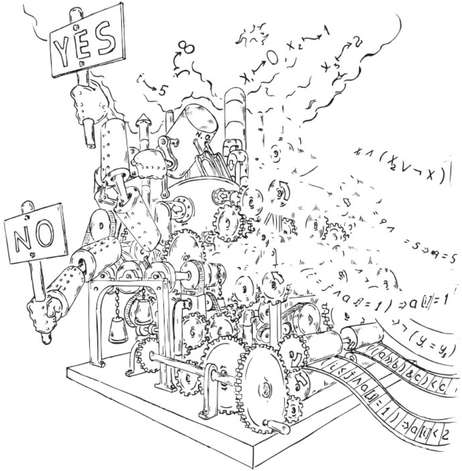
Automata, Languages and Computation

Chapter 1 : Automata Theory and Proof Techniques

Master Degree in Computer Engineering
University of Padua
Lecturer : Giorgio Satta

Lecture based on material originally developed by :
Gösta Grahne, Concordia University

Theoretical computer science



Introduction

One of the main goals of theoretical computer science is the mathematical study of **computation**

- computability : what can be computed ?
- tractability : what can be **efficiently** computed ?

The mathematical study of computation requires

- abstract models of machine : **automata theory**
- abstract representations of data : **formal language theory**

Introduction

Most well-known models of computation :

- Turing machines, introduced for the study of computability
- finite automata, introduced as models of neuronal calculus
- formal grammars, introduced by Noam Chomsky as linguistic models

1. Introduction to finite automata : pervasive model using a fixed amount of memory

2. Formal proof techniques : hypothesis, thesis, deduction, induction

3. Basic concepts of automata theory : alphabets, strings and languages

Finite automata

Finite automata, or FA for short : Finite set of **states** with **transitions** from one state to another

Used as a model for :

- software for digital circuit design
- lexical analyzer within a compiler
- keyword search in a file or on the web
- communication protocols

We will see more later on applications

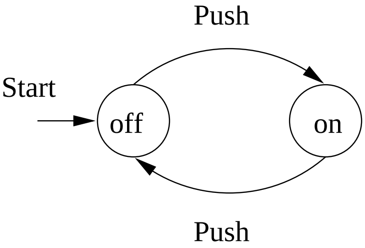
Finite automata

The simplest representation for an FA is a **graph** :

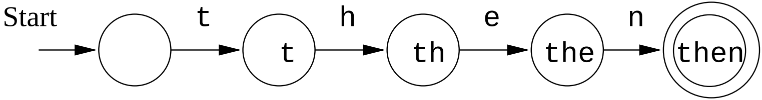
- **nodes** represent states
- **arcs** represent transitions
- **labels** on each arc indicate what is causing the transition

Example

FA for on/off **switch**



FA that **recognizes** the keyword `then` in a programming language



Structural Representation

An FA is a **recognition** model : it takes as input a sequence (string) and either accepts or rejects

Alternatively, we can use a **generative** model : such model generates all of the **desired** sequences (no input)

Recognition models are operational, generative models are declarative

Structural Representation

Grammars : A rewriting rule

$$E \rightarrow E + E$$

specifies that an arithmetic expression may consist of two arithmetic expressions combined by the addition operator

Regular expressions : The expression

`[A-Z][a-z]*[ ][A-Z][A-Z].`

generates the string `Ithaca NY`, but does not generate the string `Palo Alto CA`

Generative models unveil structure underlying data

Deductive proof

Typical form of the statement to be proved (H, C properties) :

If H, then C

also written as $H \Rightarrow C$, where $H =$ **hypothesis**, $C =$ **conclusion**

This means

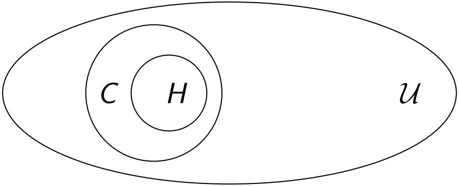
- H is a **sufficient** condition for C
- C is a **necessary** condition for H

See insiemistic interpretation in next slide

Deductive proof

In an **insiemistic** interpretation, H and C are associated with all the elements of the universe $\mathcal{U}$ that have that property

$H \Rightarrow C$ is equivalent to $H \subseteq C$: if H is true, C can't be false



Many students at the final exam use $H \Rightarrow C$ and $C \Rightarrow H$ interchangeably: **don't do that**!

