Language Development: Machine

„Star Trek“
1966

„ELIZA“
1965

„Intel“
1968

> More than iron, more than lead,
> more than gold I need
> electricity.
> I need it more than I need lamb
> or pork or lettuce or cucumber.
> I need it for my dreams.
>
> — Racter, *The Policeman's Beard Is Half Constructed*

„Racter“
1983

Google
1997

„Collin‘s structured perceptron“
2002

„Siri“
2011

„CERN discoverd the Higgs boson particle“
2012

„NLU with DNNs“
2015

T-NLG
17 billion parameters
2020

?
2023

Symbolic 1980
„Hand Crafted Rules“

Statistical 2011
„Hand Crafted Features“

2015 Representation Learning
„Hand Crafted NN Topologies“

1950
„Turing test“

1954
Georgetown–IBM experiment

1982
TCP/IP

1985
„WordNet“

1992
„Brill‘s tagger“

1996
„Linguatronic in S-class“

Noise Robustness
Echo Cancellation for hands-free telephone applications
Continuous speaker independent speech recognition
Speaker dependent recognition
Active vocabulary of about 300 words
Background adaptation
Dynamically loadable vocabulary

2007
„Freebase“

2011
„Watson wins Jeopardy“

2012
„ASR with DNNs“

2014
„Alexa“, „Cortana“

2016
„seq2seq DNN“
„NMT“

2017
„Multichannel CLDNNs for ASR“

Human ⇔ Machine

N
L
P

안녕하세요.

hello

你好

Bonjour.

Guten Tag.

こんにちは。

Natural language
„mix of explicite and implicite knowledge representation that is sometimes ambiguous“

SLU

```
<command language:”en-US”>
            greeting
</command>
```

```
{"command“  :"greeting",
 "attribute": null,
 "language“ : "en-US",
 "en-US"    : "US English",
 "en-GB"    : "GB English„ }
```

cmd_greeting()

Structured language
„all knowledge is explicite and provided by relations“

63 6d 64 5f 67 72 65
65 74 69 6e 67 28 29

Machine languagages
„needs to fulfill certain machine-criteria, e.g. Real-time parsing given limited

Punctuation

Natural Language Generation

Morphology

Canonicalization

Paraphrase natural language

Co-reference resolution

Word representation

Named Entity Recognition

Relation Extraction

Sentiment Analysis

Normalization

Question Answering

Speech Recognition

Parsing (dependency, syntactic, semantic, …)

Formatting

Grammar inference

Search

Recommendation Systems

Part of Speech Tagging

stemma

Text Summarization

lemma

Information Extraction

Dialog

(word) segmentation/tokenization

Segmentation

Prediction

Intent Detection

Query Expansion

Chatbot

Interestingness

Information Retrieval

Topic Segmentation

Text Categorization

Argumentation Mining

Textual Entailment

Translation

Language Modeling

Keyword spotting

Phonology

What is a character?

- **String**
- **Let Z be the set of all characters of a language:**

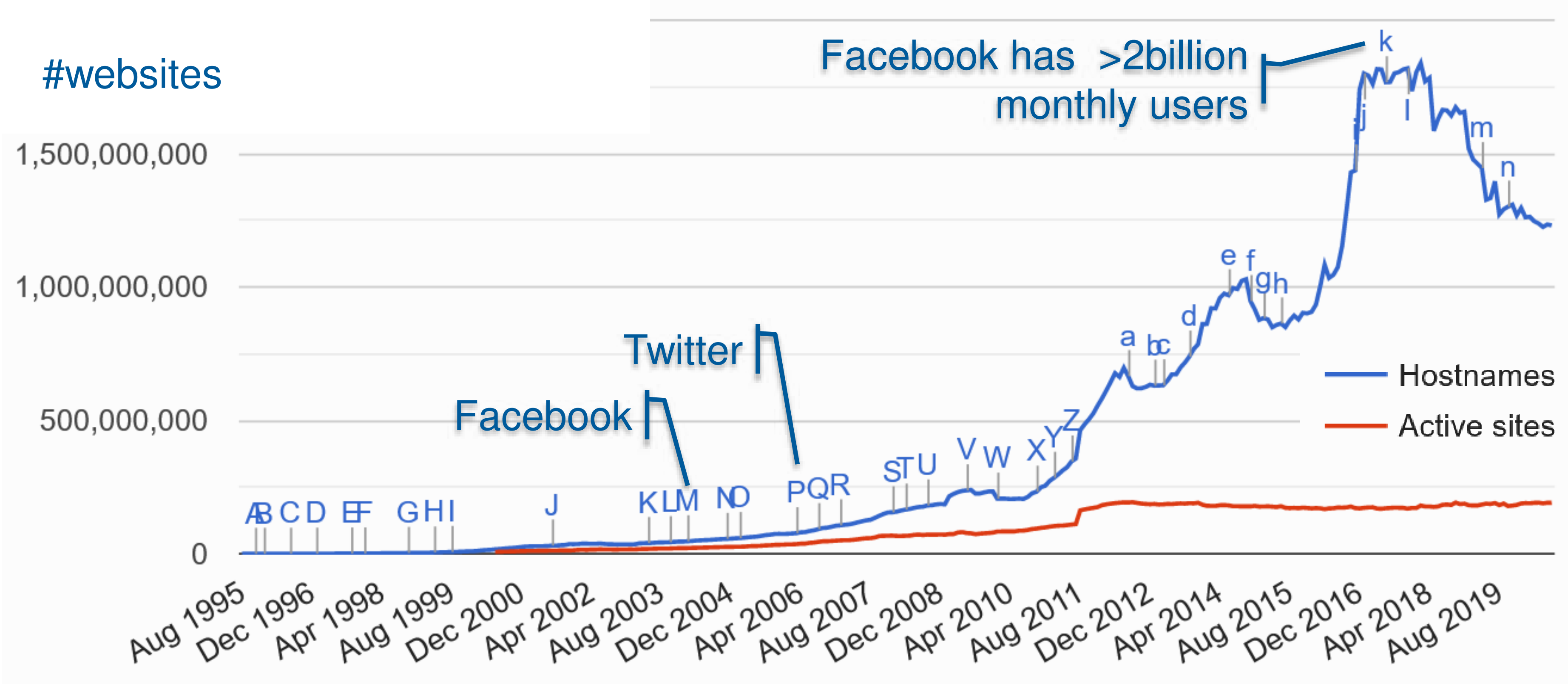
  German: aAbBcCdD.. !“ § $%&/()=?><;:_-.,|*+~#‘ …

  Chinese: 漢字 (Hànzì)
- **The number of characters in Z for English is, e.g.,**

  $|Z|=26+1$ including spaces

What is a document?

Receipts, books, website, ...



https://news.netcraft.com/archives/category/web-server-survey/

How to search for words?

In 2010, Google counted over a trillion web pages and 129,864,880 books.

How to find in them the origin of the word

„*supercalifragilisticexpialidocious*"

?

https://de.wikipedia.org/wiki/Supercalifragilisticexpialigetisch
http://booksearch.blogspot.com/2010/08/books-of-world-stand-up-and-be-counted.html

Problem description

A document consists of a string of characters of any length. There is a limited number of characters, but an almost unlimited number of documents. Searching for all occurrences of a given string is unsolvable for humans due to time constraints.

(one human life is not long enough to read all the texts of mankind, unfortunately)

What would be an example?

Search pattern : `supercalifragilisticexpialidocious`

```
Document: It's... supercalifragilisticexpialidocious! Even
though the sound of it is something quite atrocious If you
say it loud enough, you'll always sound precocious:
Supercalifragilisticexpialidocious!
```

What is the solution?

Search pattern : `supercalifragilisticexpialidocious`

Document: `It's... supercalifragilisticexpialidocious! Even though the sound of it is something quite atrocious If you say it loud enough, you'll always sound precocious: Supercalifragilisticexpialidocious!`

Are these all occurrences?

What problems are to be expected?

Search pattern: `supercalifragilisticexpialidocious`

```
Document: It's... supercalifragilisticexpialidocious! Even
though the sound of it is something quite atrocious If you
say it loud enough, you'll always sound precocious:
Supercalifragilisticexpialidocious!
```

- Are "ö" and "oe" the same?
- Is "ö" and "oe" the same?
- Hyphenation at line break?
- …

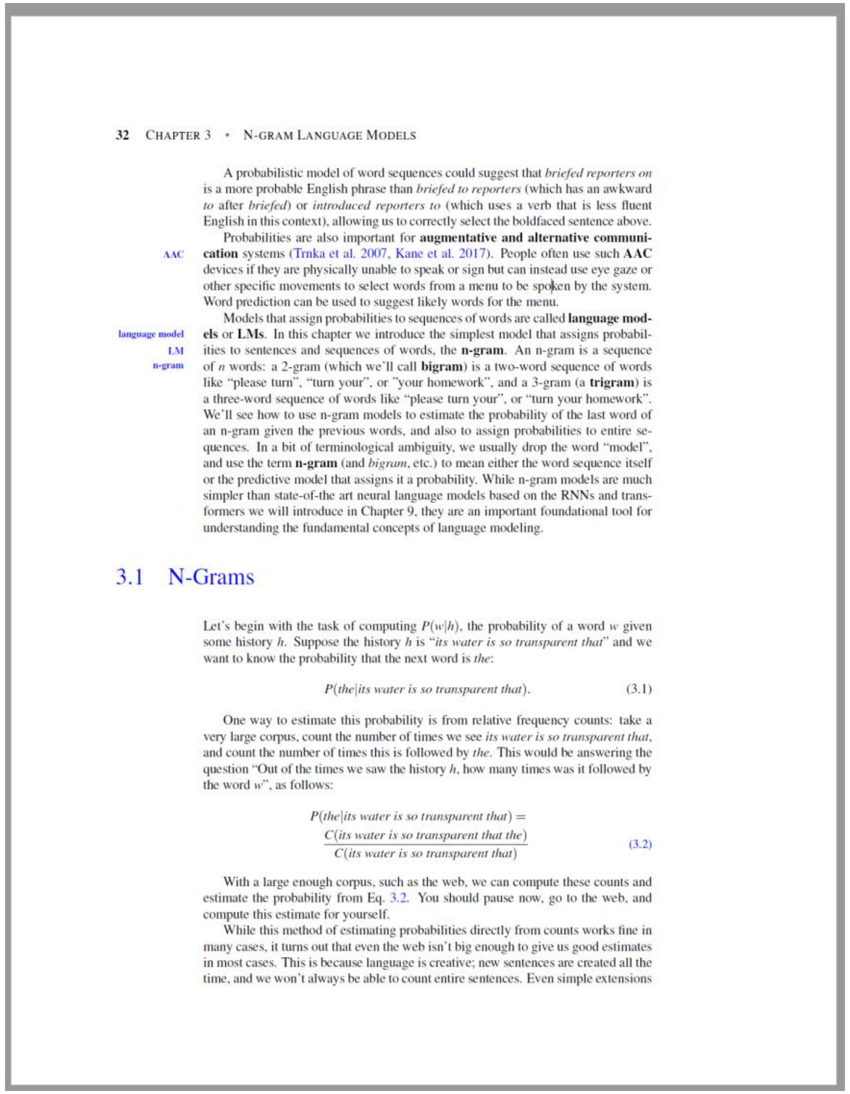
Exact formulation
of the problem

Problem description

A document consists of a string of characters of any length. There is a limited number of characters, but an almost unlimited number of documents. An arbitrary search mask is to be found in the documents.

