Synchronizing Finite Automata

Lecture I. History and Motivation

Mikhail Volkov

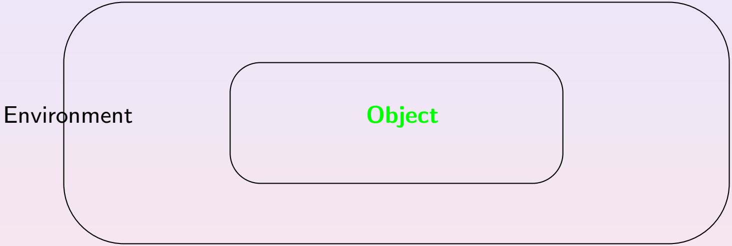
Ural Federal University

Spring of 2021

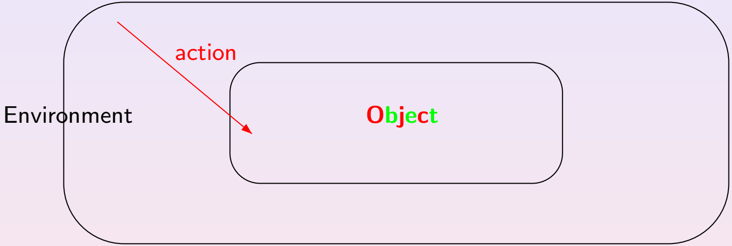
1. Finite Automata

A finite automaton is a simple but extremely productive concept that captures the very important idea of an object interacting with its environment.

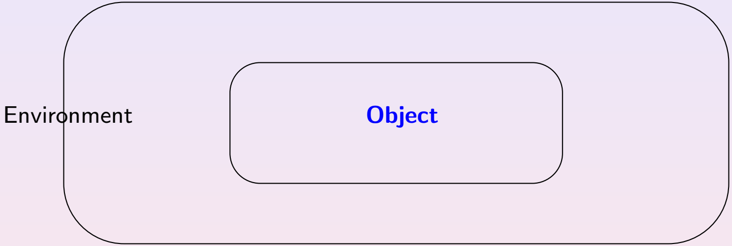
A finite automaton is a simple but extremely productive concept that captures the very important idea of an object interacting with its environment.



A finite automaton is a simple but extremely productive concept that captures the very important idea of an object interacting with its environment.



A finite automaton is a simple but extremely productive concept that captures the very important idea of an object interacting with its environment.



This notion originates in the seminal work by Alan Turing ("On Computable Numbers, With an Application to the Entscheidungsproblem", Proc. London Math. Soc., Ser. 2, 42 (1936), 230–265).

"*The behavior of the computer at any moment is determined by the symbols which he is observing, and his state of mind at that moment*".

This notion originates in the seminal work by Alan Turing ("On Computable Numbers, With an Application to the Entscheidungsproblem", Proc. London Math. Soc., Ser. 2, 42 (1936), 230–265).

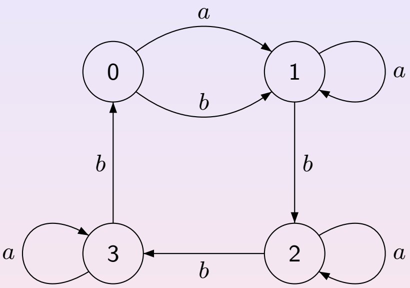
"*The behavior of the computer at any moment is determined by the symbols which he is observing, and his state of mind at that moment*".

Another important source is the work by neurobiologists Warren McCulloch and Walter Pitts ("A Logical Calculus of the Ideas Immanent in Nervous Activity", Bull. Math. Biophys. 5 (1943), 115–133).

3. Visualization

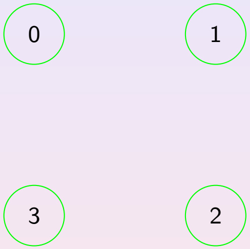
Finite automata admit a convenient visual representation.

Finite automata admit a convenient visual representation.



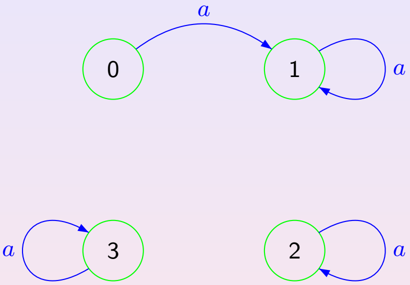
3. Visualization

Finite automata admit a convenient visual representation.



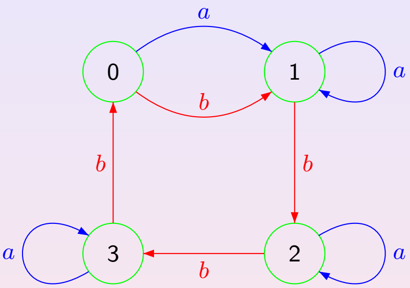
Here one sees 4 **states** called 0,1,2,3,

Finite automata admit a convenient visual representation.



Here one sees 4 **states** called 0,1,2,3, an action called a

Finite automata admit a convenient visual representation.



Here one sees 4 **states** called 0,1,2,3, an action called a and another action called b.

We consider complete deterministic finite automata (DFA):

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

We consider complete deterministic finite automata (DFA):

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta \colon Q \times \Sigma \to Q$ is the transition function.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the finite state set;
- Σ is the input finite alphabet;
- $\delta \colon Q \times \Sigma \to Q$ is the transition function.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta \colon Q \times \Sigma \to Q$ is the transition function.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta\colon Q \times \Sigma \to Q$ is the totally defined transition function.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta\colon Q \times \Sigma \to Q$ is the transition function.

We need neither initial nor final states.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta \colon Q \times \Sigma \to Q$ is the transition function.

We need neither initial nor final states.

Σ^* stands for the set of all words over Σ including the empty word. The function δ uniquely extends to a function $Q \times \Sigma^* \to Q$ still denoted by δ.

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta\colon Q \times \Sigma \to Q$ is the transition function.

We need neither initial nor final states.
Σ^* stands for the set of all words over Σ including the empty word. The function δ uniquely extends to a function $Q \times \Sigma^* \to Q$ still denoted by δ.
To simplify notation we often write $q\,.\,w$ for $\delta(q, w)$

We consider complete deterministic finite automata:

$$\mathscr{A} = \langle Q, \Sigma, \delta \rangle.$$

Here

- Q is the state set;
- Σ is the input alphabet;
- $\delta \colon Q \times \Sigma \to Q$ is the transition function.

We need neither initial nor final states.
Σ^* stands for the set of all words over Σ including the empty word. The function δ uniquely extends to a function $Q \times \Sigma^* \to Q$ still denoted by δ.
To simplify notation we often write $q\,.\,w$ for $\delta(q, w)$
and $P\,.\,w$ for $\{\delta(q, w) \mid q \in P\}$.

An automaton $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ is called synchronizing if there exists a word $w \in \Sigma^*$ whose action resets $\mathscr{A}$, that is, leaves the automaton in one particular state no matter which state in Q it started at: $\delta(q, w) = \delta(q', w)$ for all $q, q' \in Q$.

An automaton $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ is called synchronizing if there exists a word $w \in \Sigma^*$ whose action resets $\mathscr{A}$, that is, leaves the automaton in one particular state no matter which state in Q it started at: $\delta(q, w) = \delta(q', w)$ for all $q, q' \in Q$.

We can also write this as $|Q\,.\,w| = 1$.

An automaton $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ is called synchronizing if there exists a word $w \in \Sigma^*$ whose action resets $\mathscr{A}$, that is, leaves the automaton in one particular state no matter which state in Q it started at: $\delta(q, w) = \delta(q', w)$ for all $q, q' \in Q$.

We can also write this as $|Q \,.\, w| = 1$.

Any word w with this property is a reset word for $\mathscr{A}$.

