Applications of finite state machines

Aleksey Cheusov
vle@gmx.net

Minsk, Belarus, 2019

What is this presentation about?

- Finite State Automata (FSA) and Weighted Finite State Automata (WFSA)
- Regular language and Regular expressions libraries
- Deterministic (DFA) and Non-deterministic Finite State Automata (NFA)
- Moore Machines and Mealy Machines
- Finite State Transducers (FST) and Weighted Finite State Transducers (WFST)
- Algorithm of converting NFA to DFA
- DFA minimization algorithm
- Applications of all of the above

What is NOT this presentation about?

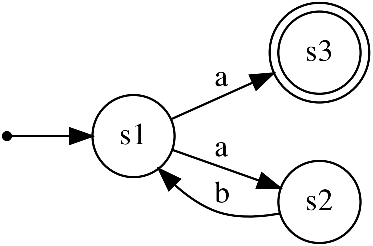
- Chomsky grammar hierarchy
- Context Free Grammars
- Context Sensitive Grammars
- Turing Machines
- Nested stack automata

Definition of FSA (non-deterministic FSA also known as NFA)

A finite state automaton is a 5-tuple $< I, S, Q, F, \delta >$. Sometimes it is called "acceptor".

- I is the input alphabet, a finite non-empty set of symbols.
- S is a finite, non-empty set of states.
- Q is the set of start states, $Q \subseteq S$.
- F is the set of final states, $F \subseteq S$.
- δ is the transition relation: $\delta \subseteq S \times I \times S$ (or, alternatively, $\delta : S \times I \rightarrow 2^S$)

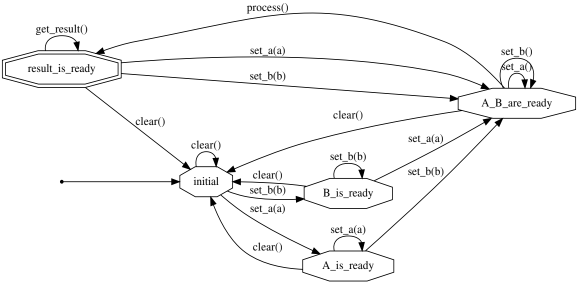
Example: $< \{a, b\}, \{s1, s2, s3\}, \{s1\}, \{s3\}, \delta >$



FSA for software design and testing

Summator:

```
template <typename T> class isummator {
    virtual void set_a(T a) = 0;
    virtual void set_b(T b) = 0;
    virtual void process() = 0;
    virtual T get_result() = 0;
    virtual void clear() = 0;
}
```



FSA for software design and testing

	set_a	set_b	process
initial	A_is_ready	B_is_ready	Error!
A_is_ready	A_is_ready	A_B_are_ready	Error!
B_is_ready	A_B_are_ready	B_is_ready	Error!
A_B_are_ready	A_B_are_ready	A_B_are_ready	result_is_ready
result_is_ready	A_B_are_ready	A_B_are_ready	Error!

Table: Transition table, part 1

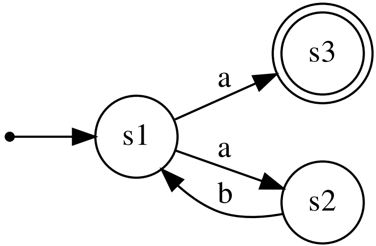
	get_result	clear
initial	Error!	initial
A_is_ready	Error!	initial
B_is_ready	Error!	initial
A_B_are_ready	Error!	initial
result_is_ready	result_is_ready	initial

Table: Transition table, part 2

IMHO, this kind of FSA for objects' states is a part of Contract Programming paradigm implemented in Eiffel programming language.

Language of FSA

The language formed by FSA consists of all distinct strings that can be accepted by FSA, i.e. sequences of input symbols that start in a start state and ends in a final state. $L(fsa) = \{(ab)^n a | n \geq 0\}$



Regular language

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- The empty language $\emptyset$ and the empty string language $\{\epsilon\}$ are regular languages.
- For each $a \in \Sigma$, the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \bullet B$ (concatenation), and $A*$ (Kleene star) are regular languages.
- No other languages over Σ are regular.

A formalism described above gives us so called "regular expressions".

Theorem: Regular Language and Finite State Automaton are equivalent formalisms. That is, for each regular language the equivalent FSA exists and vise versa.
Theorem: Regular languages are closed under concatenation, union, kleene star, intersection and complementation.

