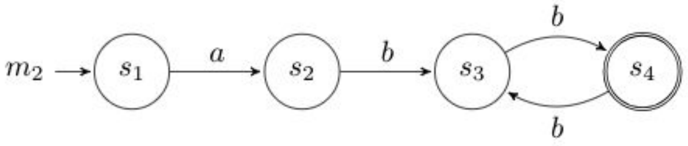
Hierarchical-Alphabet Automaton

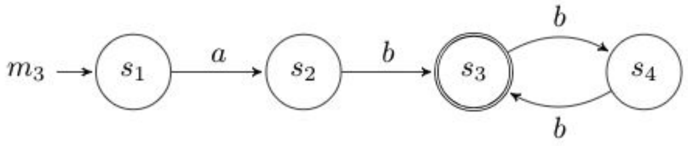
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.







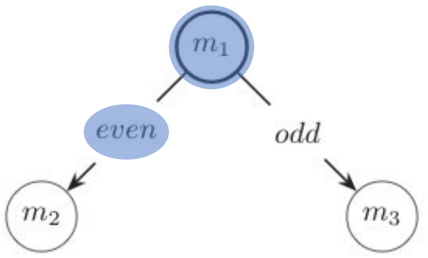


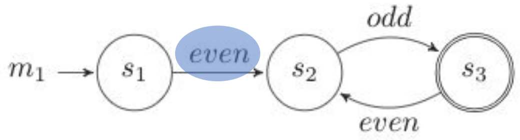


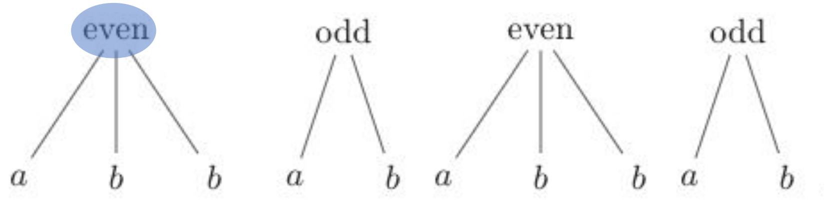
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

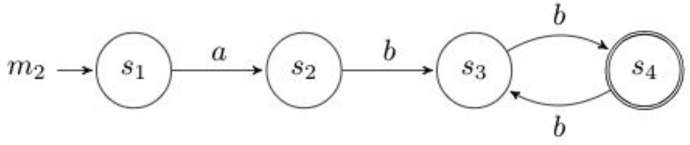
Hierarchical-Alphabet Automaton

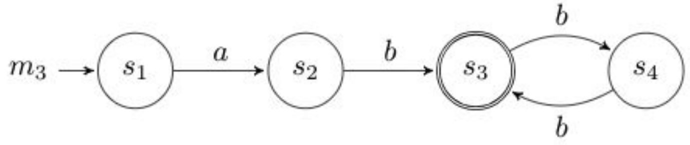
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





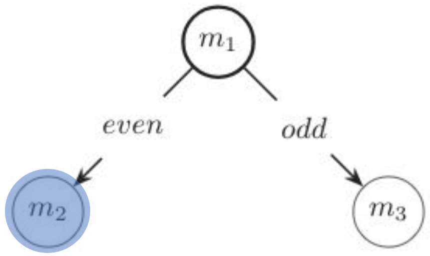


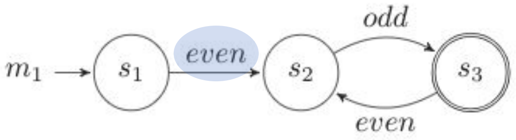


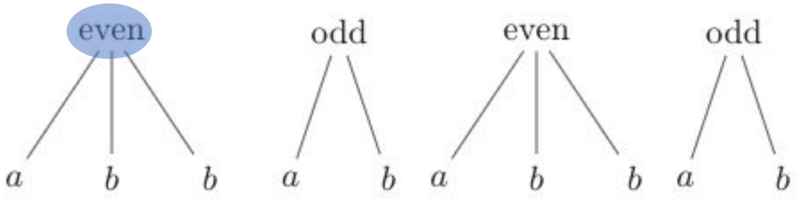


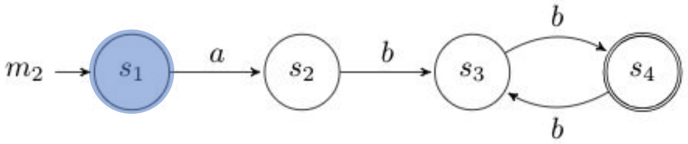
Hierarchical-Alphabet Automaton

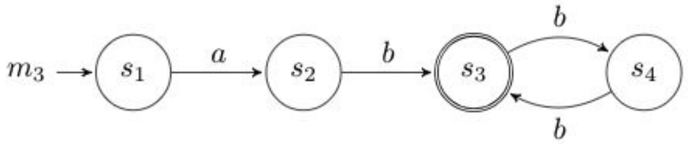
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