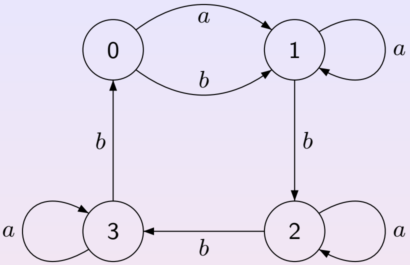
An automaton $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ is called synchronizing if there exists a word $w \in \Sigma^*$ whose action resets $\mathscr{A}$, that is, leaves the automaton in one particular state no matter which state in Q it started at: $\delta(q, w) = \delta(q', w)$ for all $q, q' \in Q$.

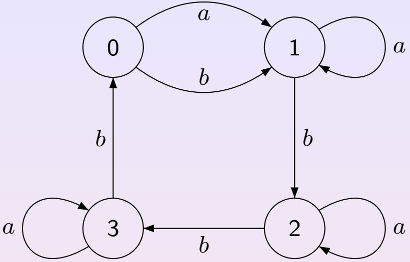
We can also write this as $|Q\,.\,w| = 1$.

Any word w with this property is a reset word for $\mathscr{A}$.

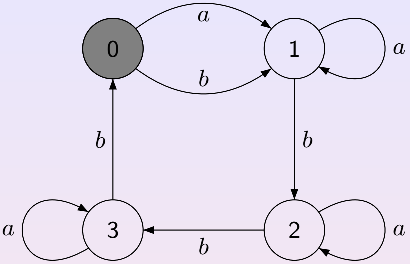
Other names:

- for automata: directable, cofinal, collapsible, etc;
- for words: directing, recurrent, synchronizing, etc.

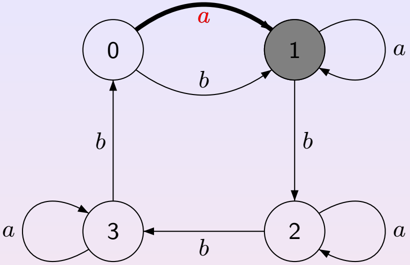

0
1
2
3
a
b
a
b
b
a
b
a



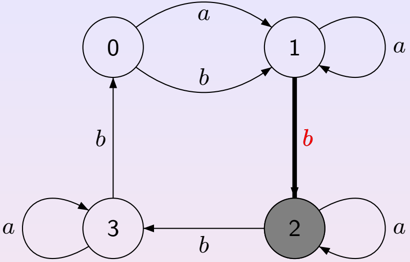
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



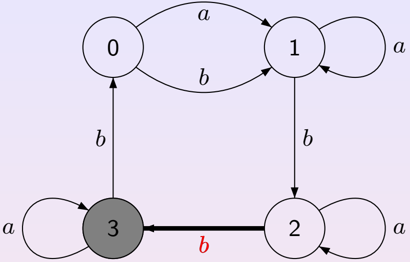
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

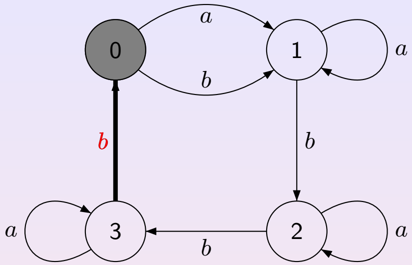


A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

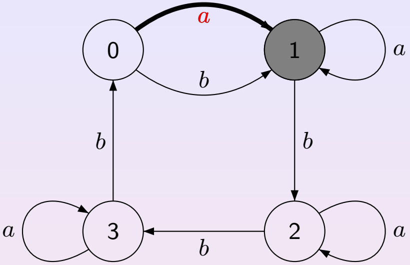


A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

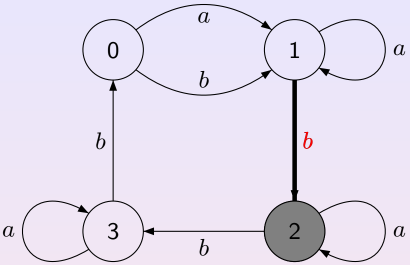
6. An Example



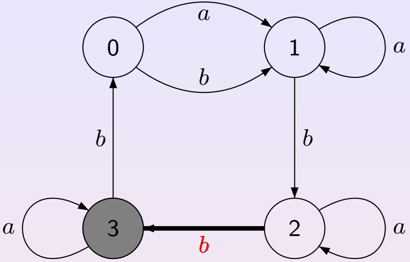
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



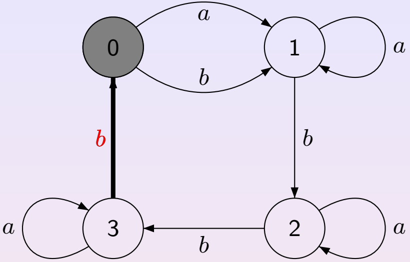
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



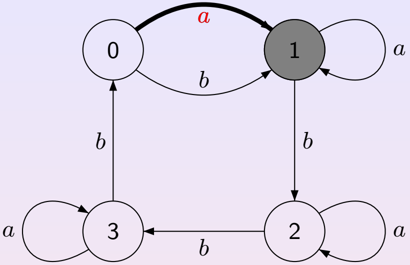
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



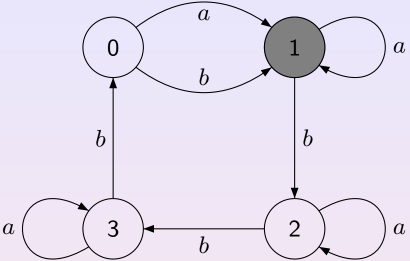
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



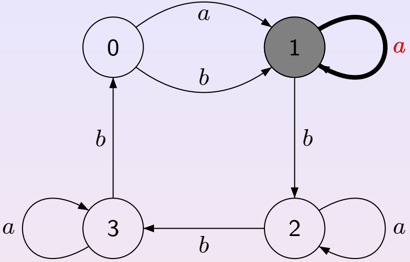
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



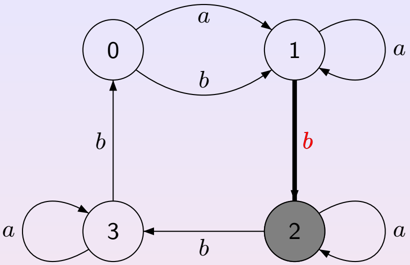
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

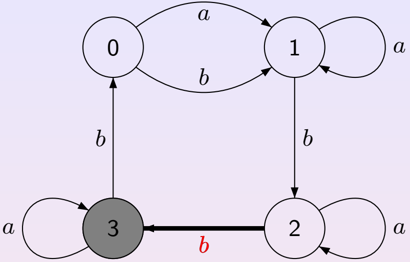


A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

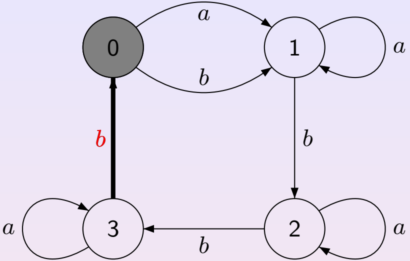


A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

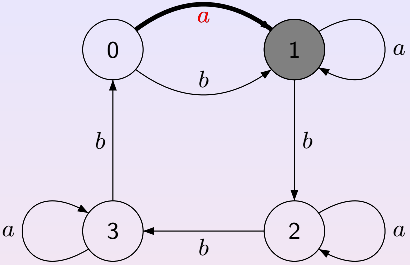
6. An Example



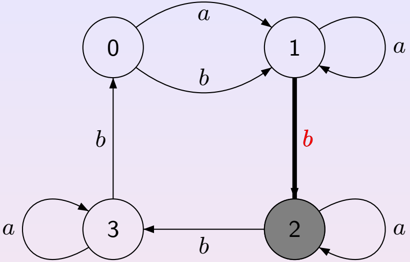
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



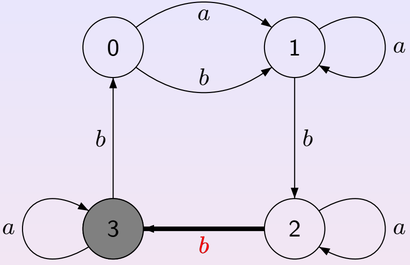
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