Deterministic FSA (also known as DFA)

A deterministic finite state automaton is a 5-tuple $< I, S, q, F, \delta >$.

- I is the input alphabet, a finite non-empty set of symbols.
- S is a finite, non-empty set of states.
- q is the start state, $q \in S$.
- F is the set of final states, $F \subseteq S$.
- δ is the state-transition function: $\delta : S \times I \to S$

Theorem: NFA and DFA are equivalent formalisms.
Theorem: DFA can be exponentially larger than equivalent NFA.
Theorem: There is only one minimal (with the minimal number of states) DFA.

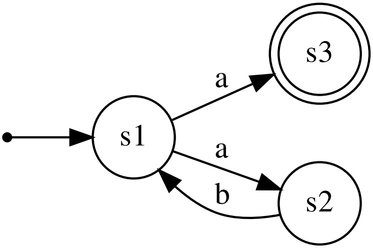
NFA to DFA conversion algorithm

Algorithm 1: nfa2dfa algorithm AKA "Subset construction"

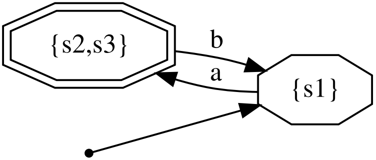
input : NFA $= < I, S, Q, F, \delta >$
output: DFA $= < I, S', q', F', \delta' >$
$\delta' := \emptyset, q' := \{s | s \in Q\}, S' := \{q'\}$
$seen := \{q'\}, queue := [q']$
while $queue \neq \emptyset$ **do**
    $src\_states \leftarrow queue$
    for $i \in I$ **do**
        $trg\_states := \{s^{trg} | (s^{src}, i, s^{trg}) \in \delta, s^{src} \in src\_states\}$
        if $trg\_states \neq \emptyset$ **then**
            $\delta' \leftarrow (src\_states, i, trg\_states)$
            $S' \leftarrow trg\_states$
            if $trg\_states \notin seen$ **then**
                $queue \leftarrow trg\_states$
                $seen \leftarrow trg\_states$
$F' := \{state\_set \in S' | \exists s \in state\_set, s \in F\}$

Example of NFA and equivalent DFA

Simple NFA

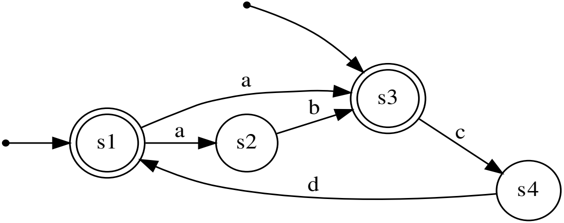


Equivalent DFA

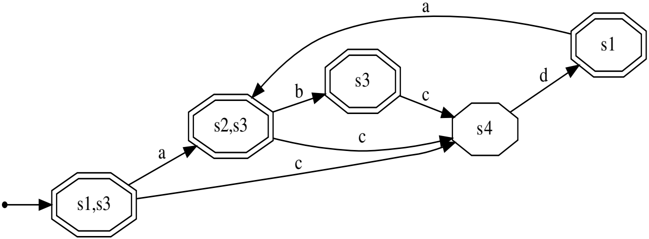


Example of NFA and equivalent DFA

NFA



Equivalent DFA



DFA Minimization, Brzozowski algorithm

DFA minimization is the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA that has a minimum number of states.

Let's define two operators:

- R – revert operator. $L(R(fsa)) = \{inverse(w) | w \in L(fsa)\}$
- D – nfa to dfa conversion.

Algorithm: $MinDFA = (D \circ R \circ D \circ R)NFA$
Note: Unlike others Brzozowski algorithm builds MinDFA for NFA!

Other algorithms are described in "A Taxonomy of Finite Automata Minimization Algorithms", Bruce Watson, 1993

Algorithm of match with a help of DFA

Algorithm 2: Match with a help of DFA. Complexity: $O(n)$

input : DFA $= < I, S, q, F, \delta >$, $Text = [t_1, t_2 \dots t_n], t_i \in I$
output: *true* **or** *false*

```
state := q
for i from 1 to n do
    if δ is defined on (state, t_i) then
        state := δ(state, t_i)
    else
        return false
    end
end
return (state ∈ F)
```