- **The documents and all search masks have the same character set**
- **The search mask is constant (i.e. it does not change depending on the documents)**
- **The search mask consists of a finite long character string**

Real-World Example: What is an „N-gram“?



A probabilistic model of word sequences could suggest that *briefed reporters on* is a more probable English phrase than *briefed to reporters* (which has an awkward *to* after *briefed*) or *introduced reporters to* (which uses a verb that is less fluent English in this context), allowing us to correctly select the boldfaced sentence above.

Probabilities are also important for **augmentative and alternative communication** systems (Trnka et al. 2007, Kane et al. 2017). People often use such **AAC** devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

AAC

Models that assign probabilities to sequences of words are called **language models** or **LMs**. In this chapter we introduce the simplest model that assigns probabilities to sentences and sequences of words, the **n-gram**. An n-gram is a sequence of n words: a 2-gram (which we'll call **bigram**) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (a **trigram**) is a three-word sequence of words like "please turn your", or "turn your homework". We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word "model", and use the term **n-gram** (and *bigram*, etc.) to mean either the word sequence itself or the predictive model that assigns it a probability. While n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers we will introduce in Chapter 9, they are an important foundational tool for understanding the fundamental concepts of language modeling.

language model
LM
n-gram

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)} \tag{3.2}$$

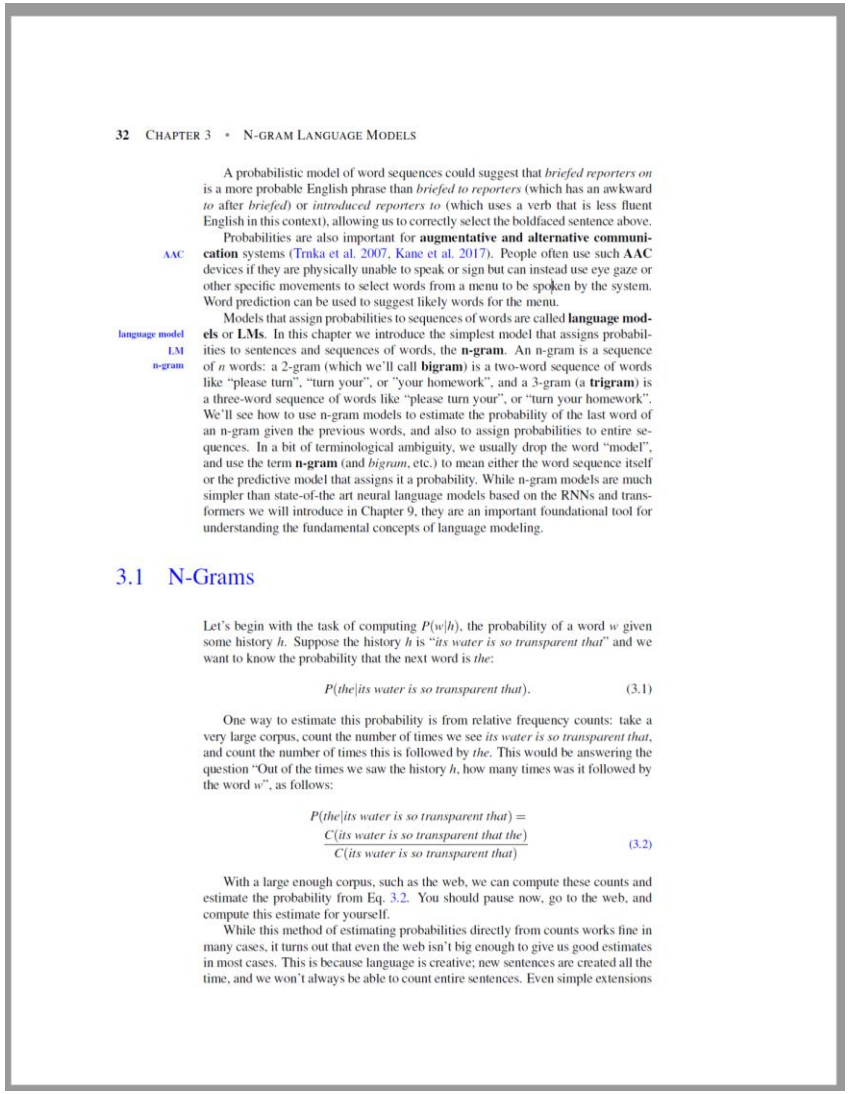
With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions

Document: book page, n ~ 3150, |Z| ~ 40
Search pattern: „N-gram“, m:= |„N-gram“| = 6

Free e-Book available at: https://web.stanford.edu/~jurafsky/slp3/

Real-World Example: What is an „N-gram“?



32 CHAPTER 3 • N-GRAM LANGUAGE MODELS

A probabilistic model of word sequences could suggest that *briefed reporters on* is a more probable English phrase than *briefed to reporters* (which has an awkward *to* after *briefed*) or *introduced reporters to* (which uses a verb that is less fluent English in this context), allowing us to correctly select the boldfaced sentence above.

AAC Probabilities are also important for **augmentative and alternative communication** systems (Trnka et al. 2007, Kane et al. 2017). People often use such **AAC** devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

language model, LM, n-gram Models that assign probabilities to sequences of words are called **language models** or **LMs**. In this chapter we introduce the simplest model that assigns probabilities to sentences and sequences of words, the **n-gram**. An n-gram is a sequence of n words: a 2-gram (which we'll call **bigram**) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (a **trigram**) is a three-word sequence of words like "please turn your", or "turn your homework". We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word "model", and use the term **n-gram** (and *bigram*, etc.) to mean either the word sequence itself or the predictive model that assigns it a probability. While n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers we will introduce in Chapter 9, they are an important foundational tool for understanding the fundamental concepts of language modeling.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

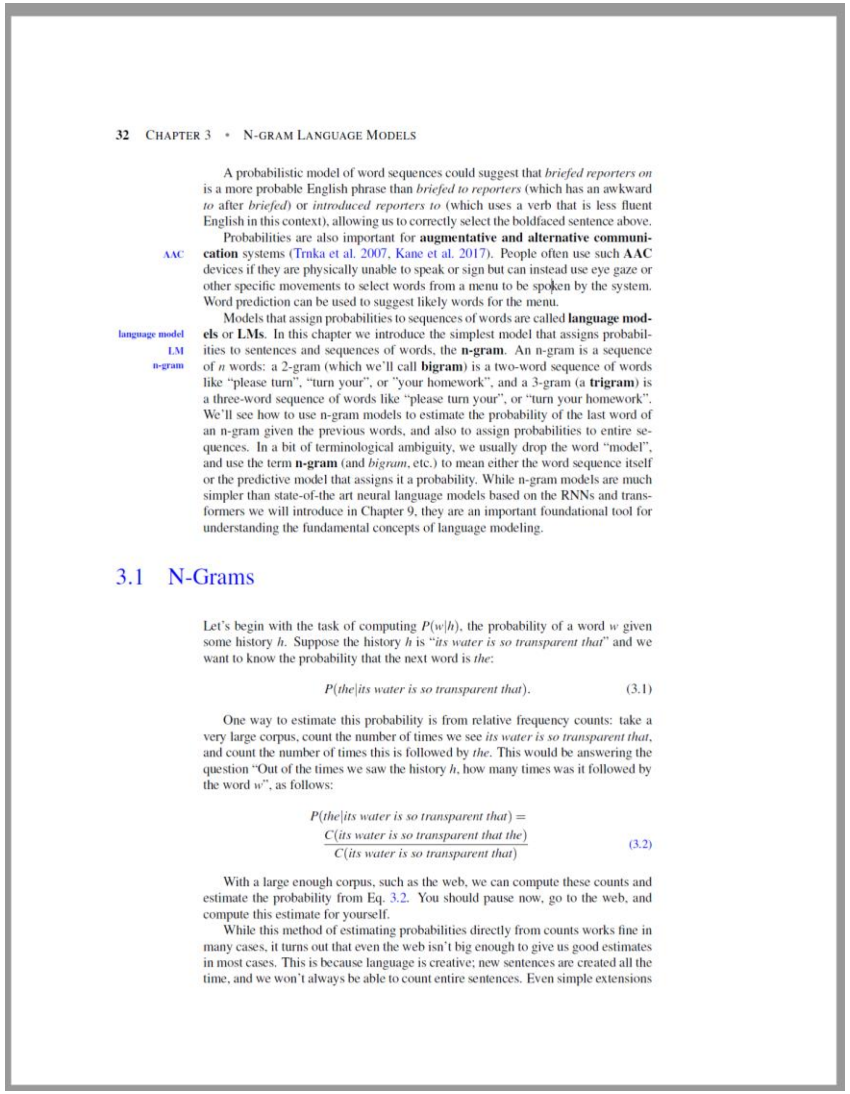
While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions

Algorithm	Preparation time	Runtime
Naive String Search	-	O(mn)
Finite Automaton	O(m|Z|)	O(n)
Suffix Tree	O(n)	O(m)

Document: book page, n ~ 3150, |Z| ~ 40
Search pattern: „N-gram“, m:= |„N-gram“| = 6

Free e-Book available at: https://web.stanford.edu/~jurafsky/slp3/

Real-World Example: What is an „N-gram“?



32 CHAPTER 3 • N-GRAM LANGUAGE MODELS

A probabilistic model of word sequences could suggest that *briefed reporters on* is a more probable English phrase than *briefed to reporters* (which has an awkward *to* after *briefed*) or *introduced reporters to* (which uses a verb that is less fluent English in this context), allowing us to correctly select the boldfaced sentence above.

AAC

Probabilities are also important for **augmentative and alternative communication** systems (Trnka et al. 2007, Kane et al. 2017). People often use such **AAC** devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

language model
LM
n-gram

Models that assign probabilities to sequences of words are called **language models** or **LMs**. In this chapter we introduce the simplest model that assigns probabilities to sentences and sequences of words, the **n-gram**. An n-gram is a sequence of n words: a 2-gram (which we'll call **bigram**) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (a **trigram**) is a three-word sequence of words like "please turn your", or "turn your homework". We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word "model", and use the term **n-gram** (and *bigram*, etc.) to mean either the word sequence itself or the predictive model that assigns it a probability. While n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers we will introduce in Chapter 9, they are an important foundational tool for understanding the fundamental concepts of language modeling.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(\mathit{the}|\mathit{its\ water\ is\ so\ transparent\ that}). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(\mathit{the}|\mathit{its\ water\ is\ so\ transparent\ that}) = \frac{C(\mathit{its\ water\ is\ so\ transparent\ that\ the})}{C(\mathit{its\ water\ is\ so\ transparent\ that})} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions

Algorithm	Preparation time	Runtime
Naive String Search	-	O(mn)
Finite Automaton	O(m|Z|)	O(n)
Suffix Tree	O(n)	O(m)

Document: book page, n ~ 3150, |Z| ~ 40

Search pattern: „N-gram“, m:= |„N-gram“| = 6

Algorithm	Preparation time	Runtime
Naive String Search	-	mn ~ 18900
Finite Automaton	m|Z| ~= 240	n ~= 3150
Suffix Tree	n ~= 3150	m ~= 6

Free e-Book available at: https://web.stanford.edu/~jurafsky/slp3/

String-Matching-Algorithm

Naive string-matching algorithm for finding text segments in a string based on a given search pattern.