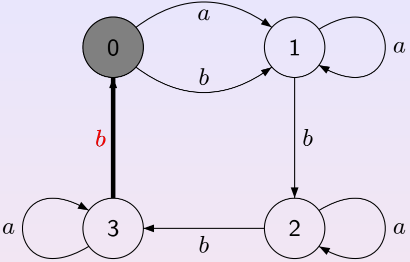
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



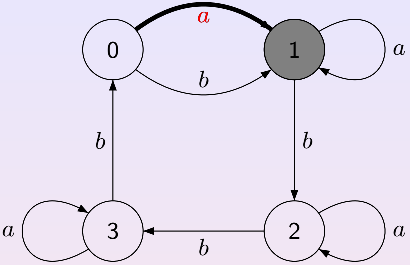
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



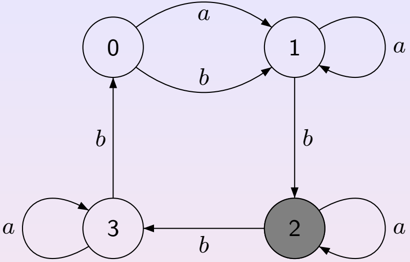
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



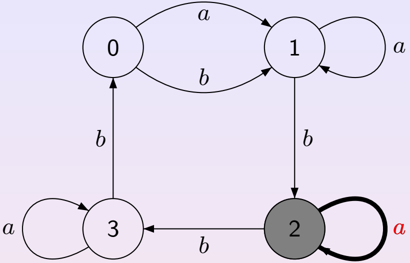
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



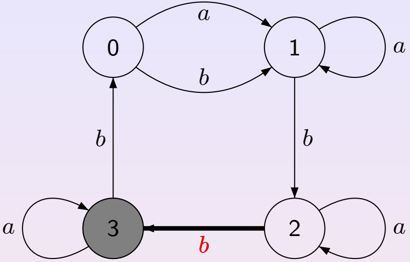
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



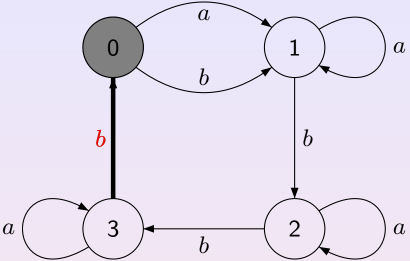
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



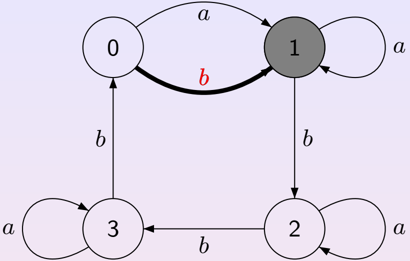
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



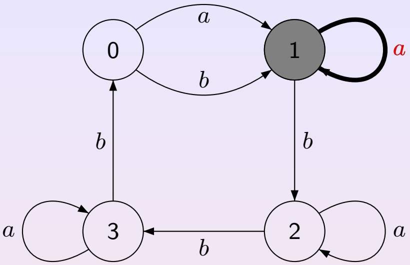
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



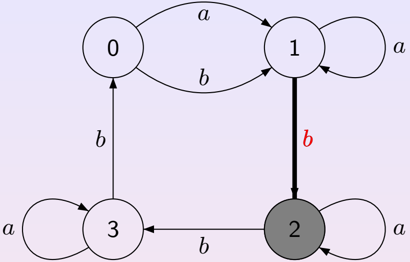
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



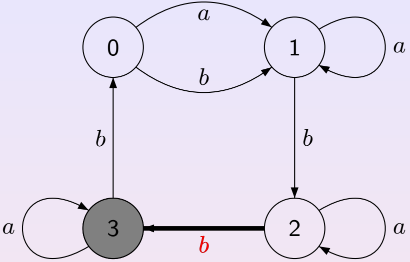
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



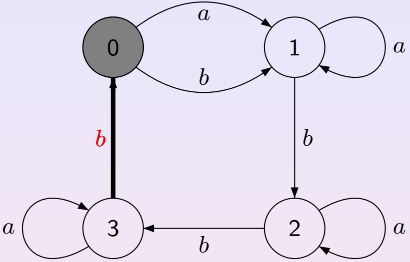
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



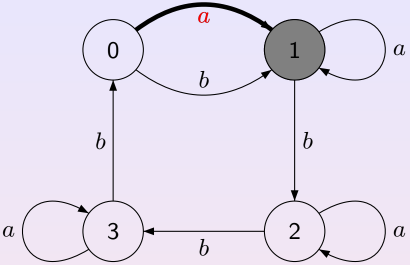
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



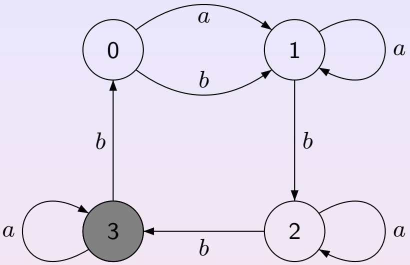
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



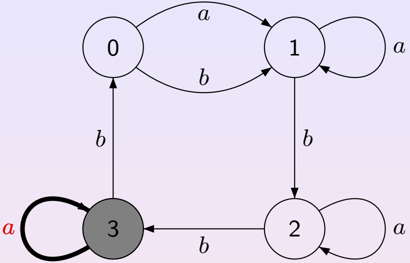
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



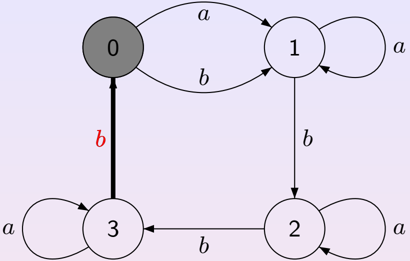
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



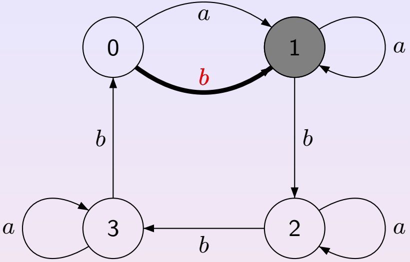
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



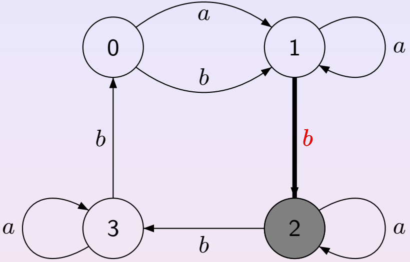
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



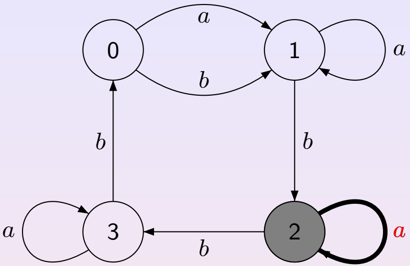
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



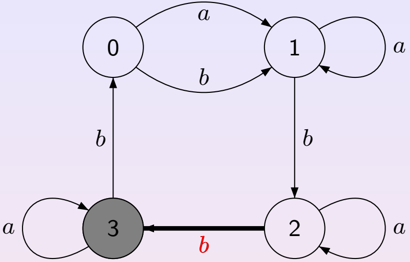
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



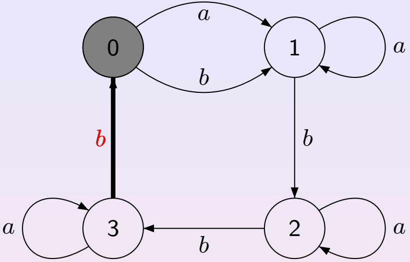
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



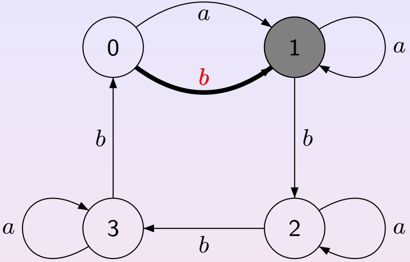
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



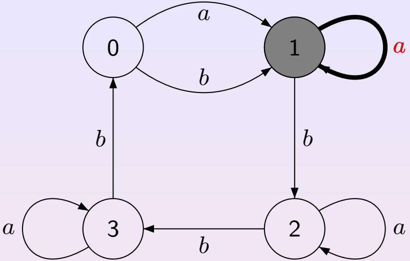
A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.