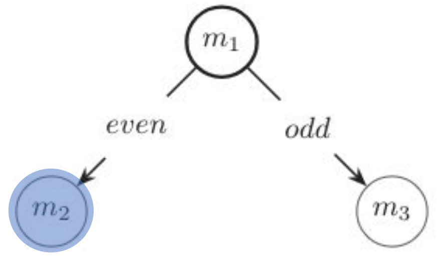


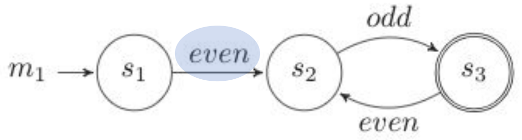


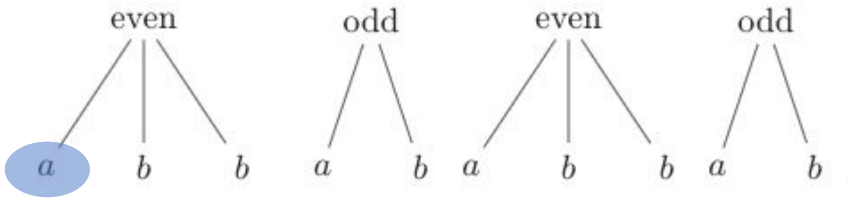


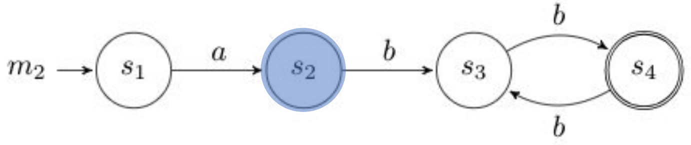
Hierarchical-Alphabet Automaton

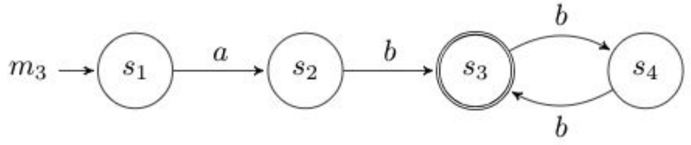
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





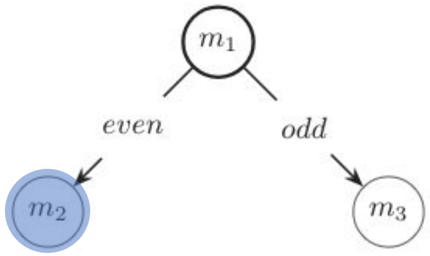


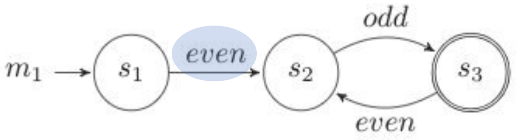


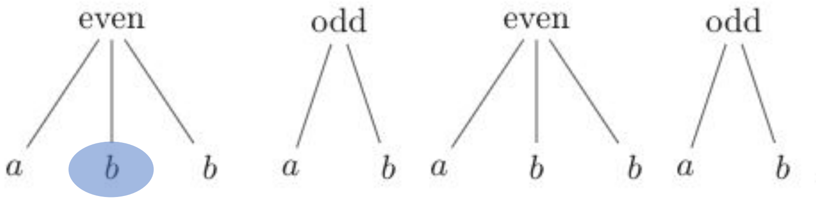


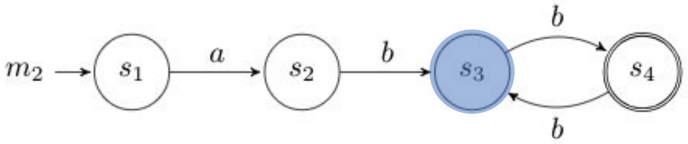
Hierarchical-Alphabet Automaton

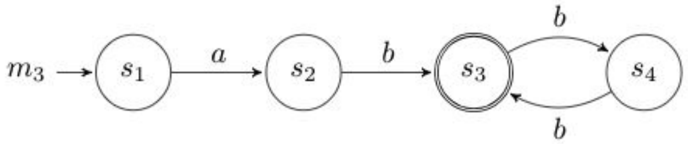
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





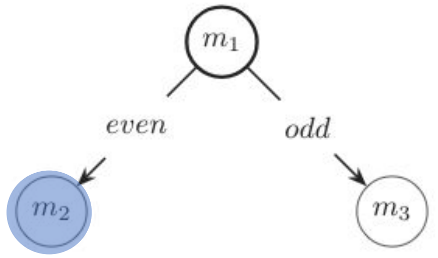


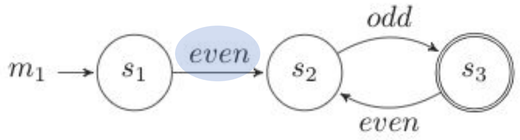


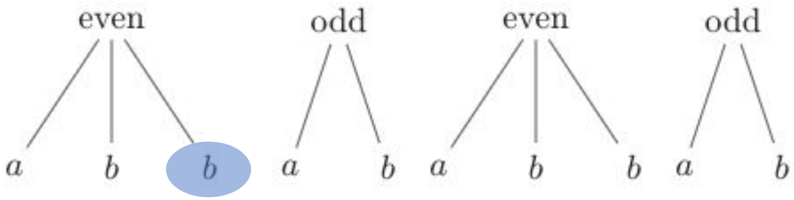


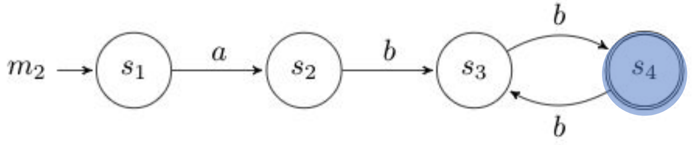
Hierarchical-Alphabet Automaton

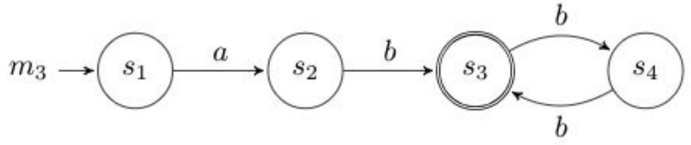
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.