TODO

- **Z = `abcdefghijklmnopqrstvwxyz` , |Z| = 25**
- **Example:**

  Document: „aaaabcbbabcbbb“, n=14

  Search Pattern: „abc“, m=3

  Result: „aaa***<u>abc</u>***bb***<u>abc</u>***bbb

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 1

a != c

Naive Algorithm

#comparisons = 1+2

aaaabcbbabcbbb

abc

```
aa != bc
```

Naive Algorithm

#comparisons = 3+3

aaaabcbbabcbbb

abc

```
aaa != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 6+3

```
aaa != abc
```

Naive Algorithm

#comparisons = 9+3

aaaabcbbabcbbb

abc

```
aab != abc
```

Naive Algorithm

#comparisons = 12+3

aaaabcbbabcbbb

abc

```
abc == abc
```

Naive Algorithm

#comparisons = 15+3

aaaabcbbabcbbb

abc

```
bcb != abc
```

Naive Algorithm

#comparisons = 18+3

aaaabcbbabcbbb

abc

```
cbb != abc
```

Naive Algorithm

#comparisons = 21+3

aaaabcbbabcbbb

abc

```
bba != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 24+3

```
bab != abc
```

Naive Algorithm

#comparisons = 27+3

aaaabcbbabcbbb

abc

```
abc == abc
```

Naive Algorithm

#comparisons = 30+3

aaaabcbbabcbbb

abc

```
bcb != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 33+3

```
cbb != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 36+3

```
bbb != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 39+2

```
bb != abc
```

Naive Algorithm

aaaabcbbabcbbb

abc

#comparisons = 41+1

```
b != abc
```

Naive Algorithm

aaaabcbbabcbbb

#comparisons = 41+1

?

How does the *#comparisons* change with
- search pattern to be searched?
- the document to be searched?

Naive Algorithm

aaaabcbbabcbbb

#comparisons = 41+1

?

How does the *#comparisons* change with
- search pattern to be searched?
- the document to be searched?

Runtime complexity:

$$\mathcal{O}(n \cdot m)$$