A reset word is $abbbabbba$: applying it at any state brings the automaton to the state 1.

The notion was formalized in 1964 in a paper by Jan Černý (Poznámka k homogénnym eksperimentom s konečnými automatami, Matematicko-fyzikalny Časopis Slovensk. Akad. Vied, 14, no.3, 208–216 [in Slovak]) though implicitly it had been around since at least 1956.

The notion was formalized in 1964 in a paper by Jan Černý (Poznámka k homogénnym eksperimentom s konečnými automatami, Matematicko-fyzikalny Časopis Slovensk. Akad. Vied, 14, no.3, 208–216 [in Slovak]) though implicitly it had been around since at least 1956.

The idea of synchronization is pretty natural and of obvious importance: we aim to restore control over a device whose current state is not known.

7. Cerný's Paper

The notion was formalized in 1964 in a paper by Jan Černý (Poznámka k homogénnym eksperimentom s konečnými automatami, Matematicko-fyzikalny Časopis Slovensk. Akad. Vied, 14, no.3, 208–216 [in Slovak]) though implicitly it had been around since at least 1956.

The idea of synchronization is pretty natural and of obvious importance: we aim to restore control over a device whose current state is not known.

Think of a satellite which loops around the Moon and cannot be controlled from the Earth while "behind" the Moon (Černý's original motivation).

8. Ashby's Ghost Taming Automaton

The earliest synchronizing automaton that I was able to trace back in the literature appeared in Ross Ashby's 'An Introduction to Cybernetics' (1956), pp. 60–61.

The earliest synchronizing automaton that I was able to trace back in the literature appeared in Ross Ashby's 'An Introduction to Cybernetics' (1956), pp. 60–61.

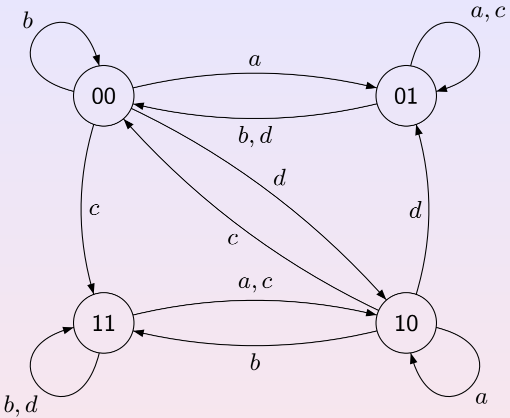
'**4/15. Materiality**. *The reader may now like to test the methods of this chapter as an aid to solving the problem set by the following letter. It justifies the statement made in S.1/2 that cybernetics is not bound to the properties found in terrestrial matter, nor does it draw its laws from them. What is important in cybernetics is the extent to which the observed behaviour is regular and reproducible.*'

The earliest synchronizing automaton that I was able to trace back in the literature appeared in Ross Ashby's 'An Introduction to Cybernetics' (1956), pp. 60–61.

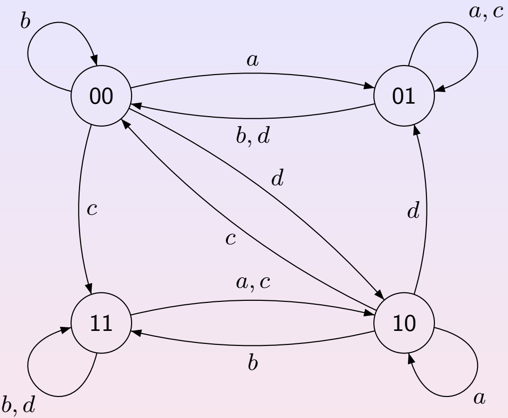
The letter presents a puzzle about two ghostly noises, Singing and Laughter, in a haunted mansion. Each of the noises can be either on or off, and their behaviour depends on combinations of two possible actions, playing the organ or burning incense.

The earliest synchronizing automaton that I was able to trace back in the literature appeared in Ross Ashby's 'An Introduction to Cybernetics' (1956), pp. 60–61.

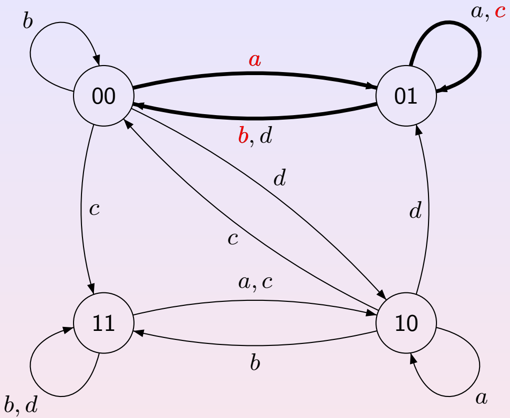
The letter presents a puzzle about two ghostly noises, Singing and Laughter, in a haunted mansion. Each of the noises can be either on or off, and their behaviour depends on combinations of two possible actions, playing the organ or burning incense.
Under a suitable encoding, this leads to an automaton with 4 states and 4 input letters shown in the next slide.



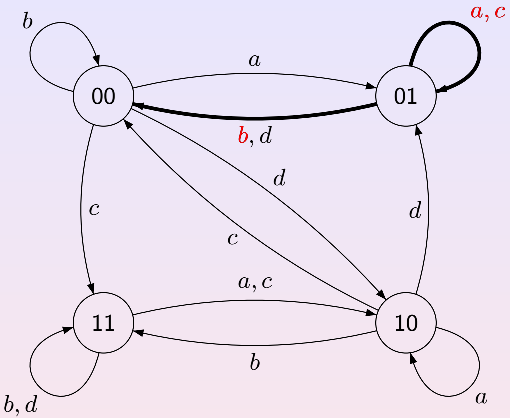
00
01
10
11
b
a, c
a
b, d
d
d
c
c
a, c
b
b, d
a



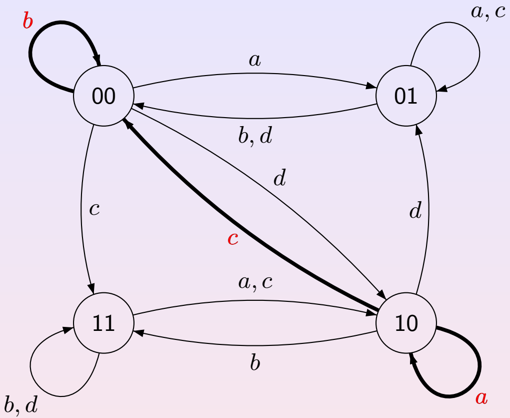
It is easy to see that this is a synchronizing automaton and acb is its shortest reset word.



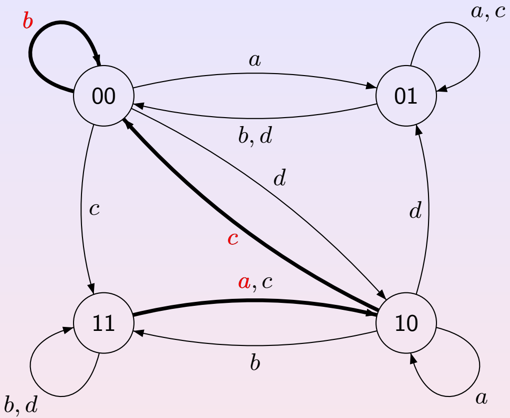
It is easy to see that this is a synchronizing automaton and acb is its shortest reset word.