Algorithm of match with a help of NFA

Algorithm 3: Match with a help of NFA. Complexity: $O(n*|S|)$

input : NFA $= < I, S, Q, F, \delta >$, $Text = [t_1, t_2 \dots t_n], t_i \in I$

output: *true* **or** *false*

$states := Q$

for i *from* 1 *to* n **and** $states \neq \emptyset$ **do**

  $states := \{trg|(src, t_i, trg) \in \delta, src \in states\}$

end

return $(\exists s \in states, s \in F)$

Search and submatch operations

Questions:

- algorithm of search: left-most longest (awk, grep, sed) or...
- algorithm of submatch: POSIX or...
- syntax for matching portions of regexps
- support of regexp negations or intersections

Article: "Efficient submatch addressing for regular expressions", Master's Thesis, Ville Laurikari
https://laurikari.net/ville/regex-submatch.pdf

Substring search and FSA

Well known algorithms:

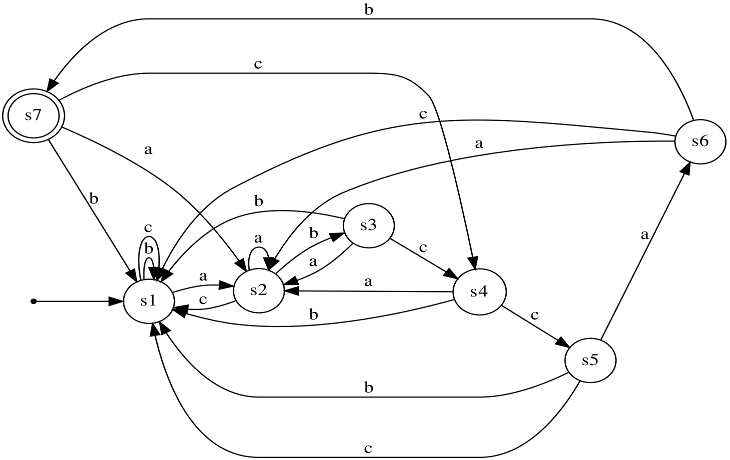
- **Knuth-Morris-Pratt** algorithm. **Task** is a search for occurrences of a fixed word W within a main text string T. **Complexity:** $(O(\mid W \mid + \mid T \mid))$. Idea is to preprocess word W and build some table-like data structure that helps us to reuse partially matched word, thus, processing each character in T only once..
- **Aho-Corasick** algorithm. **Task** is to locate elements of a finite set of strings D within an input text T. **Complexity:** $O(\mid D \mid + \mid T \mid)$.
- **Note:** These algorithm are actually equivalent to matching with a help of DFA built from search pattern(s).

Example: DFA for substring search

Pattern: "abccab"

Regexp for substring search: ".*abccab"

DFA:



Alphabet – ASCII or Unicode symbols

High-performance regexp engines:

- Ken Thompson's first implementation (1968)
- GNU libc regcomp(3)/regexec(3), GNU grep
- Google re2 library (re2j – Java reimplementation), Yandex PIRE library
- nawk (by Brian Kernighan), libtre (Finland student :-)), libuxre (Solaris OS), NetBSD libc regcomp(3)/regexec(3)

Regexp engines that suck a lot are below. They do not use FSA at all due to support of backreferences! So, they have exponential complexity of match and search.

- Perl5,6 regexps
- Java SDK regexps
- PCRE and huge amount of software based on PCRE
- Python, Ruby, PHP...
- librxspencer (by Henry Spenser, author of a crappy book...)

Very interesting article: *https://swtch.com/ rsc/regexp/regexp1.html*, also have a look at *regexp{2,3,4}.html*.

Alphabet – part-of-speech tags, e.g. PENN tagset

PENN tags are: DT, JJ, NN, NPS, VBZ . . .
POS-tagged sentence: Using/VBG italics/NNS ,/, bold/JJ or/CC underlined/JJ words/NNS can/MD change/VBP the/DT perception/NN of/IN the/DT reader/NN ./.
Regexp for matching noun phrase:
(DT? ((JJ ,)? JJ CC JJ)? ((NN | NNS)+ CD? | NP | NPS)
Extracted noun phrases:

- italics
- bold or underlined words
- the perception
- the reader

Alphabet – set of sets of words

Task: match of american or UK addresses
Regexp:[Building] PoBox City State PostCode [Phone] [Country]
where

- **Building:** <Number><Token><BuildingType>
- **Number:** '\d+(-\d+)*[a-zA-Z]*'
- **BuildingType:** Plaza *or* Tower *or* ...
- **Pobox:** PO Box *or* P.O. Box *or* P.O. BOX *or ... followed by* Number
- **City:** New York *or* Boston *Washington or* ...
- **State:** Kentucky *or* Nevada *or* ...
- **Country:** USA *or* US *or* United States of America *or* ...
- **PostCode:** '\d{5}'
- ...