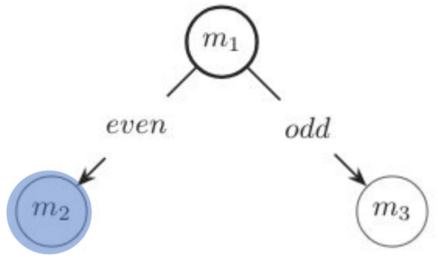


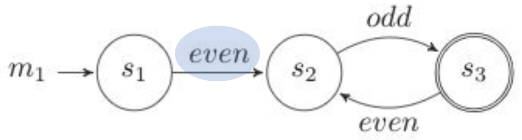


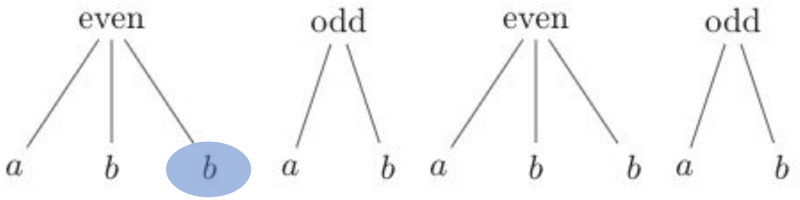
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

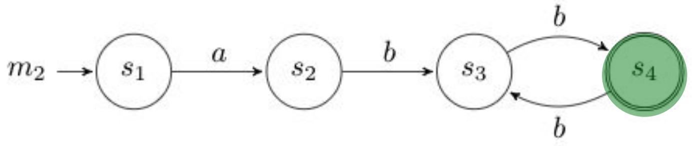
Hierarchical-Alphabet Automaton

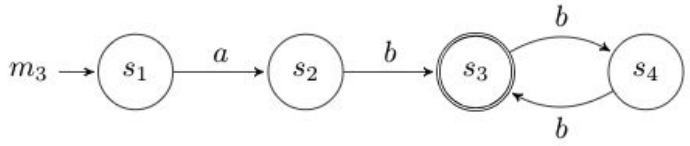
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





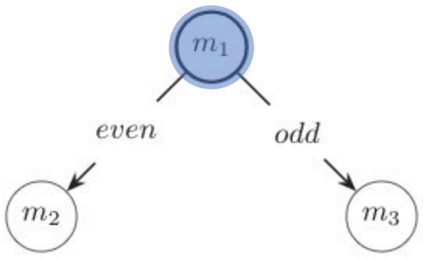


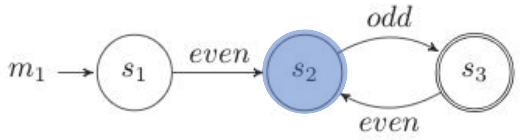


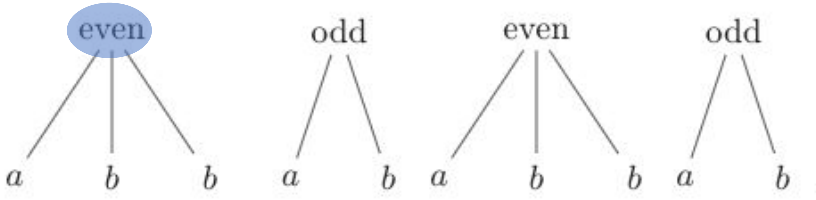


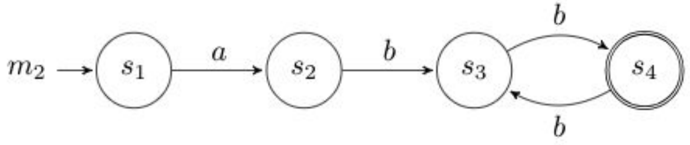
Hierarchical-Alphabet Automaton

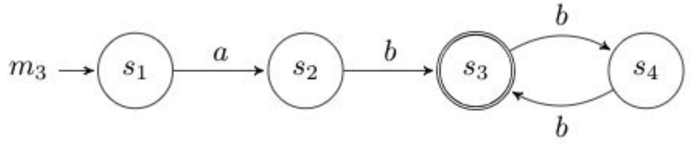
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.







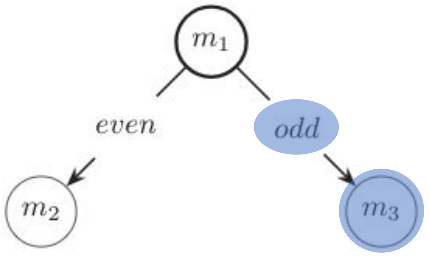


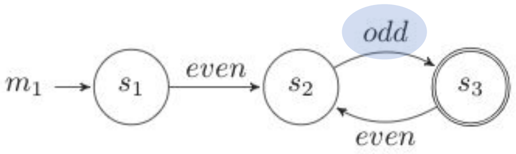


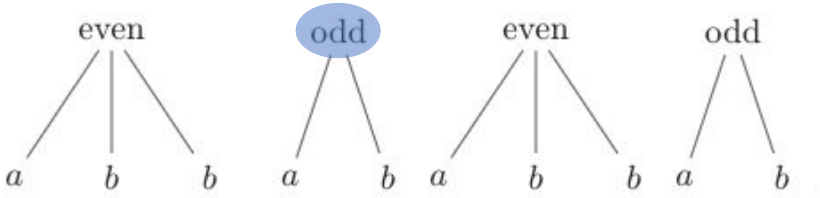
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