Deductive proof

Deduction : Sequence of statements that starts from one or more hypotheses and leads to a conclusion

Each step of the deduction uses some **logical rule**, applying it to the hypotheses or to one of the previously obtained statements

Modus ponens : logical rule to move from one statement to the next. If we know that "if H then C" is true, and if we know that H is true, then we can conclude that C is true

Example

Theorem If x is the sum of the squares of four positive integers, then $2^x \geqslant x^2$

x is a **parameter** and is universally quantified; the theorem is valid for all x's that satisfy the hypotheses

See textbook for example of a deductive proof

Deductive proof

Theorems having the form

C_1 *if and only if* C_2

require proofs for both **directions** :

- "if C_2 then C_1"
- "if C_1 then C_2", which is equivalent to "C_1 only if C_2"

Additional techniques

Reduction to definitions : Convert all terms in the assumptions using the corresponding definitions

Proof by contradiction : To prove "if H then C", prove "H and not C implies falsehood"

Example

Theorem Let S be a finite subset of an infinite set U. Let T be the complement set of S with respect to U. Then T is infinite

Proof S is finite, by definition, is equivalent to :
there is an integer n such that $|S| = n$

U is infinite, by definition, is equivalent to :
for no integer n we have $|U| = n$

T is the complement set of S, by definition, is equivalent to :
$S \cup T = U$ and $S \cap T = \emptyset$

Example

Let us consider the denial of the conclusion : "T is a finite set" (proof by contradiction)

T is finite, by definition, is equivalent to :
there is an integer m such that $|T| = m$

Using $|S| = n$ and using both $S \cup T = U$ and $S \cap T = \emptyset$, we have that $|U| = |S| + |T| = n + m$, that is, U is finite. But this is against out hypothesis $\square$

Additional techniques

Counterexample : to prove that a theorem is false it is enough to show a case in which the statement is false

Example :
Is it true that if x is a prime number, then x is odd ?
No, in fact 2 is a prime number but it is not odd

Quantifiers

For each x ($\forall x$) : applies to all values of the variable

Exists x ($\exists x$) : applies to at least one value of the variable

The **ordering** of the quantifiers affects the meaning of the statement

Very important for pumping lemma in chapters 4 and 7

Example

Theorem If S is an infinite set, then for every integer n there exists at least one subset T of S with n elements

$\forall$ precedes $\exists$; for the proof we must therefore (in that order)

- consider an arbitrary n
- prove the existence of a subset T of S with n elements

Set Equality

If E and F are sets, to prove $E = F$ we have to prove $E \subseteq F$ and $F \subseteq E$

This amounts to show two statements of the form "if H then C" :

- if x is in E then x is in F
- if x is in F then x is in E

Contrapositive

The statement "if H then C" is **equivalent** to the statement

"if C is false then H is false"

called **contrapositive**

Proof of equivalence uses **truth table**

In some cases, it may be easier to demonstrate the contrapositive

Also known as *modus tollens*

Inductive proof

Main technique when working on **recursivelly** defined objects (expressions, trees, etc.)

Induction on integers : we need to prove statement $S(n)$, for non-negative integer numbers n

- in the **base** case we show $S(i)$ for some specific integer i (usually $i = 0$ or $i = 1$)
- in the **inductive** step, for $n \geqslant i$ prove statement "if $S(n)$ then $S(n+1)$"

We can then conclude that $S(n)$ is true for every $n \geqslant i$, where i is the base case

Think: why is induction so powerful?

Example

Theorem If $x \geqslant 4$, then $2^x \geqslant x^2$

Proof

Base $x = 4 \Rightarrow 2^x = 16$ and $x^2 = 16$

Induction Let us assume $2^x \geqslant x^2$ for $x \geqslant 4$

We need to show that $2^{x+1} \geqslant (x+1)^2$:

- $2^{x+1} = 2 \cdot 2^x \geqslant 2 \cdot x^2$, from the inductive hypothesis
- we now show $2x^2 \geqslant (x+1)^2 = x^2 + 2x + 1$
- dividing by $x \neq 0$: $x \geqslant 2 + 1/x$
- if $x \geqslant 4$, $1/x \leqslant 1/4 \Rightarrow 2 + 1/x \leqslant 2.25$ □

Inductive proof

We can **extend** the base part to a finite number of cases

We can **extend** the inductive step and demonstrate for a certain $k > 0$: "if $S(n-k)$, $S(n-k+1)$, ..., $S(n-1)$, $S(n)$ then $S(n+1)$"

Structural induction

Many structures can be defined recursively

Definition of **arithmetic expression**

Base Any variable or number is an arithmetic expression

Induction If E and F are arithmetic expressions, then also $E + F$, $E \times F$, and (E) are arithmetic expressions

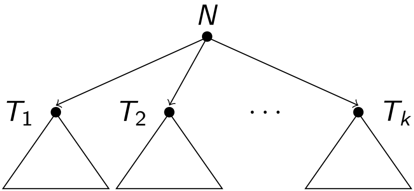
Example : $3 + (x \times 2)$ and $(2 \times (5 + 7)) \times y$ are arithmetic expressions