Alphabet – set of words specified exlicitly or by regular expression

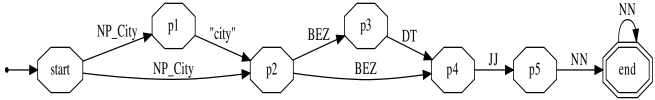
Algorithm of NFA construction – same as for alphabet with symbols but word ids are used as input weights

Algorithm of DFA construction – there is some problems with DFA. It may happen if some words are a part of regexps language, e.g. **Number** or **Token**. Solution exists! ;-)

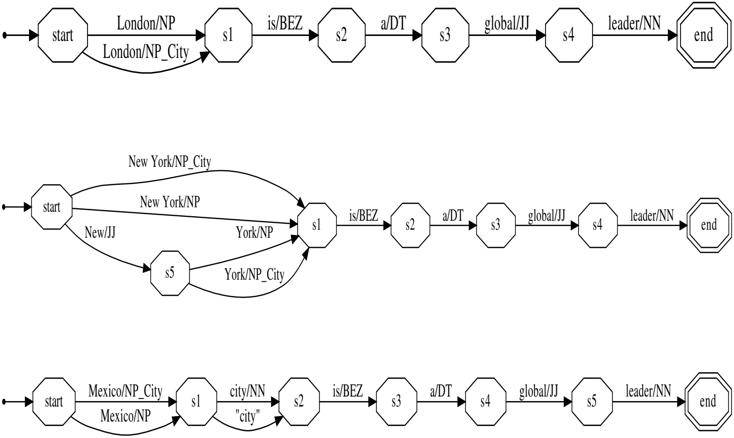
Problems: Some real words can be a part of more than one token type, e.g., **Token** or **Country**. Or we may want to treat the sequence of words as a single token, e.g., New York, or Great Britain.

Example: Complex word-based regexp

Input alphabet: $\{BEZ, JJ, DT, NN, NP, NP\_City, "city"\}$
Regexp: NN_City "city" ? BEZ DT ? JJ NN +
FSA equial to Regexp:

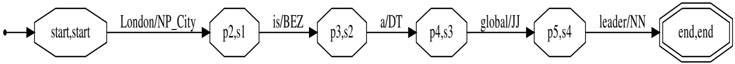


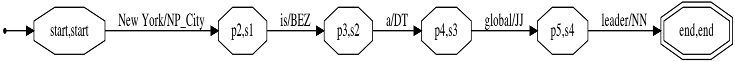
Example: Input for complex word-based regexp

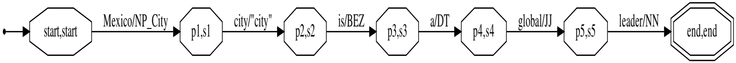


Alphabet – set of sets of words, regexps and different kinds of predicates

Match algorithm: intersection of two finite state automata (regular languages) with a help of **modified** nfa2dfa algorithm.

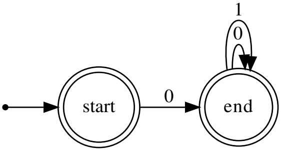




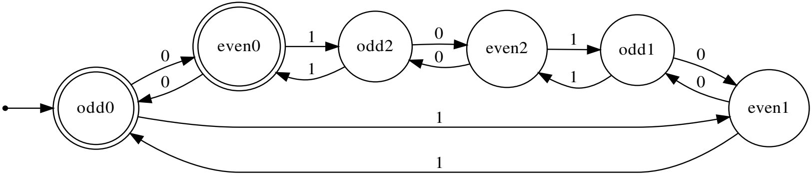


FSAs for “divisible by 2” and “divisible by 3”

Divisible by 2

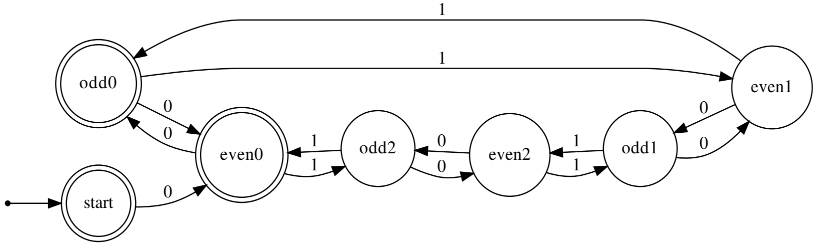


Divisible by 3



FSA for "divisible by 6"