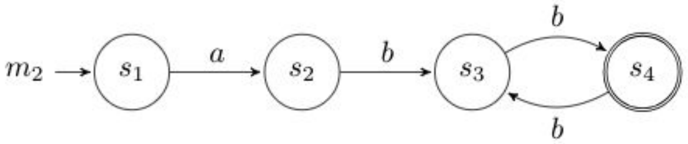
Hierarchical-Alphabet Automaton

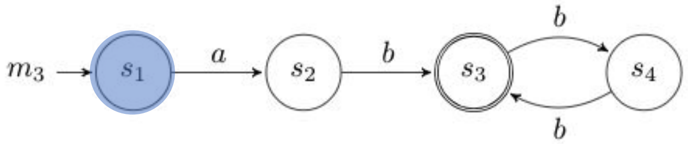
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





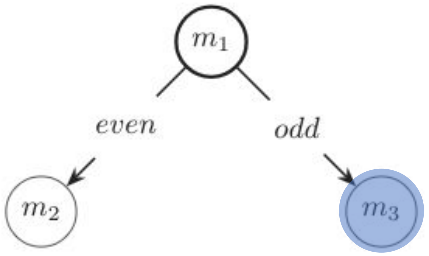


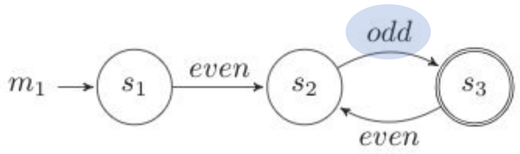


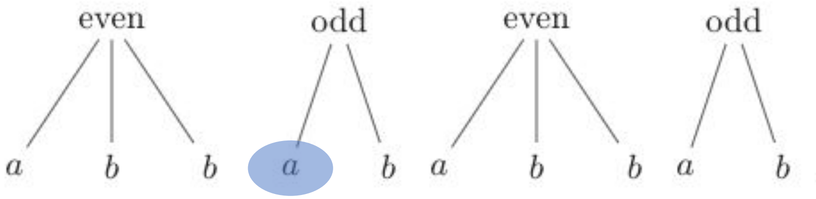


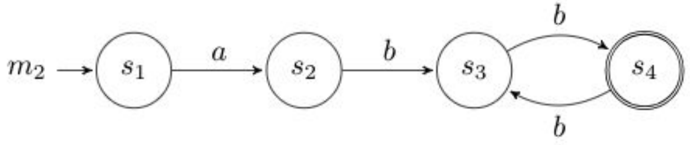
Hierarchical-Alphabet Automaton

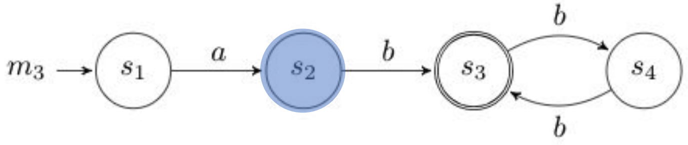
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.







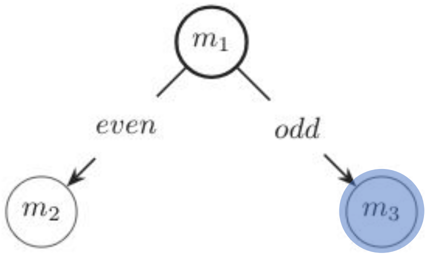


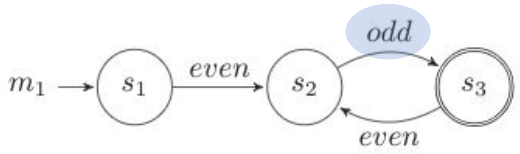


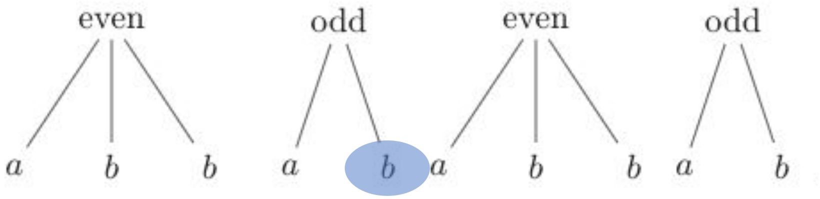
ARTIFICIAL INTELLIGENCE RESEARCH GROUP

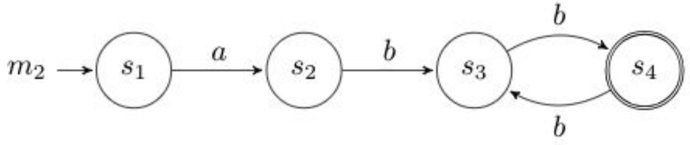
Hierarchical-Alphabet Automaton

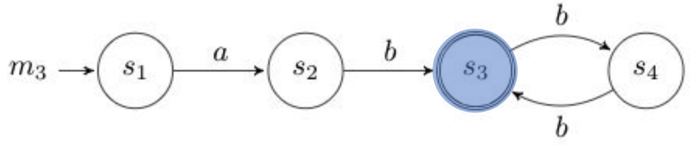
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