It is easy to see that this is a synchronizing automaton and acb is its shortest reset word.



It is easy to see that this is a synchronizing automaton and acb is its shortest reset word.



It is easy to see that this is a synchronizing automaton and acb is its shortest reset word.

It is not surprising that synchronizing automata were re-invented a number of times:

It is not surprising that synchronizing automata were re-invented a number of times:

• The notion was very natural by itself and fitted fairly well in what was considered as the mainstream of automata theory in the 1960s.

10. Other Sources

It is not surprising that synchronizing automata were re-invented a number of times:

• The notion was very natural by itself and fitted fairly well in what was considered as the mainstream of automata theory in the 1960s.

• Černý's paper published in Slovak language remained unknown in the English-speaking world for quite a long time.

It is not surprising that synchronizing automata were re-invented a number of times:

• The notion was very natural by itself and fitted fairly well in what was considered as the mainstream of automata theory in the 1960s.

• Černý's paper published in Slovak language remained unknown in the English-speaking world for quite a long time.

Example: A. E. Laemmel, B. Rudner, Study of the application of coding theory, Report PIBEP-69-034, Polytechnic Inst. Brooklyn, Dept. Electrophysics, Farmingdale, N.Y., 94 pp.

Suppose we deal with data presented as a huge word w in some finite source alphabet Θ, and we know—or can estimate—the probability of occurrence in w for each letter from Θ.

Suppose we deal with data presented as a huge word w in some finite source alphabet Θ, and we know—or can estimate—the probability of occurrence in w for each letter from Θ.

A good example is a long text in a natural language, like Marcel Proust's "À la recherche du temps perdu" with its approx. 9,609,000 characters, each letter and space being counted as one character.

Suppose we deal with data presented as a huge word w in some finite source alphabet Θ, and we know—or can estimate—the probability of occurrence in w for each letter from Θ.

A good example is a long text in a natural language, like Marcel Proust's "À la recherche du temps perdu" with its approx. 9,609,000 characters, each letter and space being counted as one character. We can quite accurately estimate the probability of occurrence in this text for each character using available information about relative frequencies of letters in the French language.

Suppose we deal with data presented as a huge word w in some finite source alphabet Θ, and we know—or can estimate—the probability of occurrence in w for each letter from Θ.

A good example is a long text in a natural language, like Marcel Proust's "À la recherche du temps perdu" with its approx. 9,609,000 characters, each letter and space being counted as one character. We can quite accurately estimate the probability of occurrence in this text for each character using available information about relative frequencies of letters in the French language. For instance, 'e' occurs in French words approx. twice as often as 'a' and the frequency of occurrence of 'k' is less than 0.9% of that of 'l'.

If we want to digitalize the data (for storing or transmitting them), we have to encode the letters of Θ with some words over a smaller alphabet Σ, usually, the binary alphabet $\{0, 1\}$.

If we want to digitalize the data (for storing or transmitting them), we have to encode the letters of Θ with some words over a smaller alphabet Σ, usually, the binary alphabet $\{0, 1\}$.
An encoding of letters by binary words of constant length (such as ANSII-codes) requires $\lceil \log_2 |\Theta| \rceil$ bits for each letter and thus $|w| \cdot \lceil \log_2 |\Theta| \rceil$ bits for the whole word w.

If we want to digitalize the data (for storing or transmitting them), we have to encode the letters of Θ with some words over a smaller alphabet Σ, usually, the binary alphabet $\{0,1\}$.
An encoding of letters by binary words of constant length (such as ANSII-codes) requires $\lceil \log_2 |\Theta| \rceil$ bits for each letter and thus $|w| \cdot \lceil \log_2 |\Theta| \rceil$ bits for the whole word w. However, by a clever variable-length encoding we may save much space (in the case of data storage) and/or time (in the case of data transmission).

If we want to digitalize the data (for storing or transmitting them), we have to encode the letters of Θ with some words over a smaller alphabet Σ, usually, the binary alphabet $\{0,1\}$.
An encoding of letters by binary words of constant length (such as ANSII-codes) requires $\lceil \log_2 |\Theta| \rceil$ bits for each letter and thus $|w| \cdot \lceil \log_2 |\Theta| \rceil$ bits for the whole word w. However, by a clever variable-length encoding we may save much space (in the case of data storage) and/or time (in the case of data transmission). For this, we should encode letters that occur in w more frequently by shorter binary words while letters with low probability of occurrence in w may be encoded by longer binary words without much harm.

If we want to digitalize the data (for storing or transmitting them), we have to encode the letters of Θ with some words over a smaller alphabet Σ, usually, the binary alphabet $\{0,1\}$.

An encoding of letters by binary words of constant length (such as ANSII-codes) requires $\lceil \log_2 |\Theta| \rceil$ bits for each letter and thus $|w| \cdot \lceil \log_2 |\Theta| \rceil$ bits for the whole word w. However, by a clever variable-length encoding we may save much space (in the case of data storage) and/or time (in the case of data transmission). For this, we should encode letters that occur in w more frequently by shorter binary words while letters with low probability of occurrence in w may be encoded by longer binary words without much harm.

This simple idea was already used in Morse code of the 19th century: 'e', the most common letter in English has the shortest Morse code, a single dot.

A complication has to be taken into account when variable-length encoding is used: the process of decoding, i.e., restoring the original word w from a stream of bits in that w has been encoded, may be not easy in general.

A complication has to be taken into account when variable-length encoding is used: the process of decoding, i.e., restoring the original word w from a stream of bits in that w has been encoded, may be not easy in general.
There is however a class of encodings for which this complication does not appear.

A complication has to be taken into account when variable-length encoding is used: the process of decoding, i.e., restoring the original word w from a stream of bits in that w has been encoded, may be not easy in general.
There is however a class of encodings for which this complication does not appear. A prefix code is a set X of words over some alphabet such that no word of X is a prefix of another word of X.

A complication has to be taken into account when variable-length encoding is used: the process of decoding, i.e., restoring the original word w from a stream of bits in that w has been encoded, may be not easy in general.
There is however a class of encodings for which this complication does not appear. A prefix code is a set X of words over some alphabet such that no word of X is a prefix of another word of X. Data encoded with a prefix code can be decoded on-the-fly: a decoder just keeps finding and removing prefixes that form valid code words from the incoming stream.

A complication has to be taken into account when variable-length encoding is used: the process of decoding, i.e., restoring the original word w from a stream of bits in that w has been encoded, may be not easy in general.
There is however a class of encodings for which this complication does not appear. A prefix code is a set X of words over some alphabet such that no word of X is a prefix of another word of X. Data encoded with a prefix code can be decoded on-the-fly: a decoder just keeps finding and removing prefixes that form valid code words from the incoming stream. At the same time, it is known that the most economical binary presentation of data that can be achieved by any variable-length encoding always can be achieved by a suitable encoding with a prefix code.

Consider the sentence YOU USE A CODE C.

Consider the sentence YOU USE A CODE C. It involves 9 different characters (8 letters and space) and has length 16 so that every its constant-length binary encoding requires $16 \cdot \lceil \log_2 9 \rceil = 64$ bits.

Consider the sentence YOU USE A CODE C. It involves 9 different characters (8 letters and space) and has length 16 so that every its constant-length binary encoding requires $16 \cdot \lceil \log_2 9 \rceil = 64$ bits.
The prefix code

$$X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$$

allows one to encode the sentence more efficiently:

space	C	E	O	U	A	D	S	Y
10	000	010	110	111	0010	0011	0110	0111

Consider the sentence YOU USE A CODE C. It involves 9 different characters (8 letters and space) and has length 16 so that every its constant-length binary encoding requires $16 \cdot \lceil \log_2 9 \rceil = 64$ bits.
The prefix code

$$X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$$

allows one to encode the sentence more efficiently:

space	C	E	O	U	A	D	S	Y
10	000	010	110	111	0010	0011	0110	0111

The sentence YOU USE A CODE C is then encoded with the binary word

$$0111|110|111|10|111|0110|010|10|0010|10|000|110|0011|010|10|000$$

of length 48 (vertical lines separating code words are inserted for readability only).