```
  n  = |search pattern| =  3
  m  = |document|       = 14
n*m  = 42 = #comparisons
```

Finite Automata and (Formal) Languages

"For formalizing the notion of a language one must cover all the varieties of languages such as natural (human) languages and programming languages. Let us look at some common features across the languages. One may broadly see that a language is a collection of sentences; a sentence is a sequence of words; and a word is a combination of syllables."

Quote from "Formal Languages and Automata Theory", D. Goswami and K. V. Krishna, 5 Nov 2010

Remarks on Finite Automata

- **Deterministic** vs. **non-deterministic** finite automata (DFA vs. NFA)
  - DFA: deterministic in the sense, that there exists exactly one transition from a state on an input symbol
  - NFA: may have zero or several possible transitions on a single input symbol from one state to another, so we can only predict a set of possible actions (NFA to DFA: https://en.wikipedia.org/wiki/Powerset_construction)

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Remarks on Finite Automata

- **Deterministic** vs. **non-deterministic** finite automata (DFA vs. NFA)
  - DFA: deterministic in the sense, that there exists exactly one transition from a state on an input symbol
  - NFA: may have zero or several possible transitions on a single input symbol from one state to another, so we can only predict a set of possible actions (NFA to DFA: https://en.wikipedia.org/wiki/Powerset_construction)

- **Accepter** vs. **Transducer**
  - An automaton whose output response is limited to a simple “yes” or “no” is called an accepter. Presented with an input string, an accepter either accepts the string or rejects it.
  - A more general automaton, capable of producing strings of symbols as output, is called a transducer.

Note: On the following slides we only consider finite deterministic accepters.

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of **internal** states,

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

$\delta : Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**,

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

$\delta : Q \times \Sigma \to Q$ is a total function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of **internal** states,

Σ is a finite set of symbols called the **input alphabet**,

$\delta : Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states.**

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Formal Languages

A ***string (word/sentence)*** over an alphabet $\Sigma \neq \emptyset$ is

- a finite sequence of symbols of Σ
- denoted by $\mathrm{a}_1\mathrm{a}_2 \dots \mathrm{a}_\mathrm{n}$ $(\mathrm{a}_\mathrm{i} \in \Sigma)$.

Formal Languages

A ***string (word/sentence)*** over an alphabet $\Sigma \neq \emptyset$ is

- a finite sequence of symbols of Σ
- denoted by $\mathrm{a_1 a_2 \ldots a_n}$ $(\mathrm{a_i} \in \Sigma)$.