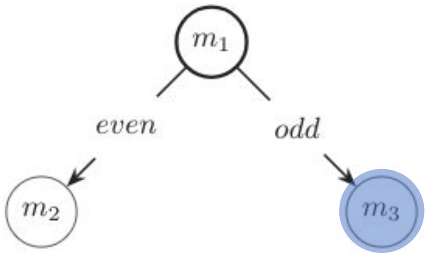


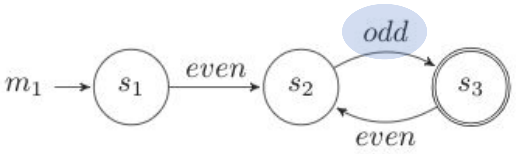


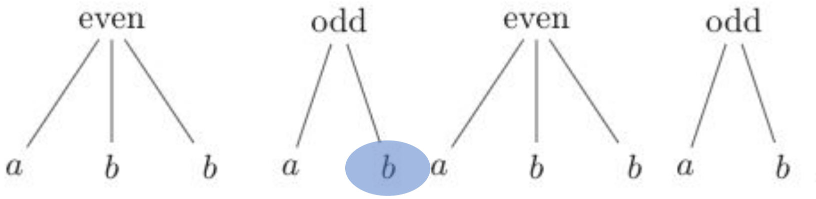


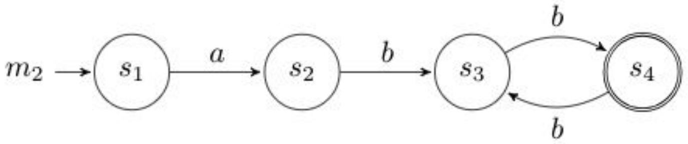
Hierarchical-Alphabet Automaton

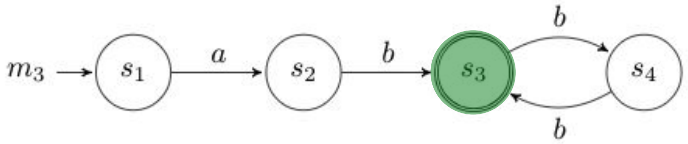
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.





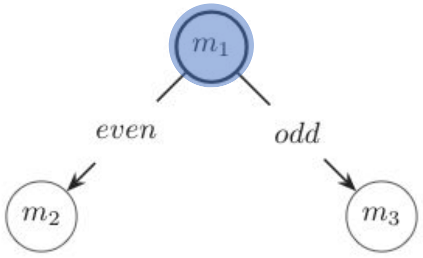


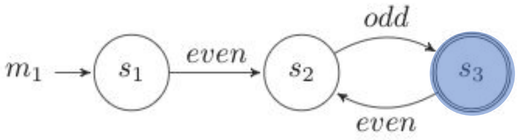


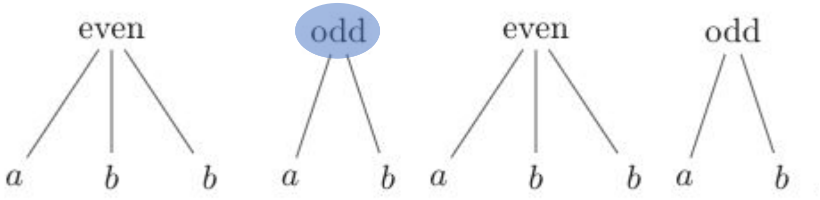


Hierarchical-Alphabet Automaton

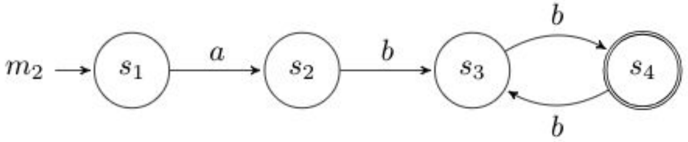
The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

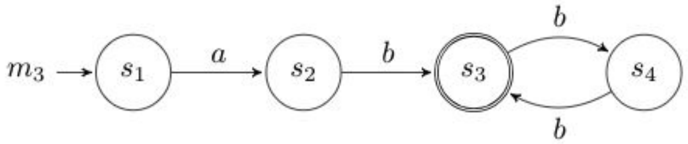






and so on...

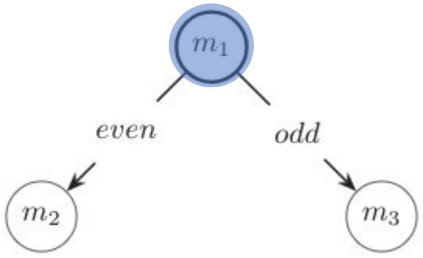


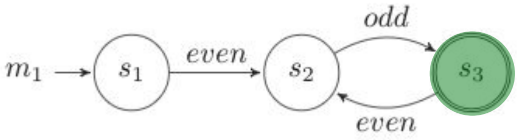


ARTIFICIAL INTELLIGENCE RESEARCH GROUP

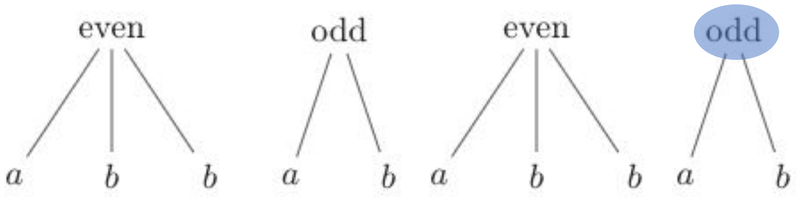
Hierarchical-Alphabet Automaton

The hierarchical-alphabet automaton is a finite poset $P = (M, \preceq)$ of machines $m_i \in M$, with one machine m_r as the greatest element of the set. We can interpret this ordering as a directed acyclic graph, with machine m_r as the only source of the DAG.