Consider the sentence YOU USE A CODE C. It involves 9 different characters (8 letters and space) and has length 16 so that every its constant-length binary encoding requires $16 \cdot \lceil \log_2 9 \rceil = 64$ bits.
The prefix code

$$X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$$

allows one to encode the sentence more efficiently:

space	C	E	O	U	A	D	S	Y
10	000	010	110	111	0010	0011	0110	0111

The sentence YOU USE A CODE C is then encoded with the binary word

$$0111|110|111|10|111|0110|010|10|0010|10|000|110|0011|010|10|000$$

of length 48 (vertical lines separating code words are inserted for readability only). We have thus reduced the binary representation size by 25%.

A prefix code over a finite alphabet Σ is a set X of words in Σ^* such that no word of X is a prefix of another word of X.

A prefix code over a finite alphabet Σ is a set X of words in Σ^* such that no word of X is a prefix of another word of X. A prefix code is maximal if it is not contained in another prefix code over the same alphabet.

A prefix code over a finite alphabet Σ is a set X of words in Σ^* such that no word of X is a prefix of another word of X. A prefix code is maximal if it is not contained in another prefix code over the same alphabet. A maximal prefix code X over Σ is synchronized if there is a word $x \in X^*$ such that for any word $w \in \Sigma^*$, one has $wx \in X^*$. Such a word x is called a synchronizing word for X.

A prefix code over a finite alphabet Σ is a set X of words in Σ^* such that no word of X is a prefix of another word of X. A prefix code is maximal if it is not contained in another prefix code over the same alphabet. A maximal prefix code X over Σ is synchronized if there is a word $x \in X^*$ such that for any word $w \in \Sigma^*$, one has $wx \in X^*$. Such a word x is called a synchronizing word for X. The advantage of synchronized codes is that they are able to recover after a loss of synchronization between the decoder and the coder caused by channel errors.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent 0 0 0

$\Sigma = \{0,1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent $\quad 0\,0\,0 \mid 0\,0\,1\,0$

$\Sigma = \{0,1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

$$\text{Sent} \quad 0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$$

The vertical lines show the partition into code words.

$\Sigma = \{0,1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$
Received	1 0 0 0 0 1 0 0 1 1 1 ...

The vertical lines show the partition into code words.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$
Received	1 0 0 0 0 1 0 0 1 1 1 ...

The vertical lines show the partition into code words.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$
Received	1 0 0 0 0 1 0 0 1 1 1 ...
Decoded	1 0

The vertical lines show the partition into code words.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, . . . is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$
Received	1 0 0 0 0 1 0 0 1 1 1 . . .
Decoded	$1\,0 \mid 0\,0\,0$

The vertical lines show the partition into code words.

$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid 0\,1\,1\,1 \mid \ldots$
Received	1 0 0 0 0 1 0 0 1 1 1 ...
Decoded	$1\,0 \mid 0\,0\,0 \mid 1\,0$

The vertical lines show the partition into code words.

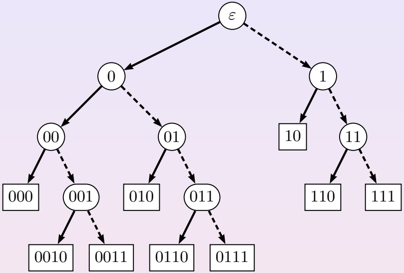
$\Sigma = \{0, 1\}$, $X = \{000, 0010, 0011, 010, 0110, 0111, 10, 110, 111\}$. Then X is a maximal prefix code and one can easily check that each of the words 010, 011110, 011111110, ... is a synchronizing word for X.

Sent	$0\,0\,0 \mid 0\,0\,1\,0 \mid \mathbf{0\,1\,1\,1} \mid \ldots$
Received	$1\,0\,0\;\;0\;0\,1\,0\;\;\;0\,1\,1\,1\;\ldots$
Decoded	$1\,0 \mid 0\,0\,0 \mid 1\,0 \mid \mathbf{0\,1\,1\,1} \mid \ldots$

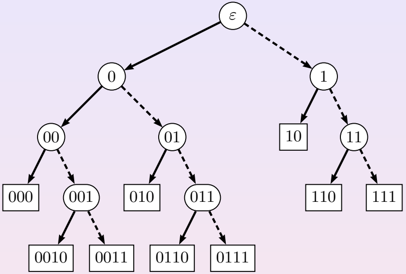
The vertical lines show the partition into code words.
The boldfaced code words indicate the position at which the decoder resynchronizes.

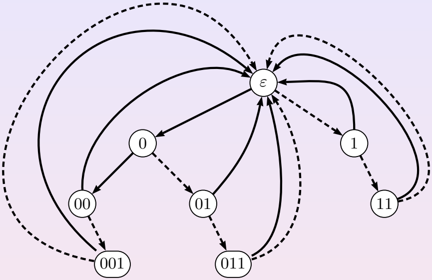
If X is a finite maximal prefix code, then its decoding can be implemented by a DFA.

If X is a finite maximal prefix code, then its decoding can be implemented by a DFA.

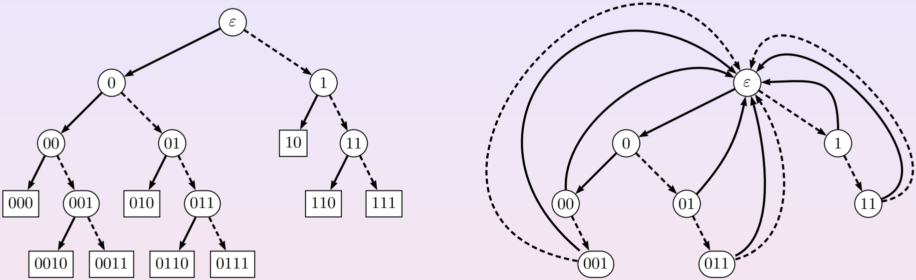


If X is a finite maximal prefix code, then its decoding can be implemented by a DFA.





If X is a finite maximal prefix code, then its decoding can be implemented by a DFA.



Synchronized codes precisely correspond to synchronizing automata!

Since the 60s synchronizing automata have been considered as a useful tool for testing of reactive systems (first circuits, later protocols)

Since the 60s synchronizing automata have been considered as a useful tool for testing of reactive systems (first circuits, later protocols) and have been also applied in coding theory.

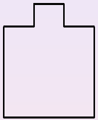
18. Re-inventing by Engineers

Since the 60s synchronizing automata have been considered as a useful tool for testing of reactive systems (first circuits, later protocols) and have been also applied in coding theory.
In the 80s, the notion was reinvented by engineers working in a branch of robotics which deals with part handling problems in industrial automation.

Since the 60s synchronizing automata have been considered as a useful tool for testing of reactive systems (first circuits, later protocols) and have been also applied in coding theory.

In the 80s, the notion was reinvented by engineers working in a branch of robotics which deals with part handling problems in industrial automation.

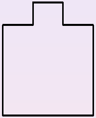
Suppose that one of the parts of a certain device has the following shape:



Since the 60s synchronizing automata have been considered as a useful tool for testing of reactive systems (first circuits, later protocols) and have been also applied in coding theory.

In the 80s, the notion was reinvented by engineers working in a branch of robotics which deals with part handling problems in industrial automation.

Suppose that one of the parts of a certain device has the following shape:



Such parts arrive at manufacturing sites in boxes and they need to be sorted and oriented before assembly.

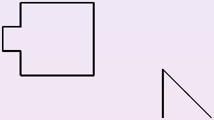
Assume that only four initial orientations of the part shown above are possible, namely, the following ones:

Assume that only four initial orientations of the part shown above are possible, namely, the following ones:

Suppose that prior the assembly the part should take the 'bump-left' orientation (the second one in the picture). Thus, one has to construct an orienter which action will put the part in the prescribed position independently of its initial orientation.