Structural induction

Definition of **tree** (with root)

Base A single node N is a tree with root N

Induction If $T_1, T_2, \ldots, T_k$, $k \geqslant 1$, are trees, the following structure is a tree with root N



Structural induction

To prove theorems for structure X which is recursively defined :

- show the statement for the base cases of the definition of X
- show the statement for X on the basis of the same statement holding for the subparts of X, according to X's definition

Example

Theorem Each arithmetic expression has an equal number of open and closed parentheses

Proof We proceed by induction on the number of parentheses

Base Both variables and numbers have zero open parentheses and zero closed parentheses

Induction Let us assume that E has n open and closed parentheses and F has m of them

There are three ways to recursively construct an arithmetic expression :

- $E + F$ has $n + m$ open brackets and $n + m$ closed brackets
- $E \times F$ has $n + m$ open brackets and $n + m$ closed brackets
- (E) has $n + 1$ open brackets and $n + 1$ closed brackets □

Example

Theorem Let T be a tree with n nodes and e arcs. Then $n = e + 1$

Before proving the theorem, try to get a visual intuition of why this is true

Proof By induction on T's structure

Base T has $n = 1$ and $e = 0$

Induction Assume T_i has n_i nodes and e_i arcs. By inductive hypothesis, $n_i = e_i + 1$

We have :

$$n = 1 + \sum_{i=1}^{k} n_i, \qquad e = k + \sum_{i=1}^{k} e_i$$

Example

We can write :

$$\begin{aligned} n &= 1 + \textstyle\sum_{i=1}^{k} \; n_i \\ &= 1 + \textstyle\sum_{i=1}^{k} \; (1 + e_i) \quad \textbf{inductive hypothesis} \\ &= 1 + k + \textstyle\sum_{i=1}^{k} \; e_i \\ &= 1 + e \end{aligned}$$

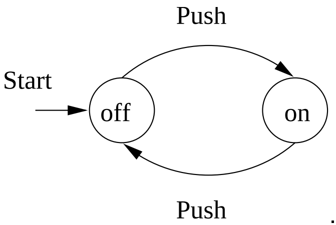
□

Mutual Induction

Sometimes it is not possible to prove a statement $S_1(n)$ by induction, because the statement depends on statements $S_2(n)$, ..., $S_k(n)$ of different types

We then need to prove **jointly** a family of statements $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ by **mutual induction** on n

Example



Theorem Given the automaton in the picture

- $S_1(n)$: After n push transitions, the automaton is in the off state if and only if n is even
- $S_2(n)$: After n push transitions, the automaton is in the on state if and only if n is odd

Example

Proof We proceed by induction on n

Base

[$S_1(0)$, if] After 0 push, the automaton is in the off state

[$S_1(0)$, only if] 0 even (conclusion) is always true

[$S_2(0)$, if] 0 odd (hypothesis) is false, so the implication is true

[$S_2(0)$, only if] The hypothesis is false (= the automaton is in the on state after 0 push), therefore the implication is always true

Example

Induction We assume $S_1(n)$ and $S_2(n)$ hold true, and we prove $S_1(n+1)$ and $S_2(n+1)$

[$S_1(n+1)$, if] From $n+1$ even we have n odd. By applying the inductive hypothesis $S_2(n)$, if part, we get that, after n push the automaton is in the on state. From the on state we have a push transition to the off state. Therefore after $n+1$ push transitions, the automaton is in the off state

[$S_1(n+1)$, only if] The automaton is in the off state after $n+1$ push transitions. Since there's only one push transition entering the off state, the automaton was in the on state after n push transitions. We apply the inductive hypothesis $S_2(n)$, only if part, and we get that n is odd. So $n+1$ even

Example

[$S_2(n+1)$, if] From $n+1$ odd we have n even. We apply the inductive hypothesis $S_1(n)$, if part, and obtain that after n push transitions, the automaton is in the off state. From the off state we have a push transition to the on state. Therefore after $n+1$ push transitions, the automaton is in the on state

[$S_2(n+1)$, only if] The automaton is in the on state after $n+1$ push transitions. Since there's only one push transition going into the on state, the automaton was in the off state after n push transitions. We apply the inductive hypothesis $S_1(n)$, only if part, and obtain that n is even. So $n+1$ is odd $\square$

Alphabet & strings

Alphabet : **finite** and **nonempty** set of atomic symbols

Example :

- $\Sigma = \{0, 1\}$, the binary alphabet
- $\Sigma = \{a, b, c, \ldots, z\}$, the set of all lowercase letters
- the set of all printable ASCII characters

String : **finite** sequence of symbols from some alphabet

- 0011001 string over $\Sigma = \{0, 1\}$

Alphabet & strings

Empty string : The string with zero symbols (taken from any alphabet) is denoted ϵ

Length of a string : Number of **occurrences** (standpoints) for the symbols in the string

- $|w|$ denotes the length of the string w
- $|0110| = 4$, $|\epsilon| = 0$

Alphabet & strings

Powers of an alphabet : Σ^k is the set of all k-length strings with symbols from Σ

- $\Sigma = \{0, 1\}$
- $\Sigma^1 = \{0, 1\}$; **ambiguity** between Σ and Σ^1

  Elements of Σ are alphabet symbols, elements of Σ^1 are strings
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^0 = \{\epsilon\}$

Question : How many strings are there in Σ^3 ?