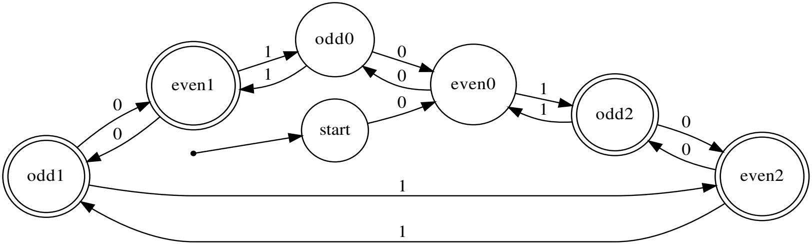
$L(div6) = L(div2) \cap L(div3)$



For building an intersection of two automata we can use nfa2dfa procedure. As a result we obtain DFA.

FSA for “divisible by 2 but not by 3”

$$L(div6) = L(div2) \setminus L(div3)$$



For building a subtraction of two automata we can also use nfa2dfa procedure. As a result we obtain DFA.

Moore and Mealy machines

Definition: A Moore machine is a 6-tuple $< I, O, S, q, \delta, \lambda >$.

- I is the input alphabet, a finite non-empty set of input symbols.
- O is the output alphabet, a finite non-empty set of output symbols.
- S is a finite, non-empty set of states.
- q is the start state, $q \in S$.
- δ is the state-transition function: $\delta : S \times I \to S$
- λ is the output function: $\lambda : S \to O$

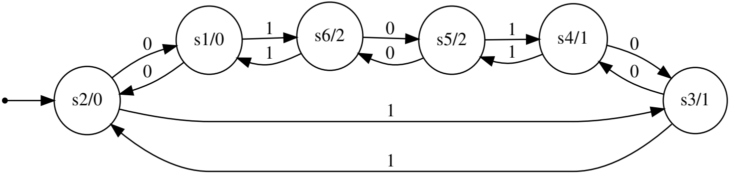
Definition: A Mealy machine is a 6-tuple $< I, O, S, q, \delta, \lambda >$.

- I is the input alphabet, a finite non-empty set of input symbols.
- O is the output alphabet, a finite non-empty set of output symbols.
- S is a finite, non-empty set of states.
- q is the start state, $q \in S$.
- δ is the state-transition function: $\delta : S \times I \to S$
- λ is the output function: $\lambda : S \times I \to O$

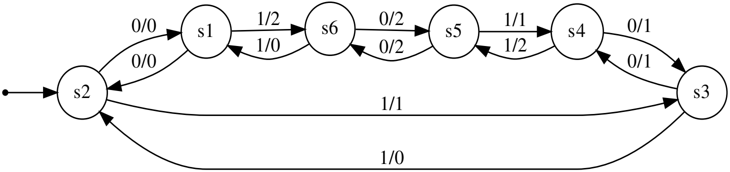
Note: In practice we often work with *partially defined* DFA, Moore and Mealy machines, that is, automata with partially defined transition function.

Moore and Mealy machines

Definition: Language of Moore/Mealy machine is $L(m) = \{(s_i, s_o) \mid$ a path from start state q produces $s_o \in O^*$ for $s_i \in I^*$ input$)\}$.
Note: Moore and Mealy machines are equivalent formalisms.
Example:



Example:



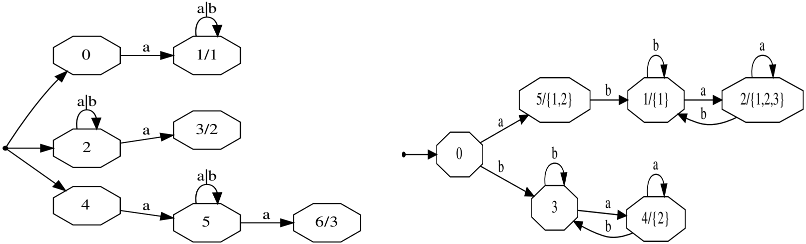
Applications of Moore and Mealy machines. Match multiple regexps