We put parts to be oriented on a conveyer belt which takes them to the assembly point and let the stream of the parts encounter a series of passive obstacles of two types (*tall* and *short*) positioned along the belt.

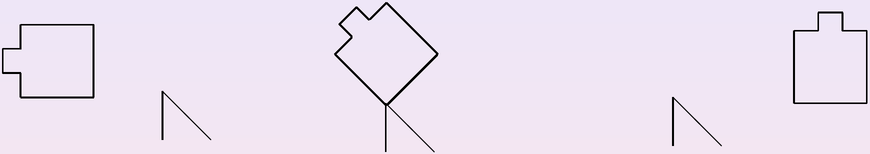
We put parts to be oriented on a conveyer belt which takes them to the assembly point and let the stream of the parts encounter a series of passive obstacles of two types (*tall* and *short*) positioned along the belt.
A tall obstacle is tall enough so that any part on the belt encounters this obstacle by its rightmost low angle.



We put parts to be oriented on a conveyer belt which takes them to the assembly point and let the stream of the parts encounter a series of passive obstacles of two types (*tall* and *short*) positioned along the belt.
A tall obstacle is tall enough so that any part on the belt encounters this obstacle by its rightmost low angle.

Being carried by the belt, the part then is forced to turn 90° clockwise.

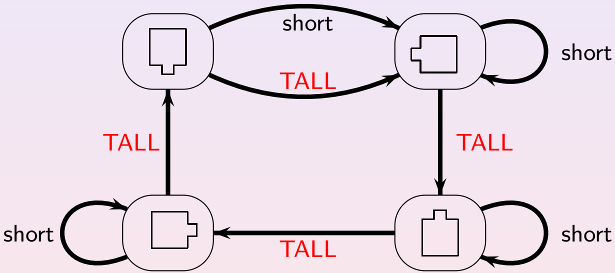
We put parts to be oriented on a conveyer belt which takes them to the assembly point and let the stream of the parts encounter a series of passive obstacles of two types (*tall* and *short*) positioned along the belt.
A tall obstacle is tall enough so that any part on the belt encounters this obstacle by its rightmost low angle.



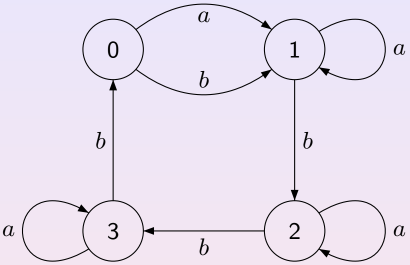
Being carried by the belt, the part then is forced to turn 90° clockwise.

A short obstacle has the same effect whenever the part is in the "bump-down" orientation; otherwise it does not touch the part which therefore passes by without changing the orientation.

A short obstacle has the same effect whenever the part is in the "bump-down" orientation; otherwise it does not touch the part which therefore passes by without changing the orientation.
The following schema summarizes how the obstacles effect the orientation of the part in question:

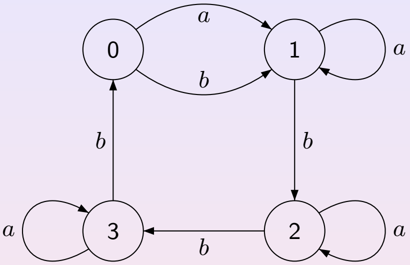


We met this picture a few slides ago:



– this was our example of a synchronizing automaton, and we saw that $abbbabbba$ is a reset sequence of actions.

We met this picture a few slides ago:



– this was our example of a synchronizing automaton, and we saw that $abbbabbba$ is a reset sequence of actions. Hence the series of obstacles

short-TALL-TALL-TALL-short-TALL-TALL-TALL-short

yields the desired sensorless orienter.

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length.

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position.

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccc} 0 & \mapsto & 11 \\ 1 & \mapsto & 12 \\ 2 & \mapsto & 20 \end{array}$$

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccccc} 0 & \mapsto & 11 & \mapsto & 1212 \\ 1 & \mapsto & 12 & \mapsto & 1220 \\ 2 & \mapsto & 20 & \mapsto & 2011 \end{array}$$

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccccccc} 0 & \mapsto & 11 & \mapsto & 1212 & \mapsto & 12201220 \\ 1 & \mapsto & 12 & \mapsto & 1220 & \mapsto & 12202011 \\ 2 & \mapsto & 20 & \mapsto & 2011 & \mapsto & 20111212 \end{array}$$

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccccccccc}
0 & \mapsto & 11 & \mapsto & 1212 & \mapsto & 12201220 & \mapsto & 1220201112202011 \\
1 & \mapsto & 12 & \mapsto & 1220 & \mapsto & 12202011 & \mapsto & 1220201120111212 \\
2 & \mapsto & 20 & \mapsto & 2011 & \mapsto & 20111212 & \mapsto & 2011121212201220
\end{array}$$

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccccccccc}
0 & \mapsto & 11 & \mapsto & 1212 & \mapsto & 12201220 & \mapsto & 122020\mathbf{1}112202011 \\
1 & \mapsto & 12 & \mapsto & 1220 & \mapsto & 12202011 & \mapsto & 122020\mathbf{1}120111212 \\
2 & \mapsto & 20 & \mapsto & 2011 & \mapsto & 20111212 & \mapsto & 201112\mathbf{1}212201220
\end{array}$$

Thus, τ satisfies the coincidence condition (with $k = 4$, $m = 7$).

A substitution on a finite alphabet X is a map $\sigma : X \to X^+$; the substitution is said to be of constant length if all words $\sigma(x)$, $x \in X$, have the same length. One says that σ satisfies the coincidence condition if there exist positive integers m and k such that all words $\sigma^k(x)$ have the same letter in the m-th position. For an example, consider the substitution τ on $X = \{0,1,2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$. Calculate the iterations of τ up to τ^4:

$$\begin{array}{ccccccccc}
0 & \mapsto & 11 & \mapsto & 1212 & \mapsto & 12201220 & \mapsto & 122020\mathbf{1}112202011 \\
1 & \mapsto & 12 & \mapsto & 1220 & \mapsto & 12202011 & \mapsto & 122020\mathbf{1}120111212 \\
2 & \mapsto & 20 & \mapsto & 2011 & \mapsto & 20111212 & \mapsto & 201112\mathbf{1}212201220
\end{array}$$

Thus, τ satisfies the coincidence condition (with $k = 4$, $m = 7$).
The coincidence condition completely characterizes the constant length substitutions that give rise to dynamical systems measure-theoretically isomorphic to a translation on a compact Abelian group (Dekking, 1978).

There is a straightforward bijection between DFAs and constant length substitutions. Each DFA $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ with $\Sigma = \{a_1, \dots, a_\ell\}$ defines a length ℓ substitution on Q that maps every $q \in Q$ to the word $(q \,.\, a_1) \dots (q \,.\, a_\ell) \in Q^+$.

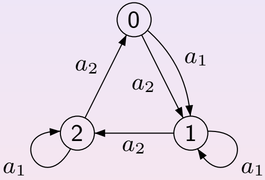
There is a straightforward bijection between DFAs and constant length substitutions. Each DFA $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ with $\Sigma = \{a_1, \ldots, a_\ell\}$ defines a length ℓ substitution on Q that maps every $q \in Q$ to the word $(q \,.\, a_1) \ldots (q \,.\, a_\ell) \in Q^+$. For instance, the automaton

$$\begin{tikzcd} 0 \arrow[r, bend left, "a"] \arrow[r, bend right, "b"'] & 1 \arrow[loop right, "a"] \arrow[d, "b"] \\ 3 \arrow[u, "b"] \arrow[loop left, "a"] & 2 \arrow[l, "b"] \arrow[loop right, "a"] \end{tikzcd}$$

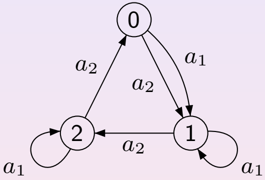
induces the substitution $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 23,\ 3 \mapsto 30.$

Conversely, each substitution $\sigma : X \to X^+$ such that all words $\sigma(x)$, $x \in X$, have the same length ℓ gives rise to a DFA for which X is the state set and which has ℓ input letters $a_1, \dots, a_\ell$ acting on X as follows: $x \, . \, a_i$ is the symbol in the i-th position of the word $\sigma(x)$.