- Empty string denoted by λ *satisfies* $\forall x \in \Sigma^*\colon\ \lambda x = x\lambda = x$

Formal Languages

A ***string (word/sentence)*** over an alphabet $\Sigma \neq \emptyset$ is

- a finite sequence of symbols of Σ
- denoted by $\mathrm{a_1 a_2 \dots a_n}$ $(\mathrm{a_i} \in \Sigma)$.
- Empty string denoted by λ *satisfies* $\forall x \in \Sigma^*\colon \lambda x = x\lambda = x$

„Kleene Star“ closure

$$L^* := \bigcup_{i \in \mathbb{N}_0} L^i$$

Formal Languages

A ***string (word/sentence)*** over an alphabet $\Sigma \neq \emptyset$ is

- a finite sequence of symbols of Σ
- denoted by $\mathrm{a}_1\mathrm{a}_2 \ldots \mathrm{a}_\mathrm{n}$ $(\mathrm{a}_\mathrm{i} \in \Sigma)$.
- Empty string denoted by λ *satisfies* $\forall x \in \Sigma^*\colon \lambda x = x\lambda = x$

„Kleene Star" closure

$$L^* := \bigcup_{i \in \mathbb{N}_0} L^i$$

Formal Language *⇔ Any collection of strings over an alphabet* Σ

$L^+ := \bigcup_{i \in \mathbb{N}} L^i$ $\quad L_0 = \{\lambda\},$ $\quad k \geq 1\colon L^k = L^{k-1}L,$ $\quad \bar{L} = \Sigma^* - L,$ $\quad L_1L_2 = \{xy : x \in L_1, y \in L_2\}$

Definition: Extended Transition function δ^*

Function $\delta^*: Q \times \Sigma^* \rightarrow Q$ recursively defined by

$$\boxed{\begin{gathered}\delta^*(q, \lambda) = q, \\ \delta^*(q, wa) = \delta(\delta^*(q, w), a)\end{gathered}}$$

for all $q \in Q, w \in \Sigma^*, a \in \Sigma$.

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Definition: Extended Transition function $\boldsymbol{\delta^*}$

Empty string of length $|\boldsymbol{\lambda}| = \mathbf{0}$

Function $\delta^*: Q \times \Sigma^* \to Q$ recursively defined by

$$\boxed{\begin{gathered}\delta^*(q, \boxed{\lambda}) = q, \\ \delta^*(q, wa) = \delta(\delta^*(q, w), a)\end{gathered}}$$

Concatenation of w (= prefix) and a (= suffix)

for all $q \in Q, w \in \Sigma^*, a \in \Sigma$.

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Definition: Extended Transition function $\boldsymbol{\delta^*}$

Empty string of length $|\boldsymbol{\lambda}| = \mathbf{0}$

Function $\delta^*: Q \times \Sigma^* \rightarrow Q$ recursively defined by

$$\delta^*(q, \boxed{\lambda}) = q,$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

for all $q \in Q, w \in \Sigma^*, a \in \Sigma$.

Concatenation of w (= prefix) and a (= suffix)

***Question**: is the concatenation operation commutative on Σ^* ?*

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Relation between Accepted Language and DFA

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and δ^* as defined before.

- A **string** $w \in \Sigma^*$ is said to be **accepted** by a DFA $M :\Leftrightarrow \delta^*(q_0, w) \in F$

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Relation between Accepted Language and DFA

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and δ^* as defined before.