Task: We have a number of regexps and want to know, which one (potentially more than one!) match the specified text.
Solution1: 1) Mark finite state of *regexp*1 with 1, *regexp*2 with 2 etc. Also mark all other states with *empty* output symbol. 2) At the end of match operation, analyse output weight of states the match operation ends in.
Solution2: Perform *nfa2dfa* operation and assign the set of finite states that correspond to original regexps to the output weight of Moore machine. For example output alphabet for Moore machine that matches three regexps may be $\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Applications of Moore and Mealy machines. Match multiple regexps

Regexps ($I = \{a, b\}$)

- $a(a \mid b)*$
- $(a \mid b) * a$
- $a(a \mid b) * a$



(a) Non-deterministic FSM

(b) Deterministic FSM

Figure: Matching three regexps with a help of Moore machines

Weighted finite state machine and applications

Definition: Weighted finite state automaton is a 6-tuple $< I, S, Q, F, \delta, \omega >$.

- I is the input alphabet, a finite non-empty set of symbols.
- S is a finite, non-empty set of states.
- Q is the set of start states, $Q \subseteq S$.
- F is the set of final states, $F \subseteq S$.
- δ is the transitions relation: $\delta \subseteq S \times I \times S$
- $\omega : \delta \to \mathbb{R}$

ω may be distances, probabilities, penalties etc., even not limited to $\mathbb{R}$.

Weighted finite state machine and applications

Information extraction (IE) is the task of automatic extraction of structured information from unstructured and/or semi-structured machine-readable documents.

Named Entity Recognition (NER) is an information extraction technique to identify and classify named entities in text.

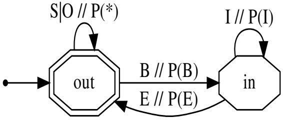
Example: Alex/S-PER is/O going/O with/O Marty/B-PER A./I-PER Rick/E-PER to/O Los/B-LOC Angeles/E-LOC

So called BIOES notation is used for mark up. There are also BIO and IO notations.

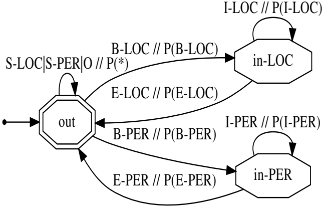
Solution: Hidden Markov Models (HMM), Maximum Entropy Markov Model (MEMM), Conditional Random Fields (CRF), Bi-directional LSTMs and other techniques are used.

Alternative solution: next slide :-)

Weighted finite state automata for BIOES notation



(a) WFSA for single-label extraction



(b) WFSA for two-label extraction

Figure: Weighted finite state automata for BIOES notations

Approach: Independent classification of each token, then extraction of correct sequences using FSA shown above. **Solution 1:** Maximum joint probability of B_i, I_i, O_i, E_i, S_i, i.e., product of P(*) along the path. **Solution 2:** Minimum sum of penalties along the path, i.e., sum of subtraction of selected probability and maximum probability for each token. **Note:** Best path can easily be found with a help Viterbi algorithm. **Note:** Advantage of this approach is that we can easily set a threshold for entity extraction, for example, product of B, I and E labels. Thus, we can balance between Precision and Recall.

Weighted finite state automata for BIOES notation

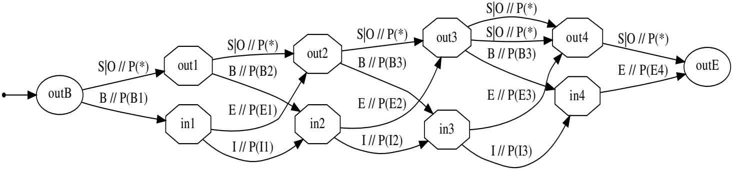


Figure: BIOES WFSA for single-label 5-word input

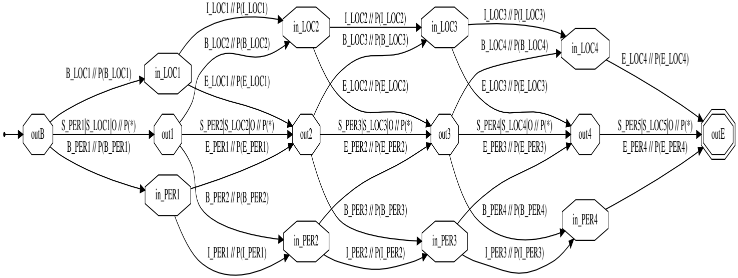


Figure: BIOES WFSA for two-label 5-word input

Finite state transducer

Definition: A finite state transducer is a 6-tuple $< I, O, S, Q, F, \delta >$.