Conversely, each substitution $\sigma : X \to X^+$ such that all words $\sigma(x)$, $x \in X$, have the same length ℓ gives rise to a DFA for which X is the state set and which has ℓ input letters $a_1, \ldots, a_\ell$ acting on X as follows: $x \, . \, a_i$ is the symbol in the i-th position of the word $\sigma(x)$. For instance, the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$ induces the automaton:



Conversely, each substitution $\sigma : X \to X^+$ such that all words $\sigma(x)$, $x \in X$, have the same length ℓ gives rise to a DFA for which X is the state set and which has ℓ input letters $a_1, \dots, a_\ell$ acting on X as follows: $x \,.\, a_i$ is the symbol in the i-th position of the word $\sigma(x)$. For instance, the substitution τ on $X = \{0, 1, 2\}$ defined by $0 \mapsto 11,\ 1 \mapsto 12,\ 2 \mapsto 20$ induces the automaton:



Under this bijection substitutions satisfying the coincidence condition correspond precisely to synchronizing automata, and moreover, given a substitution, the number of iterations at which the coincidence first occurs is equal to the minimum length of reset word for the corresponding automaton.

26. An Algebraic Framework

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set.

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set. A term in the language of such unary algebras is an expression t of the form $x\,.\,w$, where x is a variable and w is a word over an alphabet Σ.

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set. A term in the language of such unary algebras is an expression t of the form $x\,.\,w$, where x is a variable and w is a word over an alphabet Σ. An identity is a formal equality between two terms. A DFA $\mathscr{A} = \langle Q, \Sigma, \delta\rangle$ satisfies an identity $t_1 = t_2$, where the words involved in the terms t_1 and t_2 are over Σ, if t_1 and t_2 take the same value under each interpretation of their variables in the set Q.

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set. A term in the language of such unary algebras is an expression t of the form $x\,.\,w$, where x is a variable and w is a word over an alphabet Σ. An identity is a formal equality between two terms. A DFA $\mathscr{A} = \langle Q, \Sigma, \delta\rangle$ satisfies an identity $t_1 = t_2$, where the words involved in the terms t_1 and t_2 are over Σ, if t_1 and t_2 take the same value under each interpretation of their variables in the set Q.
Identities of unary algebras can be of the from either $x\,.\,u = x\,.\,v$ or $x\,.\,u = y\,.\,v$ with $x \neq y$.

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set. A term in the language of such unary algebras is an expression t of the form $x \,.\, w$, where x is a variable and w is a word over an alphabet Σ. An identity is a formal equality between two terms. A DFA $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ satisfies an identity $t_1 = t_2$, where the words involved in the terms t_1 and t_2 are over Σ, if t_1 and t_2 take the same value under each interpretation of their variables in the set Q.
Identities of unary algebras can be of the from either $x \,.\, u = x \,.\, v$ or $x \,.\, u = y \,.\, v$ with $x \neq y$. A DFA is synchronizing if and only if it satisfies an identity of the second type. Thus studying synchronizing automata may be considered as a part of the equational logic of unary algebras.

One may treat DFAs as unary algebras since each letter of the input alphabet defines a unary operation on the state set. A term in the language of such unary algebras is an expression t of the form $x\,.\,w$, where x is a variable and w is a word over an alphabet Σ. An identity is a formal equality between two terms. A DFA $\mathscr{A} = \langle Q, \Sigma, \delta \rangle$ satisfies an identity $t_1 = t_2$, where the words involved in the terms t_1 and t_2 are over Σ, if t_1 and t_2 take the same value under each interpretation of their variables in the set Q.
Identities of unary algebras can be of the from either $x\,.\,u = x\,.\,v$ or $x\,.\,u = y\,.\,v$ with $x \neq y$. A DFA is synchronizing if and only if it satisfies an identity of the second type. Thus studying synchronizing automata may be considered as a part of the equational logic of unary algebras. In particular, synchronizing automata over a fixed alphabet form a pseudovariety of unary algebras.

In DNA-computing, there is fast progressing work by Ehud Shapiro's group on "*soup of automata*" (Programmable and autonomous computing machine made of biomolecules, Nature 414, no.1 (November 22, 2001) 430–434; DNA molecule provides a computing machine with both data and fuel, Proc. National Acad. Sci. USA 100 (2003) 2191–2196, etc).

In DNA-computing, there is fast progressing work by Ehud Shapiro's group on "*soup of automata*" (Programmable and autonomous computing machine made of biomolecules, Nature 414, no.1 (November 22, 2001) 430–434; DNA molecule provides a computing machine with both data and fuel, Proc. National Acad. Sci. USA 100 (2003) 2191–2196, etc).
They have produced a solution containing 3×10^{12} identical DNA-based automata per μl.

In DNA-computing, there is fast progressing work by Ehud Shapiro's group on "*soup of automata*" (Programmable and autonomous computing machine made of biomolecules, Nature 414, no.1 (November 22, 2001) 430–434; DNA molecule provides a computing machine with both data and fuel, Proc. National Acad. Sci. USA 100 (2003) 2191–2196, etc).
They have produced a solution containing 3×10^{12} identical DNA-based automata per μl. These automata can work in parallel on different inputs (DNA strands), thus ending up in different and unpredictable states.

In DNA-computing, there is fast progressing work by Ehud Shapiro's group on "*soup of automata*" (Programmable and autonomous computing machine made of biomolecules, Nature 414, no.1 (November 22, 2001) 430–434; DNA molecule provides a computing machine with both data and fuel, Proc. National Acad. Sci. USA 100 (2003) 2191–2196, etc).
They have produced a solution containing 3×10^{12} identical DNA-based automata per μl. These automata can work in parallel on different inputs (DNA strands), thus ending up in different and unpredictable states. One has to feed the automata with an reset sequence (again encoded by a DNA-strand) in order to get them ready for a new use.

- From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• Synchronizing automata constitute an interesting combinatorial object. Their studies from a combinatorial viewpoint are mainly motivated by the Černý Conjecture.

28. Outline of these Lectures

• From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• Synchronizing automata constitute an interesting combinatorial object. Their studies from a combinatorial viewpoint are mainly motivated by the Černý Conjecture.

• Interesting connections to symbolic dynamics have led to the Road Coloring Problem.

• From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• Synchronizing automata constitute an interesting combinatorial object. Their studies from a combinatorial viewpoint are mainly motivated by the Černý Conjecture.

• Interesting connections to symbolic dynamics have led to the Road Coloring Problem.

• We present in detail a proof of the Černý Conjecture for the special case of aperiodic automata.

• From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• Synchronizing automata constitute an interesting combinatorial object. Their studies from a combinatorial viewpoint are mainly motivated by the Černý Conjecture.

• Interesting connections to symbolic dynamics have led to the Road Coloring Problem.

• We present in detail a proof of the Černý Conjecture for the special case of aperiodic automata.

• There are also interesting connections with the Perron–Frobenius theory of non-negative matrices.

28. Outline of these Lectures

• From the viewpoint of applications, real or yet imaginary, algorithmic issues are of crucial importance.

• Synchronizing automata constitute an interesting combinatorial object. Their studies from a combinatorial viewpoint are mainly motivated by the Černý Conjecture.

• Interesting connections to symbolic dynamics have led to the Road Coloring Problem.

• We present in detail a proof of the Černý Conjecture for the special case of aperiodic automata.

• There are also interesting connections with the Perron–Frobenius theory of non-negative matrices.

• We also formulate several tantalizing open problems.