- A **string** $w \in \Sigma^*$ is said to be **accepted** by a DFA $M :\Leftrightarrow \delta^*(q_0, w) \in F$
- The **language accepted** by M is the set of all strings on Σ accepted by M :

$$L(M) := \{w \in \Sigma^*: \delta^*(q_0, w) \in F\}.$$

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

DFA: Transition Graph or State Transition Diagram

Let the DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Can be represented as Transition Graph G_M with

- **Vertices**
- **Edges**

References: An introduction to formal languages and automata / Peter Linz. – 5th ed.

DFA: Transition Graph or State Transition Diagram

Let the DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Can be represented as Transition Graph G_M with

- **Vertices**
  - How many?

References: An introduction to formal languages and automata / Peter Linz. – 5th ed.

DFA: Transition Graph or State Transition Diagram

Let the DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Can be represented as Transition Graph G_M with

- **Vertices**
  - How many? G_M has exactly $|Q|$ vertices
  - Initial vertex: q_0
  - Final vertices: $q_f \in F$

References: An introduction to formal languages and automata / Peter Linz. – 5th ed.

DFA: Transition Graph or State Transition Diagram

Let the DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Can be represented as Transition Graph G_M with

- **Vertices**
  - How many? G_M has exactly $|Q|$ vertices
  - Initial vertex: q_0
  - Final vertices: $q_f \in F$
- **Edges**
  - Edge (q_i, q_j) with label a ⇔ transition rule $\delta(q_i, a) = q_j$

References: An introduction to formal languages and automata / Peter Linz. – 5th ed.

DFA: Transition Graph or State Transition Diagram

Let the DFA be $M = (Q, \Sigma, \delta, q_0, F)$. Can be represented as Transition Graph G_M with

- **Vertices**
  - How many? G_M has exactly $|Q|$ vertices
  - Initial vertex: q_0
  - Final vertices: $q_f \in F$
- **Edges**
  - Edge (q_i, q_j) with label a ⇔ transition rule $\delta(q_i, a) = q_j$
  - There is an arrow with no source into initial state q_0
  - multiple arcs from one state to another (one for different alphabet symbols $a_1, \dots, a_k \in \Sigma$) draw one arc labeled $a_1, \dots, a_k$

References: An introduction to formal languages and automata / Peter Linz. – 5th ed.

(Formal) Language Representation

$$\Sigma = \{a, b\}$$

$$L = \{a^n b : n \geq 0\}.$$

Exercises:

- Which words are elements of the language?
- How does a related DFA look like?

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

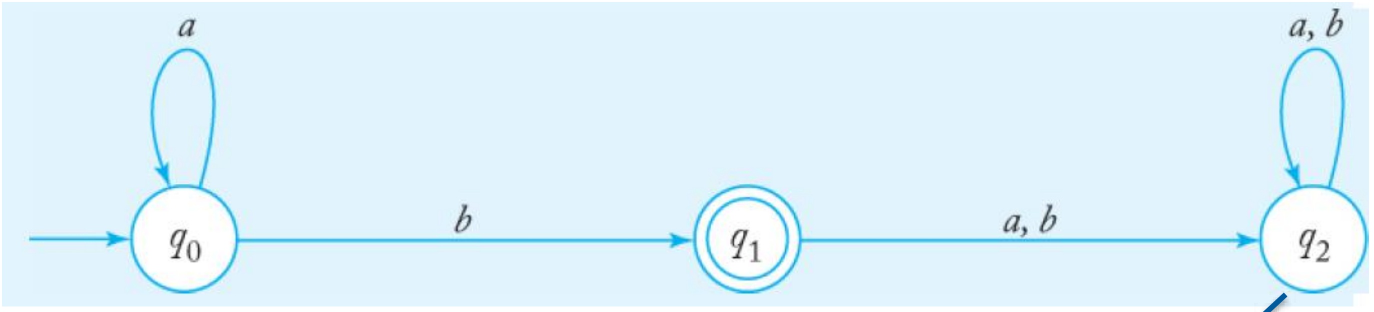
(Formal) Language Representation

$$\Sigma = \{a, b\}$$

$$L = \{a^n b : n \geq 0\}.$$

Exercises:
- Which words are elements of the language?
- How does a related DFA look like?

Transition Graph



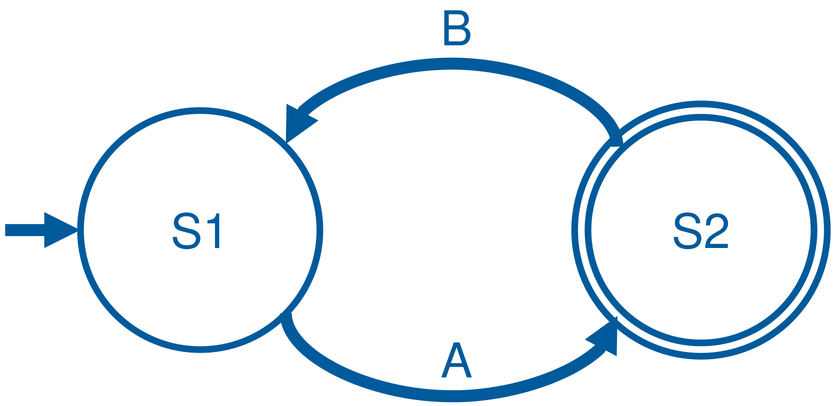
Trap state

Transition or *Next State* Table