- I is the input alphabet, a finite non-empty set of symbols.
- O is the output alphabet, a finite non-empty set of symbols.
- S is a finite, non-empty set of states.
- Q is the set of start states, $Q \subseteq S$.
- F is the set of final states, $F \subseteq S$.
- δ is the transition relation: $\delta \subseteq S \times (I \cup \{\epsilon\}) \times (O \cup \{\epsilon\}) \times S$ where ϵ is the empty string.

Note: Weighted FST is defined the same way as WFSA.
Applications of WFST: speech recognition, speech synthesis, optical character recognition, machine translation, a variety of other natural language processing tasks including parsing and language modeling, image processing and computational biology.
WFST Guru: Mehryar Mohri

OCR CUSIP Correction

CUSIP is a nine-character alphanumeric code that identifies a North American financial security for the purposes of facilitating clearing and settlement of trades.
Task: CUSIP is extracted from PDF and TIFF documents which are OCRed first. The problem is OCR leads to huge amount of errors. Obviously, quality of extraction of busines information such as amount of money, currencies, CUSIPs, IBANs, BICs etc. is extreamly important. So, our goal is to correct incorrectly OCRed CUSIPs.
Dataset: A list of pairs $(extractedCUSIP, correctCUSIP)$.
CUSIP is described here: *https://en.wikipedia.org/wiki/CUSIP*
Note: CUSIP has a check sum.

OCR CUSIP Correction. Dataset

extracted	correct	comment
42884VAN1	42884VAN1	everything is correct
42884VAN1	42804VAM3	N $\rightarrow$ M
D0100UAE2	00100UAE2	D $\rightarrow$ 0
09179FAS1	09179FAS1	everything is correct
D9I79FASi	09179FAS1	i $\rightarrow$ 1, I $\rightarrow$ 1, D $\rightarrow$ 0,
256684BD8	256604BD0	8 $\rightarrow$ 0
42884VAM3	42804VAM3	8 $\rightarrow$ 0
8485OXAB8	84850XAB8	O $\rightarrow$ 0
...	...	

Table: Dataset for OCR CUSIP correction

CUSIP check sum

Algorithm 4: Calculate 9th CUSIP character (check sum)

input : CUSIP characters $cusip[i], 1 \leq i \leq 8$
output: 9th CUSIP character which is a check sum
$sum := 0$
for $1 \leq i \leq 8$ **do**
  $c := cusip[i]$
  if $c \in \{"0", "1" \ldots "9"\}$ **then**
    $v :=$ numeric value of the digit c
  else if $c \in \{"A", "B" \ldots "Z"\}$ **then**
    $p :=$ ordinal position of c in the alphabet $(A = 1, B = 2...)$
    $v := p + 9$
  else if $c = " * "$ **then** $v := 36$
  else if $c = "@"$ **then** $v := 37$
  else if $c = "\#"$ **then** $v := 38$

  if i *is even* **then** $v := v * 2$

  $sum := sum + int(v \textbf{ div } 10) + (v \textbf{ mod } 10)$
return *(10 - (sum* **mod** *10))* **mod** *10*

CUSIP check sum (simplified version)

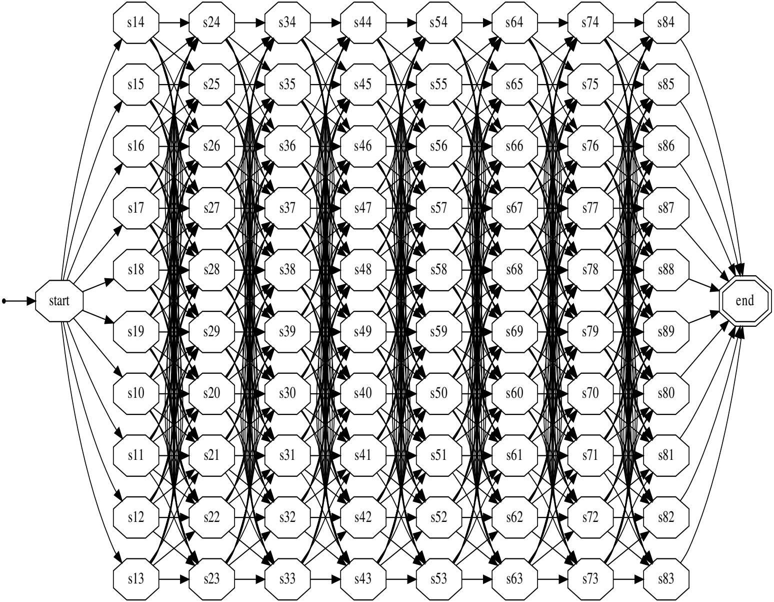
Notes about algorithm shown above:

- Expression $sum + int(v \textbf{ div } 10) + (v \textbf{ mod } 10)$ can be simplified.
- Condition "i is even" can be moved to $v :=$ assignments.
- **return** statement uses "sum **mod** 10", so we can calculate this value within the loop. So, value of sum within a loop can just be modified as $sum = sum + f(cusip[i], i) \textbf{ mod } 10$.