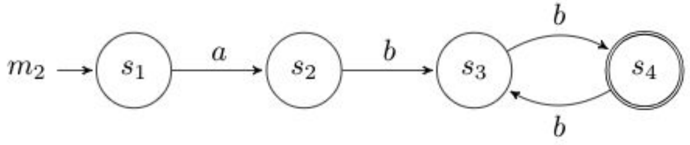


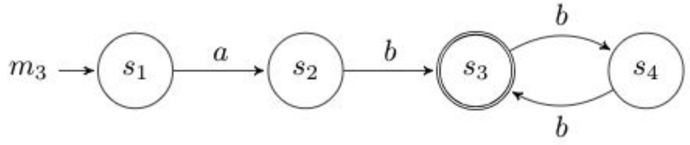


In the end:



✅

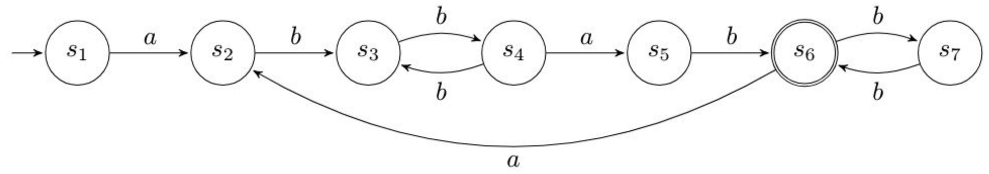


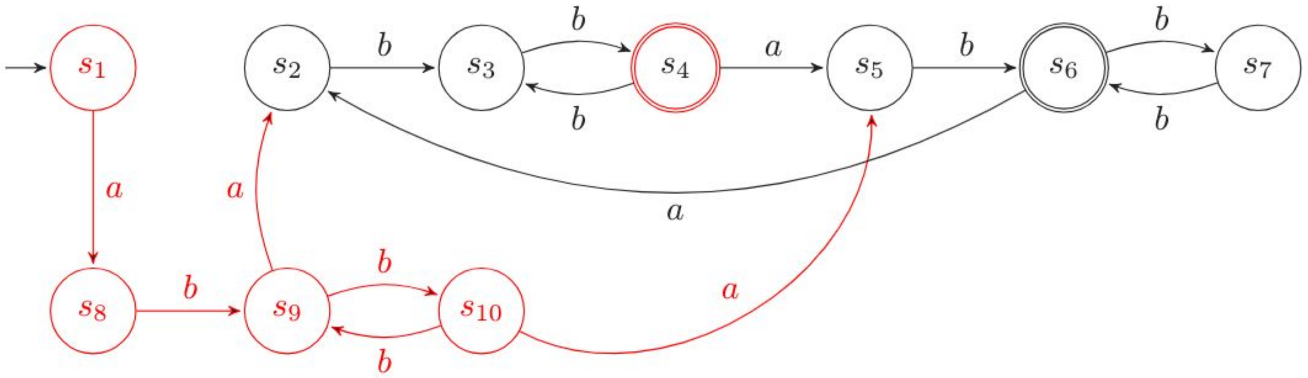


Advantages

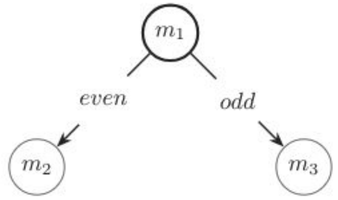
- **What about evolving behaviour?**
  - Remember:

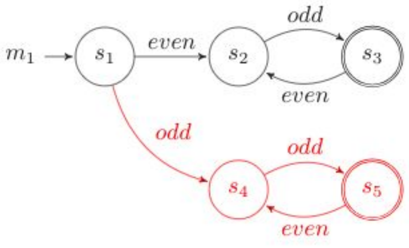
Regular FSM

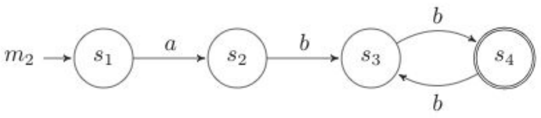


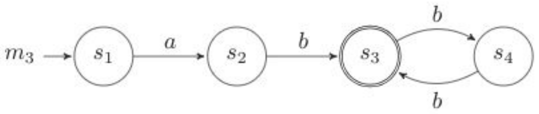


Hierarchical-Alphabet Automaton









ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Advantages and Disadvantages

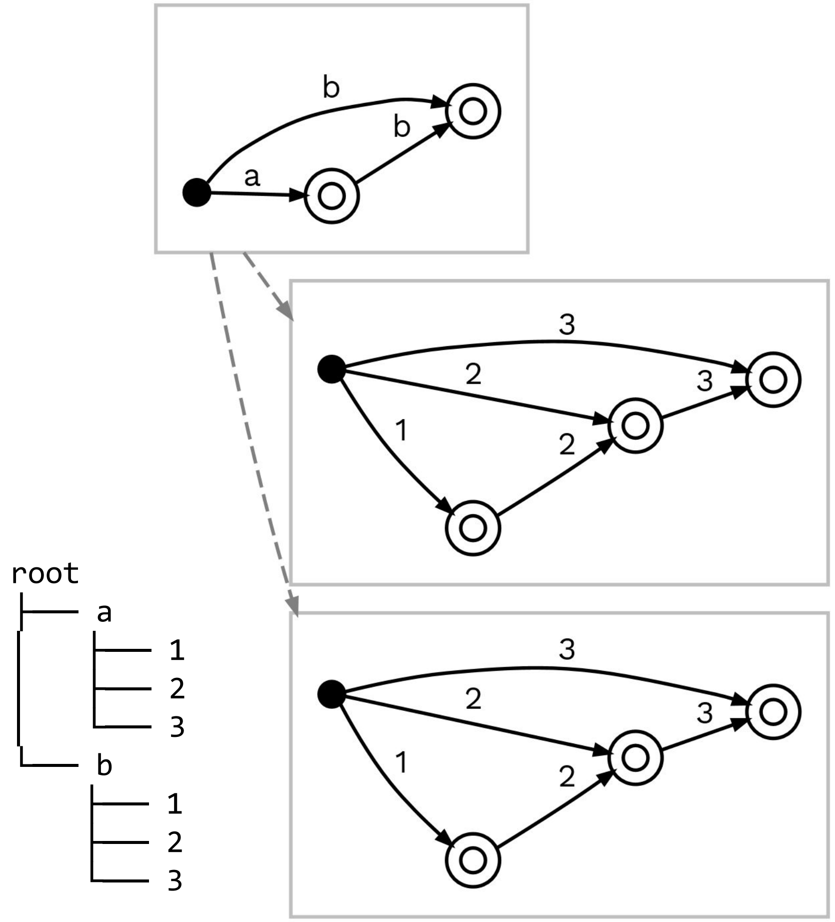
- **Modularity, interpretability, ...** ✅
  - When *behaviour* changes, we can easily build on existing HAA without having to change (all) submachines + introduce new submachines on top of them (= modularity)