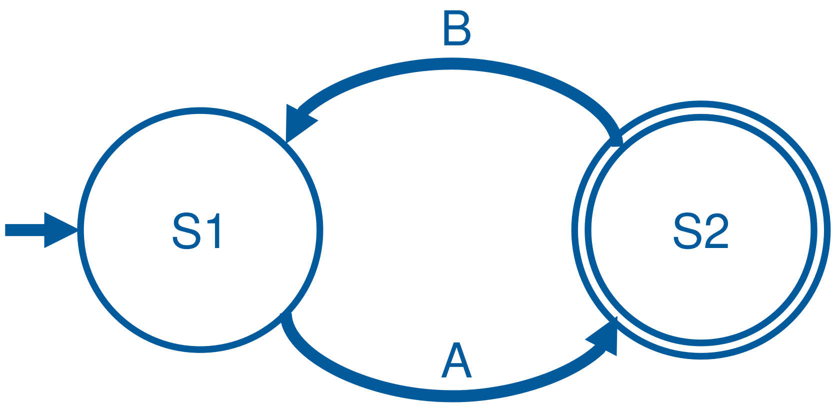
	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Reference: An introduction to formal languages and automata / Peter Linz. – 5th ed.

Exercise: Which language is accepted?



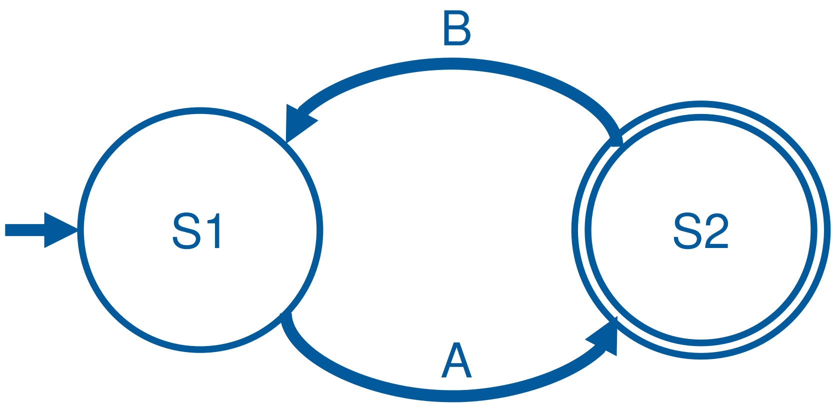
.. and formally described?



Accepted Language:

A
ABA
ABABA
ABABABA
ABABABABA
…

Finite Automaton ⇔ Formal Language



Accepted Language:

A
ABA
ABABA
ABABABA
ABABABABA
…
A(BA)*

(.)* := „star“ closure $L^* := \bigcup_{i \in \mathbb{N}_0} L^i$

(.)+ := „positive“ closure $L^+ := \bigcup_{i \in \mathbb{N}} L^i$

. := arbitrary symbol from alphabet

Finite Automaton

Document: n = „aaaabcbbabcbbb“, |n| = 14

Search pattern: m = „abc“, |m| = 3

Input Alphabet: Z = {a,b,c}, |Z| = 3

How does the automaton look like?

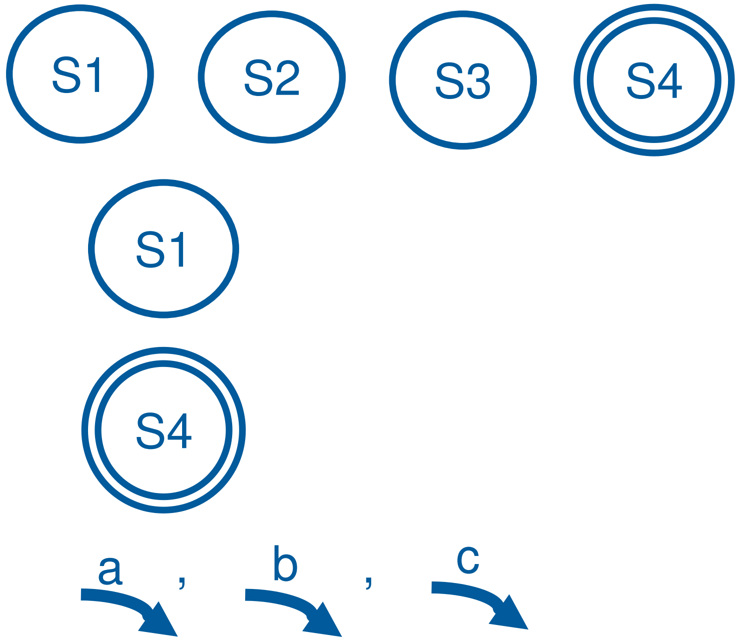
Finite Automaton

States:

Start-State:

End-State:

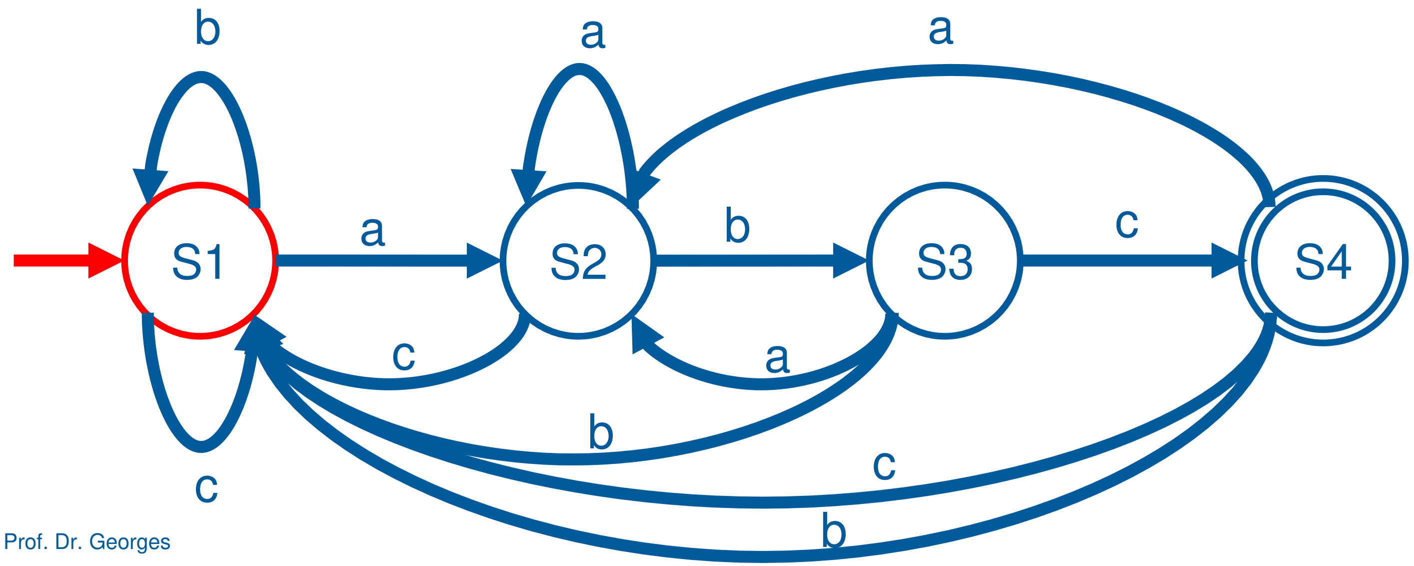
Transition:



Finite Automaton

aaaabcbbabcbbb

Search pattern: „abc“

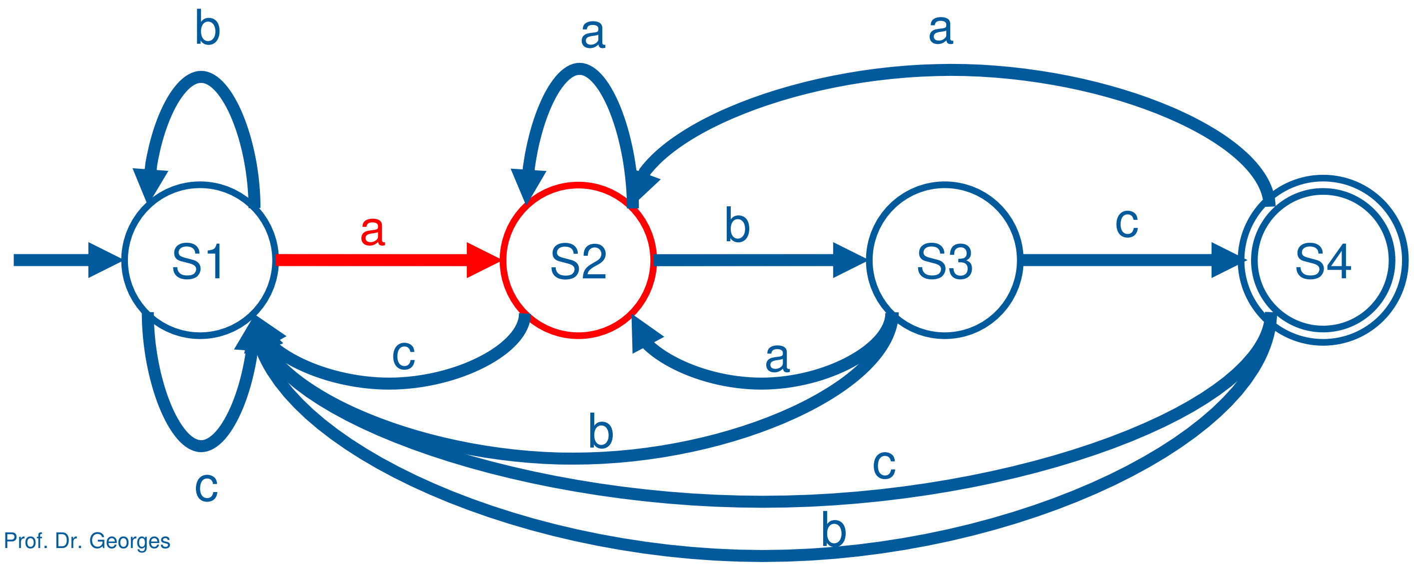


Search with Finite Automaton

#comparisons = 1

aaaabcbbabcbbb

abc

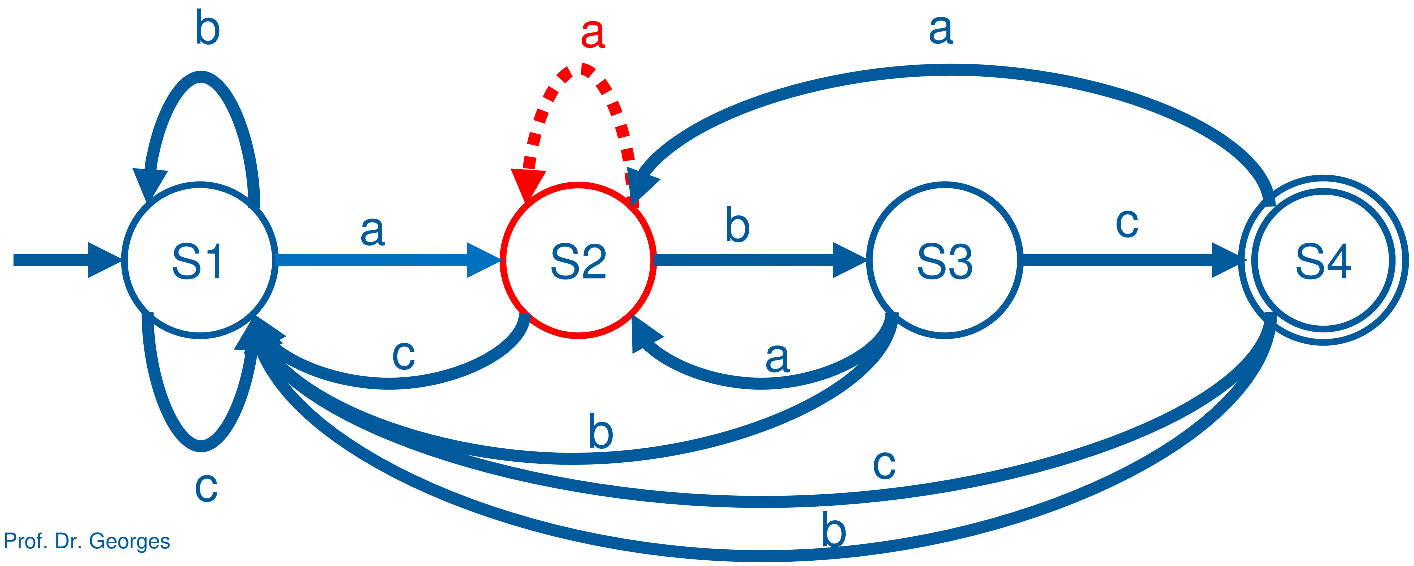


Search with Finite Automaton

#comparisons = 1+1

aaaabcbbabcbbb

abc

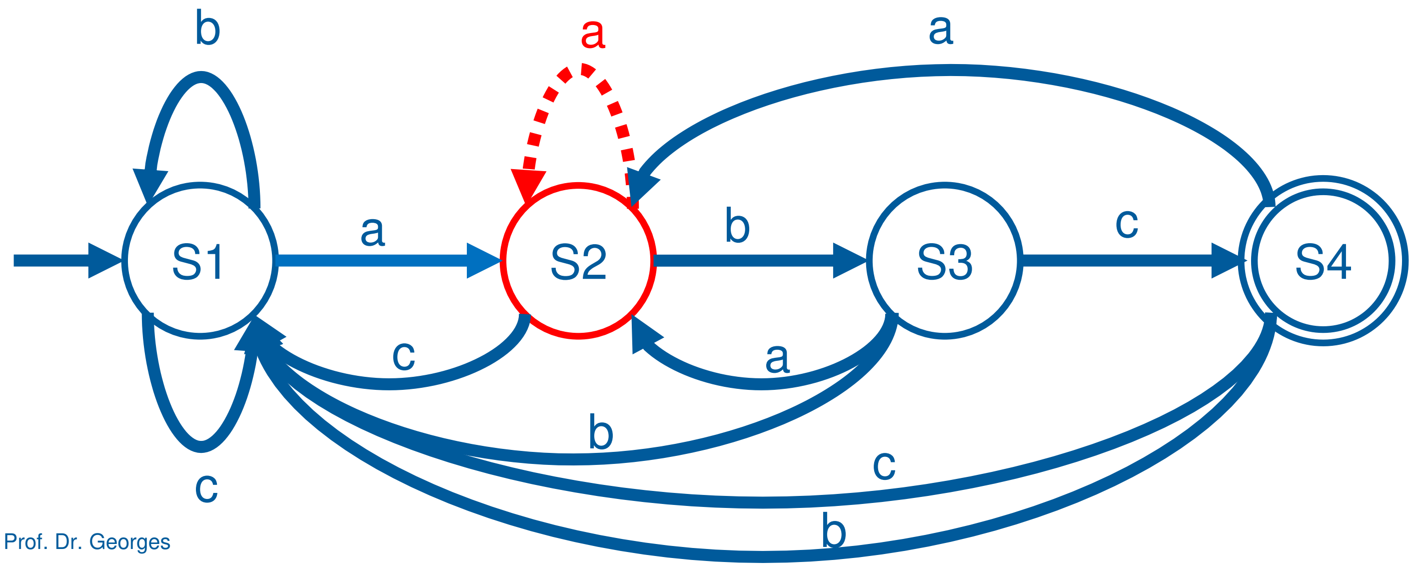


Search with Finite Automaton

#comparisons = 2+1

aaaabcbbabcbbb

abc

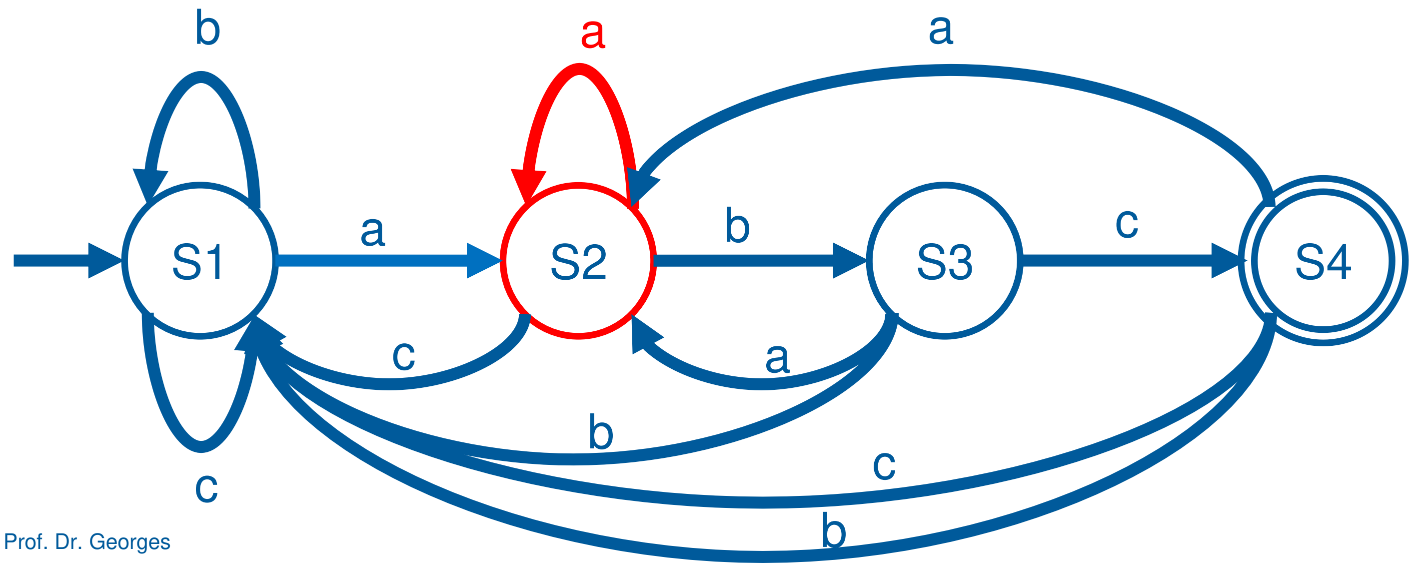


Search with Finite Automaton

#comparisons = 3+1

aaaabcbbabcbbb

abc

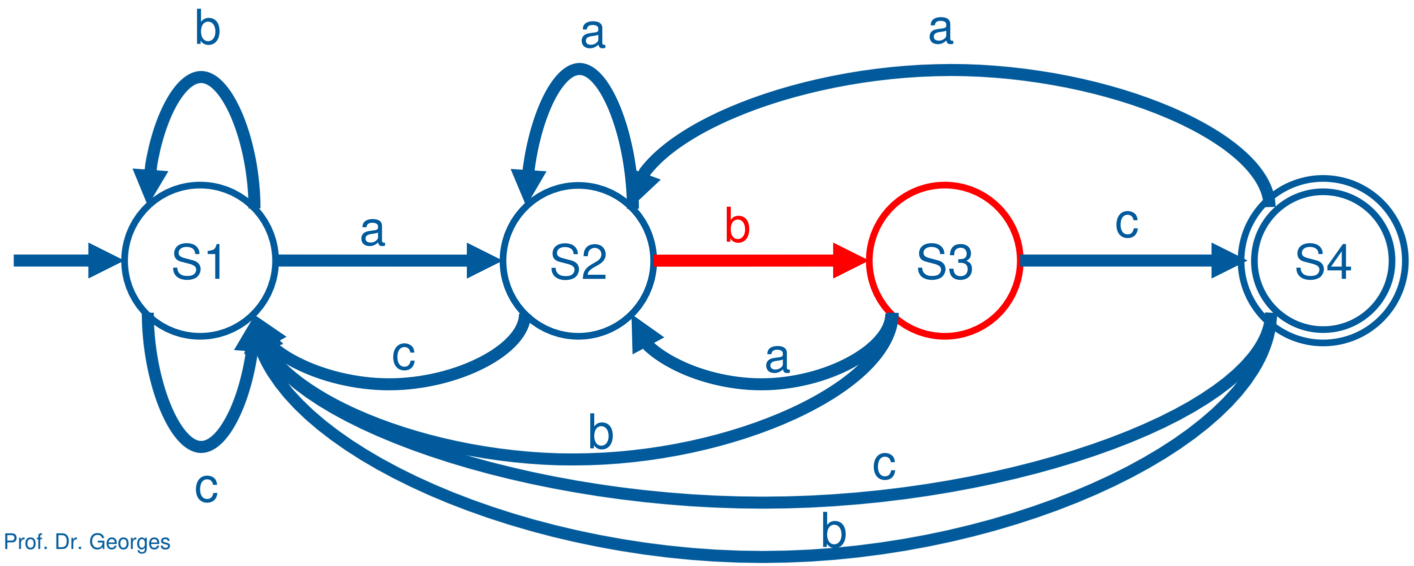


Search with Finite Automaton

#comparisons = 4+1

aaaabcbbabcbbb

abc

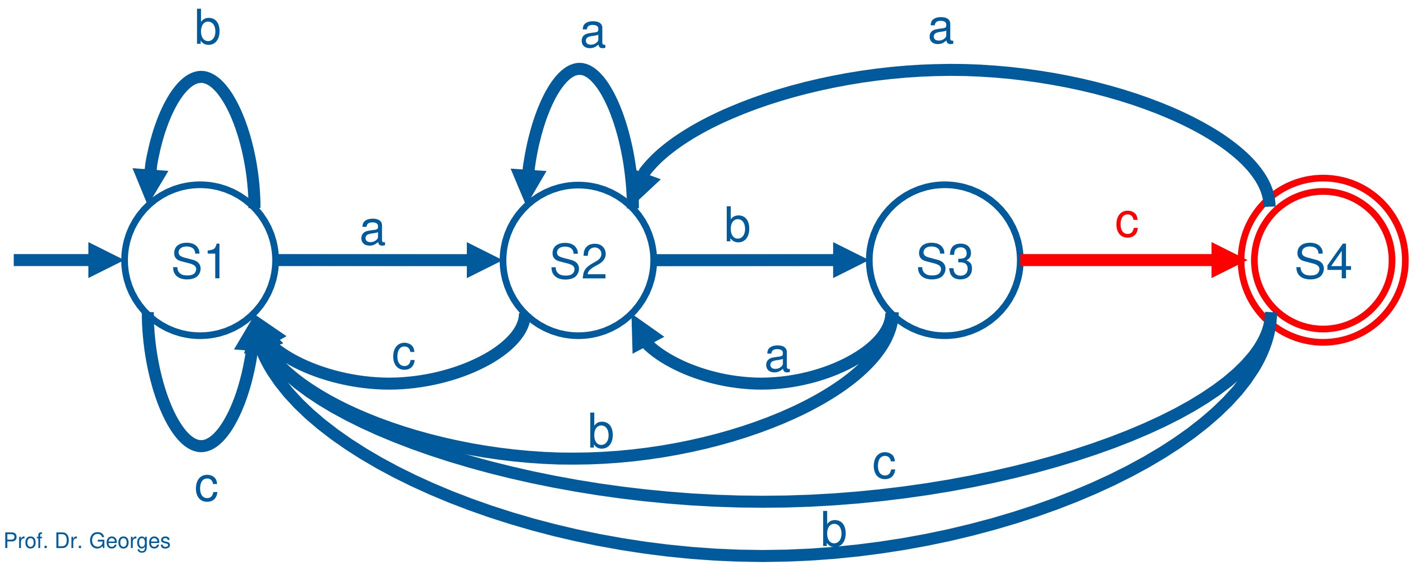


Search with Finite Automaton

#comparisons = 5+1

aaaabcbbabcbbb

abc

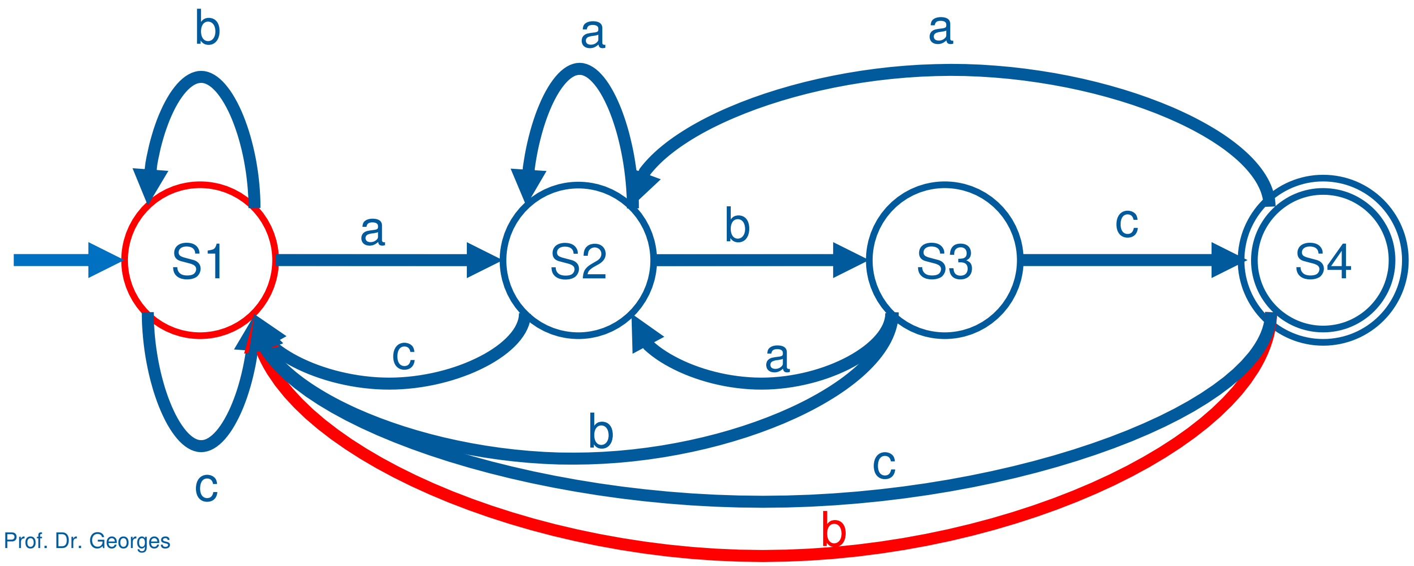


Search with Finite Automaton

#comparisons = 6+1

aaaabcbbabcbbb

abc

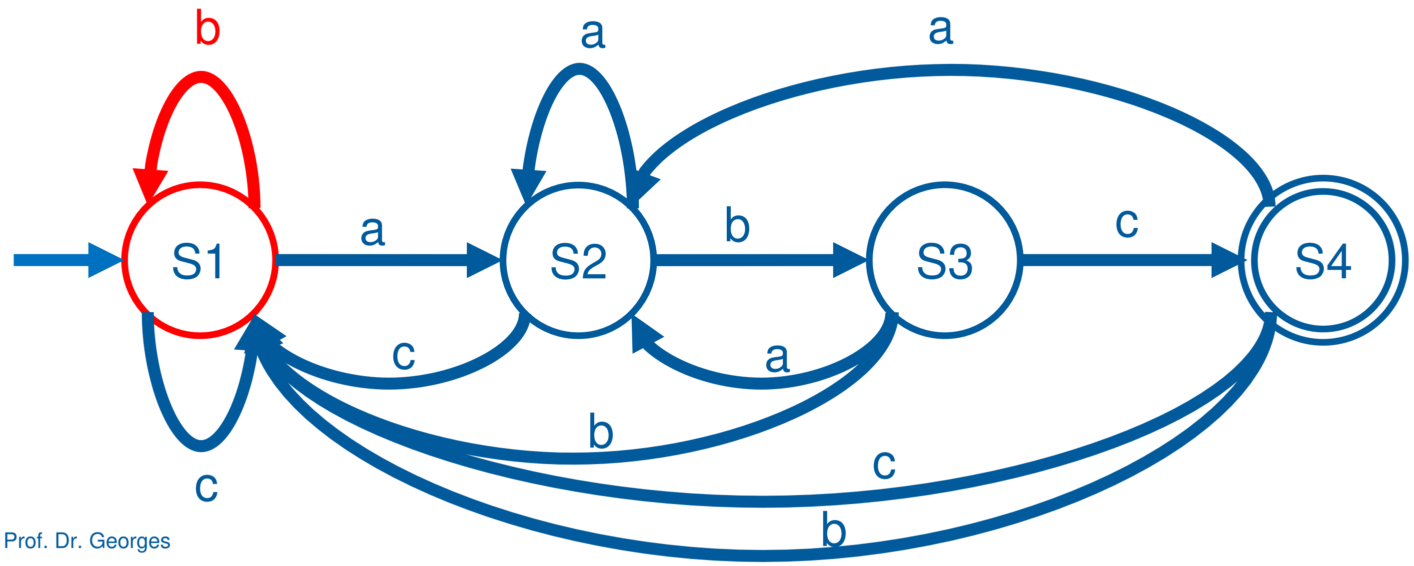


Search with Finite Automaton

#comparisons = 7+1

aaaabcbbabcbbb

abc

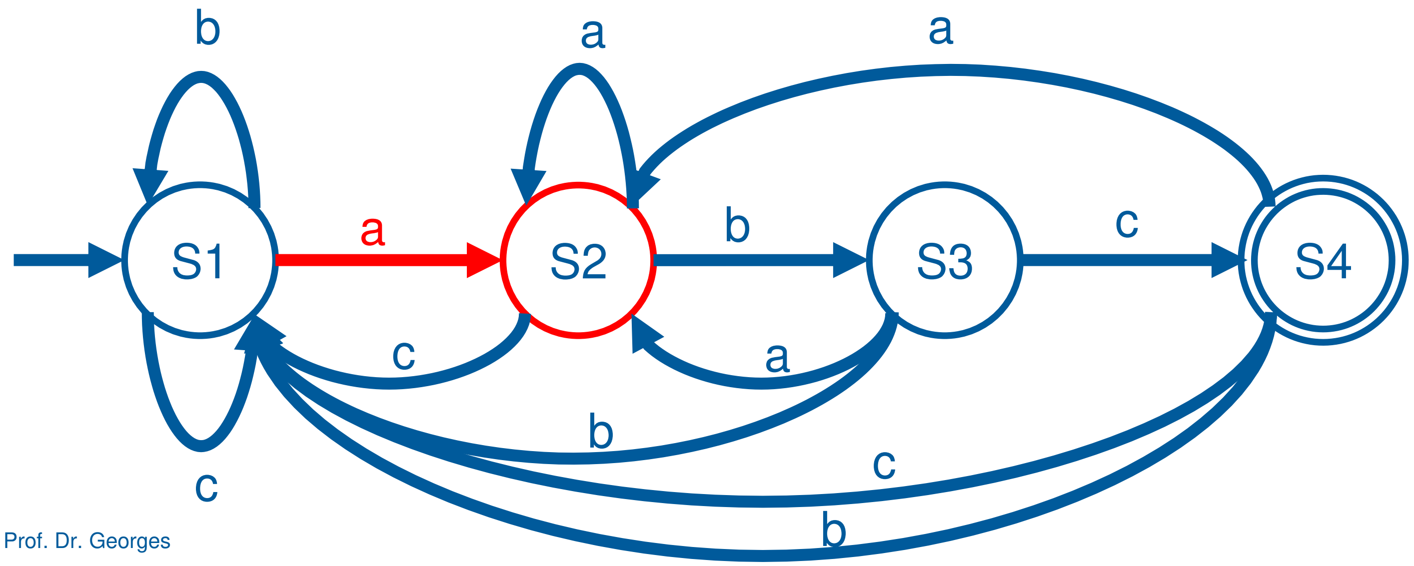


Suche mit Endlichem Automaten

#comparisons = 8+1

aaaabcbbabcbbb

abc

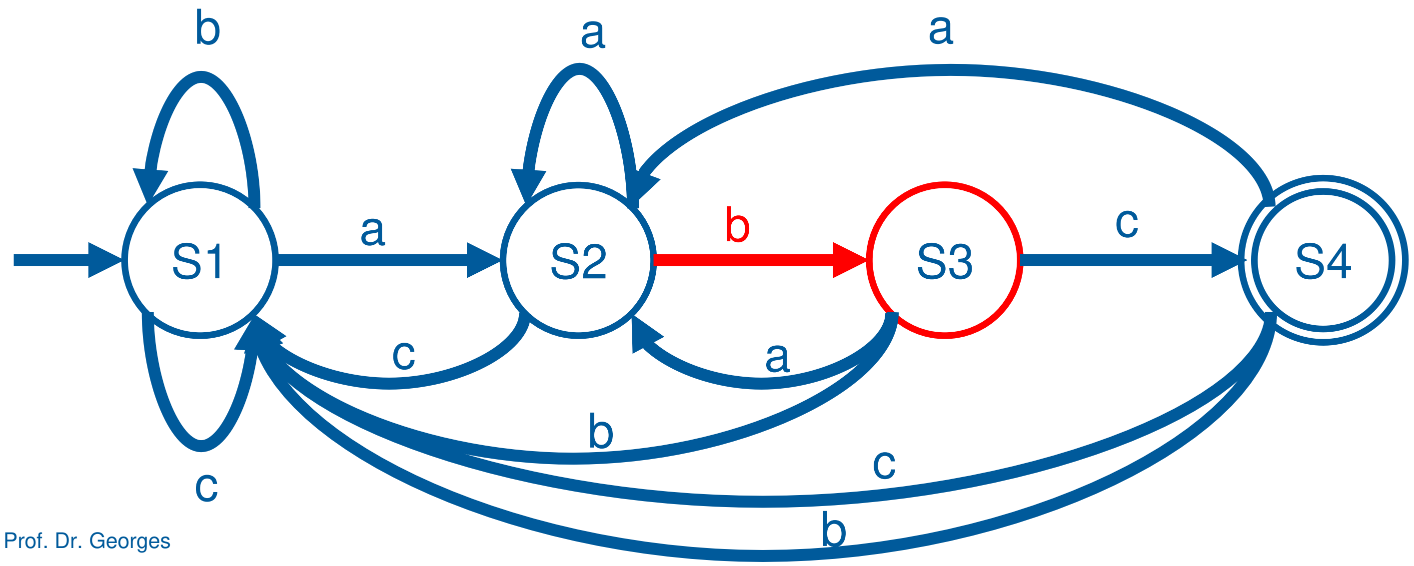


Search with Finite Automaton

#comparisons = 9+1

aaaabcbbabcbbb

abc

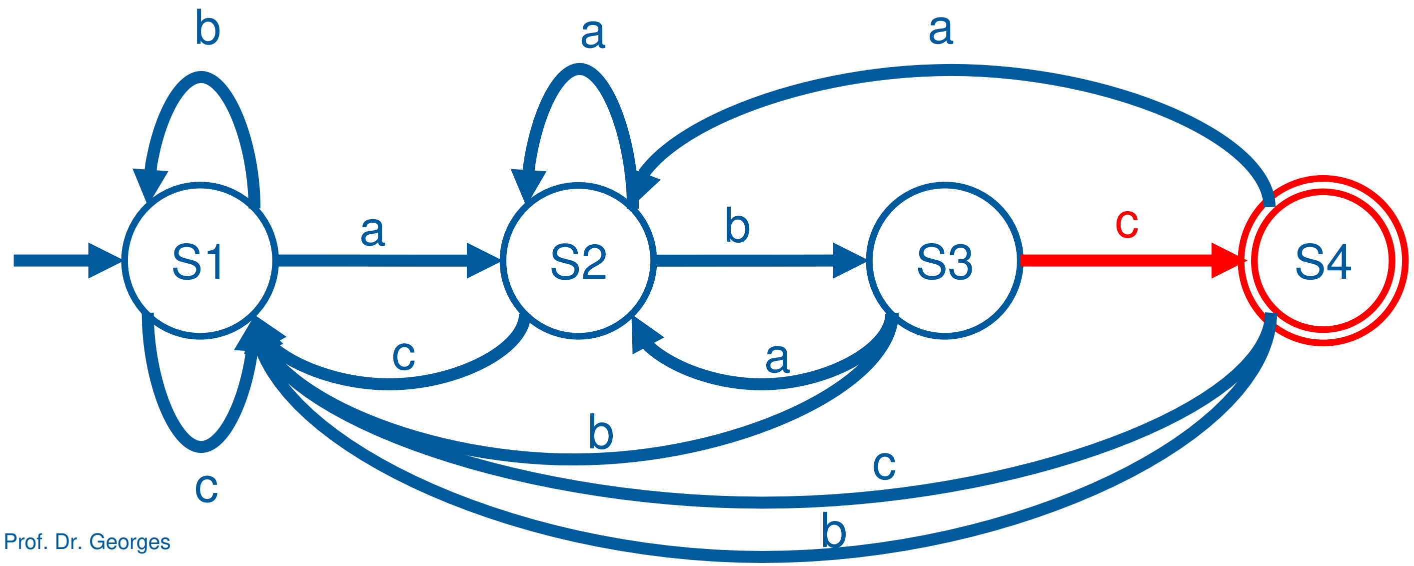


Search with Finite Automaton

#comparisons = 10+1

aaaabcbbabcbbb

abc

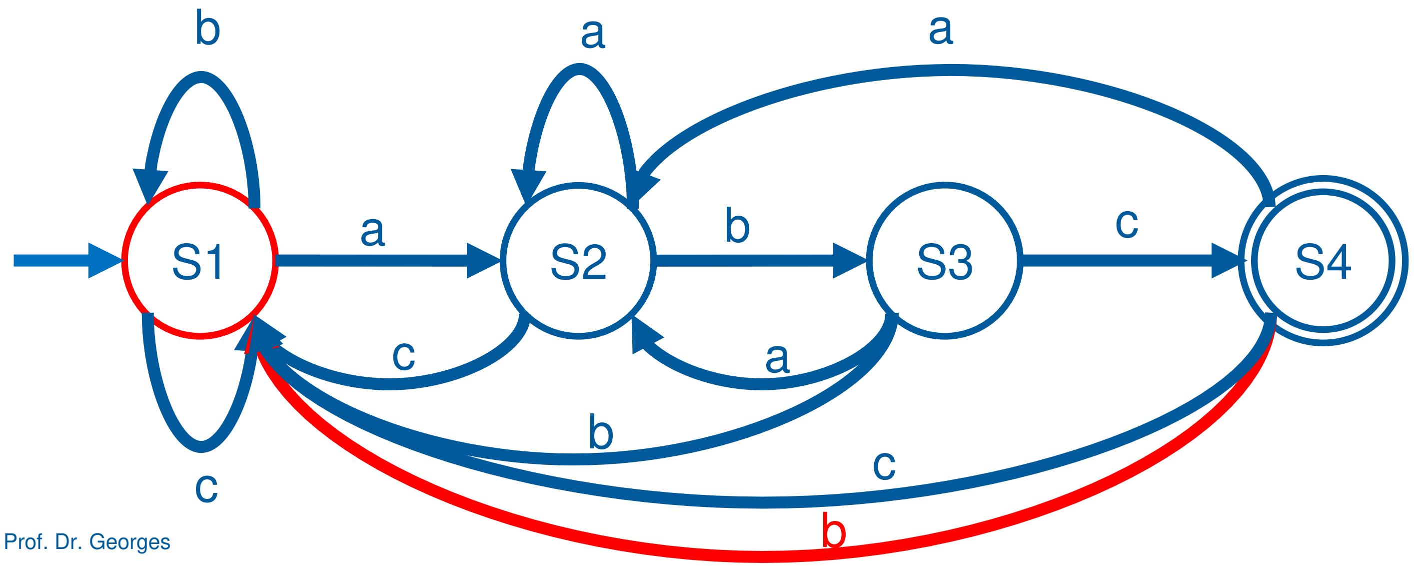


Search with Finite Automaton

#comparisons = 11+1

aaaabcbbabcbbb

abc

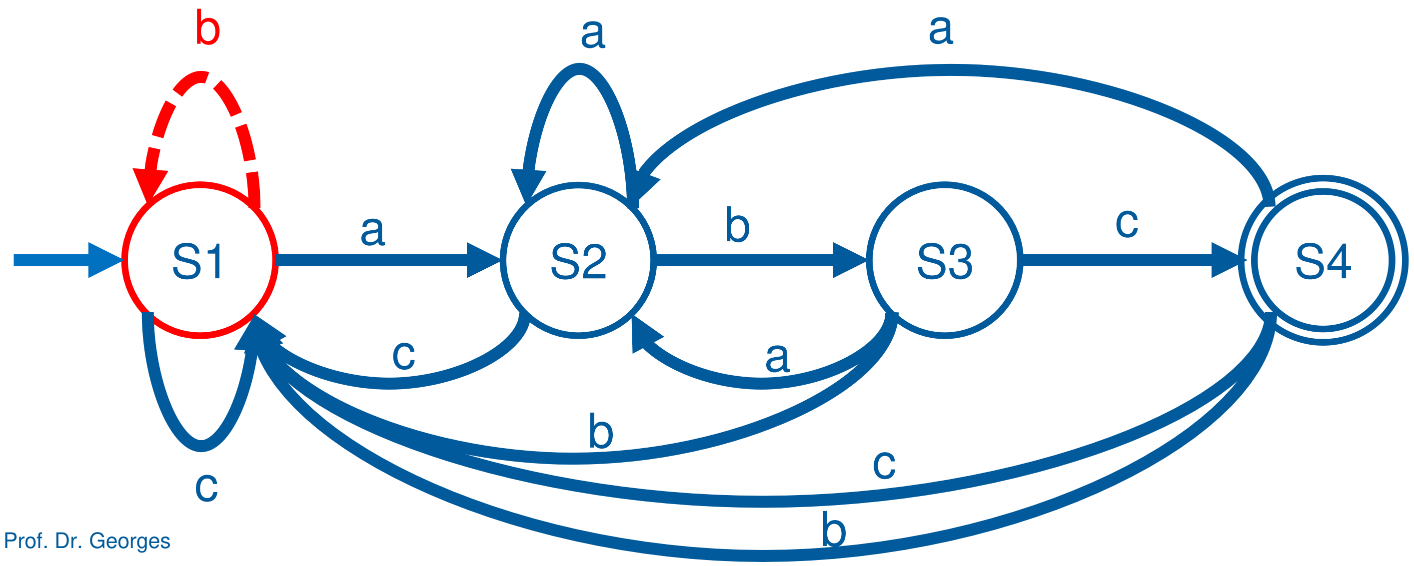


Search with Finite Automaton

#comparisons = 12+1

aaaabcbbabcbbb

abc

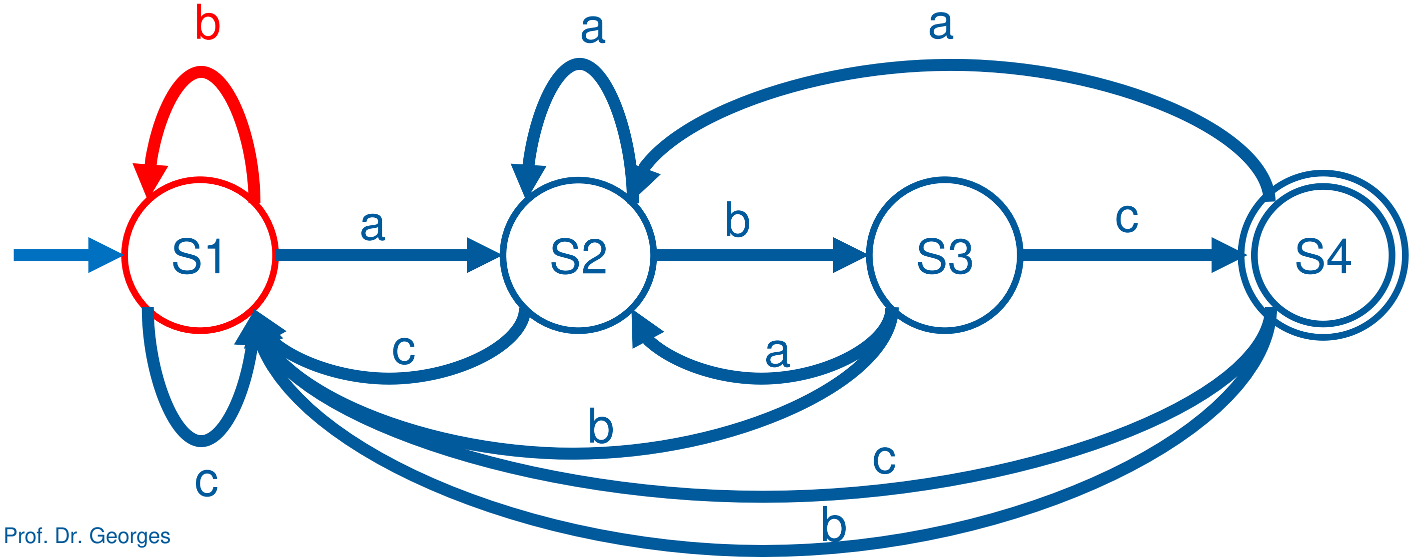


Search with Finite Automaton

#comparisons = 13

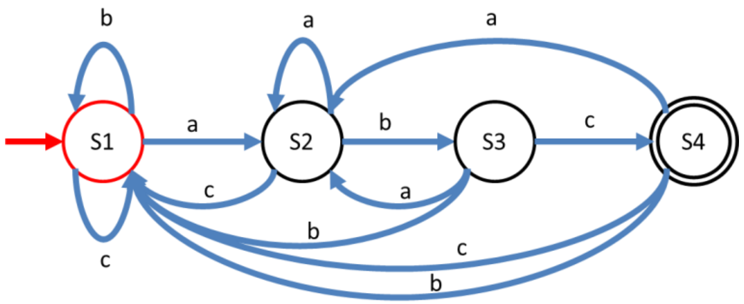
aaaabcbbabcbbb

abc



Search with Finite Automaton

aaa<u>*abc*</u>bb<u>*abc*</u>bbb



Runtime complexity:

$$\mathcal{O}(n)$$

Preparation complexity:

$$\mathcal{O}(mw)$$

#comparisons = 14

?

How does the *#comparisons* change with

- search pattern to be searched?
- the document to be searched?