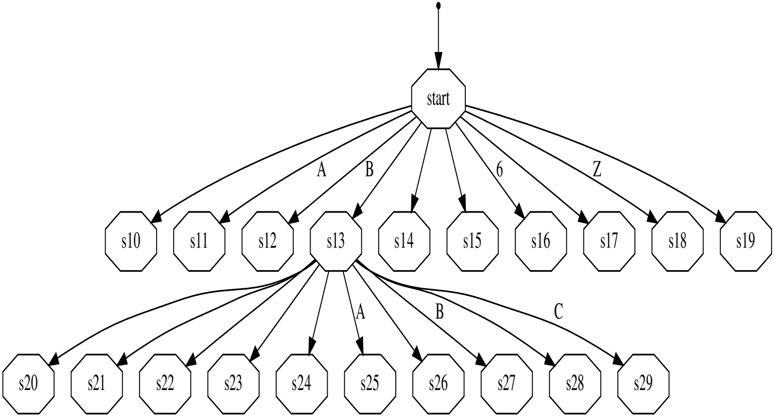
Algorithm 5: Calculate 9th CUSIP character (simplified version)

input : CUSIP characters $cusip[i], 1 \leq i \leq 8$
output: 9th CUSIP character which is a check sum
$sum := 0$
for $1 \leq i \leq 8$ **do**
  $sum := (sum + f(i, cusip[i])) \textbf{ mod } 10$
end
return $(10 - sum) \textbf{ mod } 10$

CUSIP finite state automaton



Partial transition function of CUSIP finite state automaton

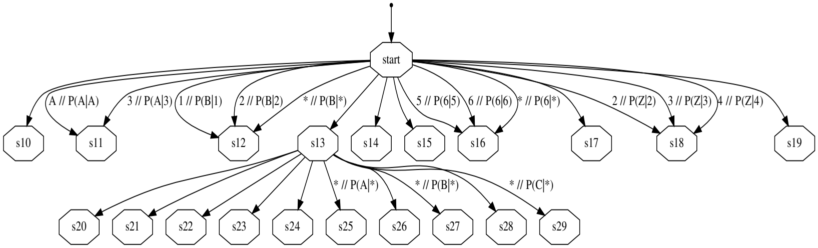


Partial transition function of CUSIP WFST

- Conditional probability of correct symbol $S_{correct}$ given S_{seen} for position i.

$$P^i(S_{correct} \mid S_{seen}) := \frac{\sum_{j=1}^{N}[S^i_{j,correct} == S_{correct}] * [S^i_{j,seen} == S_{seen}]}{\sum_{j=1}^{N}[S^i_{j,seen} == S^i_{j,seen}]}$$

where N is the number of pairs in dataset, $1 \leq i \leq 8$



OCR CUSIP Correction. Algorithm.

Algorithm: given 9 character CUSIP as input, the corrected CUSIP is the sequence of output symbols of WFST along the best path from start to finite state. Best path is the path with maximum product of conditional probabilities.
Question: Hidden Markov Model? Maximization of Joint Probability? Weighted Finite State Transducer?
Results: 99.7% accuracy on 5-fold cross-validation. Two diverse datasets of size 10^6 pairs. Input datasets correctness: 65% and 95%.
Note: Probabilities must be smoothed in order to avoid multiplying by zero. Examples: Good Turing, Add-lambda, Katz smoothing etc.

OCR IBAN Correction. Approach.

IBAN is an internationally agreed system of identifying bank accounts across national borders.
IBAN format is specified by regular expression using letters and digits. Example (Belarus): “BYkk bbbb aaaa cccc cccc cccc cccc” where “b” = national bank or branch code, “a” – balance account number, “c” – account number and “k” – check sum.
IBAN is validated by converting it into an integer and performing a basic $mod 97$ operation (“kk” portion of IBAN).

Approach: same as for CUSIP correction – WFST based on IBAN regular expression and “mod 97” check sum.

The End