- **Hierarchical inference?** ❌
  - Here, we assume we have the hierarchy already
  - How do we **infer** the hierarchy from data alone?

Building HAA - *hierarchical factor oracle*

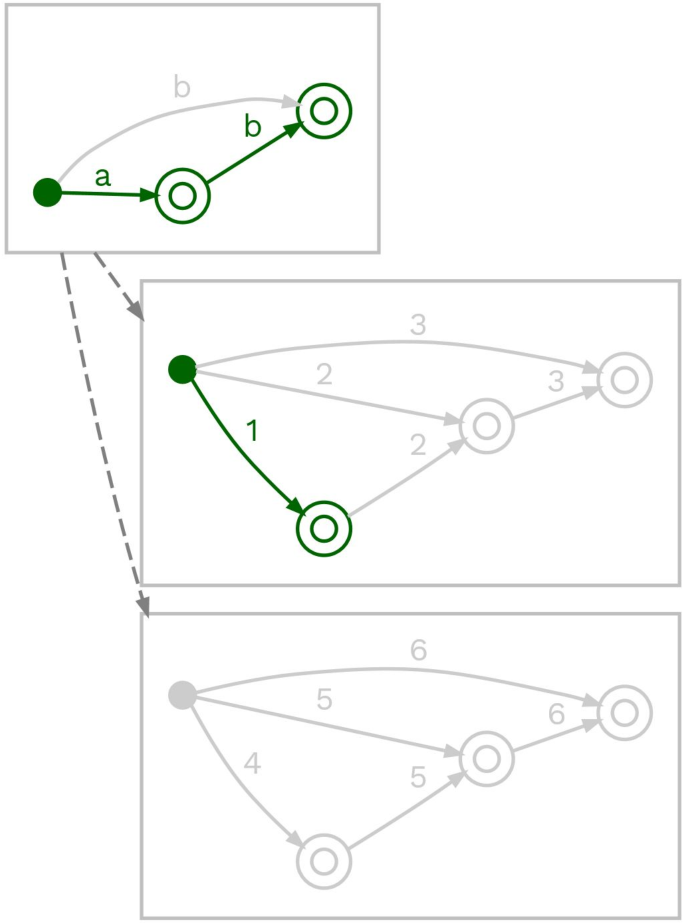
- **Intuitively, the HFO is the hierarchical variant of the *factor oracle***

- **Accepts language of hierarchical factors:**
  - Accepts *at least* the set of ordered subtrees which we used to create the HFO



Building HAA - *hierarchical factor oracle*

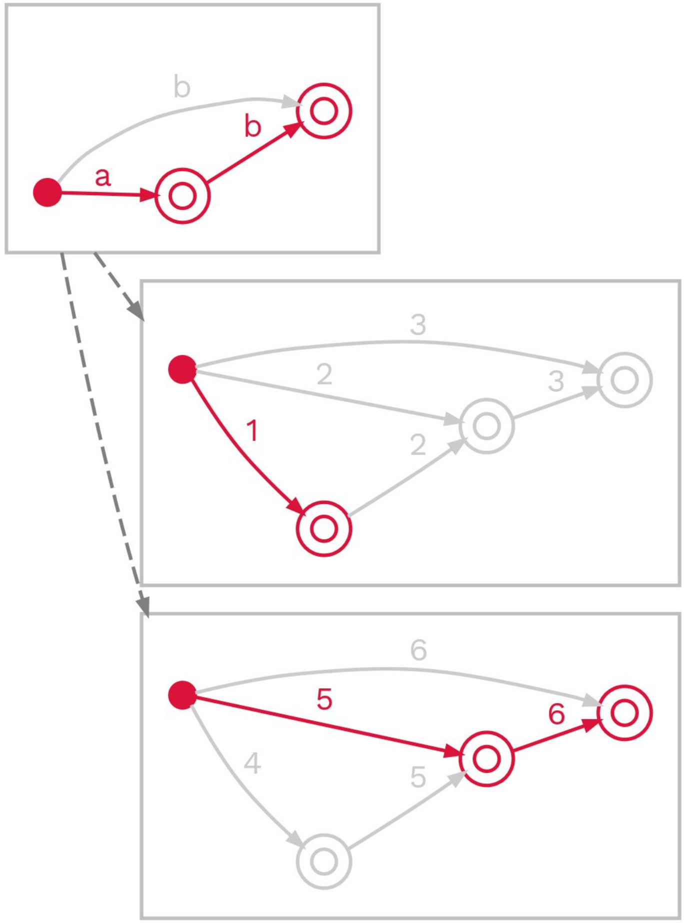
```
root
├── a
│   └── 1
└── b
```



ARTIFICIAL
INTELLIGENCE
RESEARCH GROUP

Building HAA – *hierarchical factor oracle*

```
root
├── a
│   └── 1
└── b
    ├── 5
    ├── 6
    └── 7
```