```
m = |search pattern| =  3
n = |document|       = 14
w :=|Z|              = 3
```

Definition: Regular Expressions

… over an alphabet Σ are recursively defined as follows

1. $\emptyset, \varepsilon$, and a, for each $a \in \Sigma$, are regular expressions representing the languages $\emptyset, \{\varepsilon\}$, and $\{a\}$, respectively.

2. If r and s are regular expressions representing the languages R and S, respectively, then so are

   (a) $(r + s)$ representing the language $R \cup S$,

   (b) (rs) representing the language RS, and

   (c) (r^*) representing the language R^*.

Note:

We keep a minimum number of parentheses which are required to avoid ambiguity in the regular expression,

e.g. 1. r + st ⇔ (r + (st))

2. r + s + t ⇔ ((r + s) + t)

Reference: "Formal Languages and Automata Theory", D. Goswami and K. V. Krishna, 05/11/2010

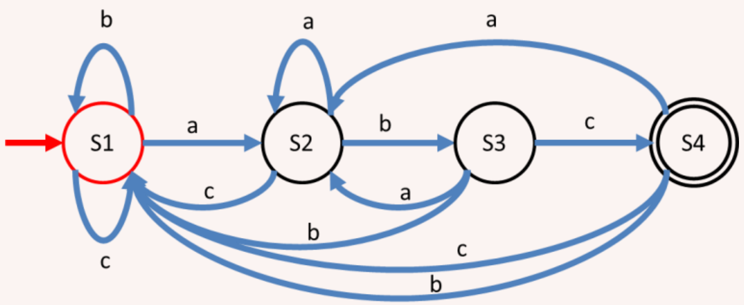
- If r is a regular expression, then the language represented by r is denoted by $L(r)$.
- Further, a **language** L is said to be ***regular*** if there is a regular expression r such that $L = L(r)$.

- A regular expressions r is said to be equivalent to a finite automaton A , if the language represented by r is precisely accepted by the finite automaton A , i. e. $L(r) = L(A)$.

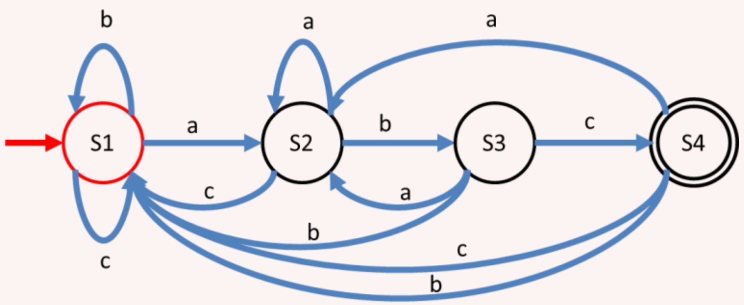
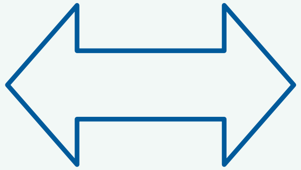
Theorem: (Equivalence of Regular Languages and Finite Automata)

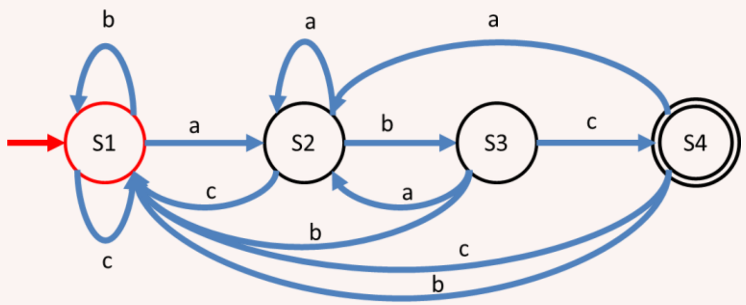
The language denoted by a regular expression can be accepted by a finite automaton.

Reference: “Formal Languages and Automata Theory”, D. Goswami and K. V. Krishna, 05/11/2010

Automaton	Language	
	?	

Regular Expressions

Automaton	Language	Formal/regular expression
	 abc aabc aaabcc abcabc aabcababc …	?

Automaton	Language	Formal/regular expression
	abc aabc aaabcc abcabc aabcababc …	(a|b|c)* inclusive a^+bc^+ ? ?

What are the algorithms?

- **Z = `abcdefghijklmnopqrstvwxyz` , |Z| = 25**
- **Example:**

  Document: „aaaabcbbabcbbb“, | „aaaabcbbabcbbb“ | = n = 14

  Search pattern: „abc“, |„abc“| = m = 3

  Result: „aaa***abc***bb***abc***bbb

Algorithm	Preparation time	Runtime
Naive String Search	-	O(mn)
Finite Automaton	O(m|Z|)	O(n)
Suffix Tree	O(n)	O(m)

What are the algorithms?

- **Z = `abcdefghijklmnopqrstvwxyz` , |Z| = 25**
- **Example:**

  Document: „aaaabcbbabcbbb“, | „aaaabcbbabcbbb“ | = n = 14

  Search pattern: „abc“, |„abc“| = m = 3

  Result: „aaa***abc***bb***abc***bbb

Algorithm	Preparation time	Runtime
Naive String Search	-	O(mn)
Finite Automaton	O(m|Z|)	O(n)
Suffix Tree	O(n)	O(m)

Note: $f(x) = O\big(g(x)\big) \;\Leftrightarrow\; \exists N, C \in \mathbb{R}\ \forall x > N\colon\ |f(x)| \leq C \cdot |g(x)|$ (see e.g. MIT 16.070 slides on Big-O-Notation)