Applications of HAA

- **Anomaly detection:**
  - Finding patterns in data that do not conform to expected behaviour

- **Main focus:**
  - Needs to be **lightweight** due to constraints in resources
  - We should **never forget rarely-occurring events**
  - We should be able to **adapt to new behavior**
  - It **should be general enough** to work on different time series
  - It should **recognize non-strictly periodic events**
  - Should have a **minimum amount of false positives**

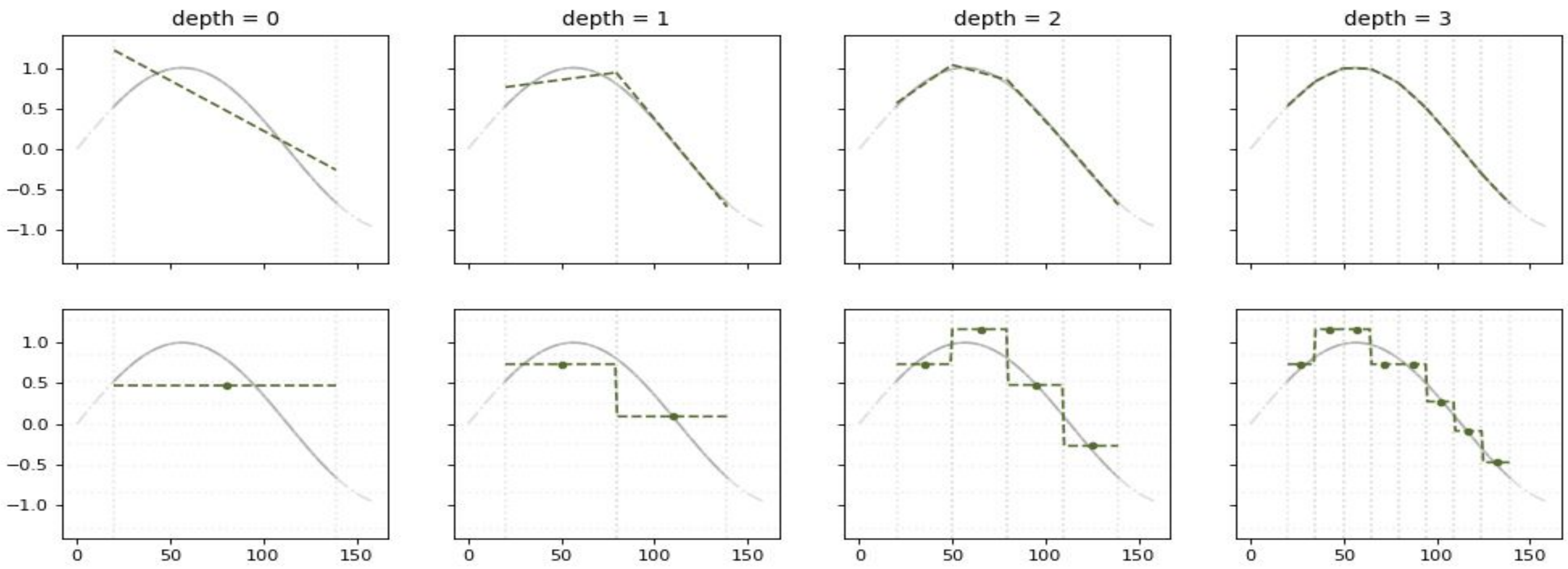
Van Onsem, M., De Paepe, D., Hautte, S. V., Bonte, P., Ledoux, V., Lejon, A., ... & Van Hoecke, S. (2022). Hierarchical pattern matching for anomaly detection in time series. Computer Communications, 193, 75-81.

Applications of HAA

- **How do we use HAA for time series?**
  - Extract overlapping subsequences using a sliding window
  - Discretise segments, and use these as symbols of alphabet

- **How do we have hierarchy in time series?**
  - Discretise in increasing granularity

Applications of HAA



ARTIFICIAL INTELLIGENCE RESEARCH GROUP

Applications of HAA

- **Some projects of the AI lab's *Applied Research* team:**
  - "For **cybersecurity**, our primary application involves anomaly detection using both discrete and continuous data. For instance, endpoints can generate discrete data like process creation and termination events, or continuous data such as CPU usage or domain requests over time. Using this data, we model the processes or systems that generate this data, and apply these models for anomaly detection."

Applications of HAA

- **Some projects of the AI lab's *Applied Research* team:**
  - "For **predictive building maintenance**, the goal is to move away from employing corrective or periodic maintenance approaches. We aim to predict potential system failures by monitoring and modelling the system's functionality over time, facilitating corrective action before an actual failure that impacts building user comfort or energy-efficiency occurs. This is particularly challenging due to complex interactions among various components in large building installations."

Applications of HAA

- **Some projects of the AI lab's *Applied Research* team:**
    - "For **neuroscience**, collected data can help gain insights into the spatio-temporal dynamics between brain regions. High-resolution techniques such as magnetoencephalography (MEG) have allowed for the study of brain networks. Therefore, we aim to develop new techniques for context-aware modelling and long-term dependency modelling of these networks, taking into account factors such as subject fatigue and age regression. We apply these techniques to improve our understanding of the impact of Multiple Sclerosis on brain functioning."