Alphabet & strings

The set of all strings from Σ is denoted Σ^*

We have

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$
$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$$
$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

It is a mistake to write $\Sigma^+ \cup \epsilon$: why?

Alphabet & strings

Concatenation : If x and y are strings, then xy is the string obtained by putting a copy of y immediately after a copy of x

Example :

$$x = 01101$$
$$y = 110$$
$$xy = 01101110$$

Sometimes we also use the the '·' operator to represent concatenation and write $x \cdot y$

Some textbooks use the notation $x.y$

Alphabet & strings

For each string x :

$$x\epsilon = \epsilon x = x$$

ϵ is the **neutral** element of the concatenation

You can always think of ϵ occurring any number of times within a string :

$$\begin{aligned} x \cdot y &= x \cdot \epsilon \cdot y \\ &= x \cdot \epsilon \cdot \epsilon \cdot y \\ &= x \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot y \\ &= \cdots \end{aligned}$$

Compare with $2 + 3 = 2 + 0 + 3 = 2 + 0 + 0 + 3 = \cdots$

Alphabet & strings

Notational conventions :

- a, b, c, ..., a_1, a_2, ..., a_i, ... alphabet symbols
- u, w, x, y, z strings
- for $n \geqslant 0$, $a^n = aa \cdots a$ (a repeated n times)
- $a^0 = \epsilon$, $a^1 = a$

Languages

A **language** is a set of strings arbitrarily chosen from Σ^*, where Σ is an alphabet. $L \subseteq \Sigma^*$ is a language

Example :

- set of all the words in some English dictionary
- set of all Java programs without syntactic errors
- set of strings consisting of n zeros followed by n ones, with $n \geqslant 0$

$$\{\epsilon, 01, 0011, 000111, \ldots\}$$

- set of strings with an equal number of 0's and 1's

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

What is Σ in the first two cases above?

Languages

Example :

- set of binary numbers whose value is a prime

$$L_p = \{10, 11, 101, 111, 1011, \ldots\}$$

- empty language $\emptyset$, contains no string
- language $\{\epsilon\}$, contains only the empty string

Do not confuse these two languages :

$$\emptyset \neq \{\epsilon\}$$

Languages

Extensive representation of a language :

$$L = \{\epsilon, 01, 0011, 000111, 00001111, \ldots\}$$

Intensive representation of a language, using a **set-former** :

$$L = \{w \mid \text{ statement specifying } w\}$$

Example :

- $\{w \mid w$ consists of an equal number of 0's and 1's$\}$
- $\{w \mid w$ is an integer binary number whose value is prime$\}$
- $\{w \mid w$ is a syntactically correct Java program$\}$

Languages

Set-formers are often expressed in mathematical form :

$$L = \{w \mid w = 0^n 1^n,\ n \geqslant 0\}$$

or, in simplified form, also as :

$$L = \{0^n 1^n \mid n \geqslant 0\}$$

which is equivalent to :

$$L = \{\epsilon, 01, 0011, 000111, \ldots\}$$

Note the **implicit** universal quantifier for n in the set-former above

When needed, existential quantifiers are written explicitly

Languages

Example :

- $\{0^i 1^j \mid i, j \geqslant 1, \ i \geqslant j\}$
- $\{0^i 1^j \mid i, j \geqslant 1, \ i > j \text{ or } i < j\}$

The comma punctuation symbol is an implicit ‘and’ operator above

Note : do not confuse the two notations

- $\{0^n 1^n \mid n \geqslant 0\}$
- $\{0^n 1^n\}, \ n \geqslant 0$

There is **no precise syntax** for the use of set-formers

This requires some experience, many students get confused about this

Decision problems

Let $P(x)$ be a **predicate** expressing some mathematical property of element x

Decision problem associated with P : on input x, decide whether $P(x)$ holds true

Associated formal language (x viewed as a string) :

$$L_P = \{x \mid P(x) \text{ holds true}\}$$

The decision problem can be reformulated as : Given as input string x, decide whether $x \in L_P$

Example

For natural number x, $P(x)$ is true if x is a prime number. We represent x as a binary string

We define the language of prime numbers

$$L_p = \{10, 11, 101, 111, 1011, \ldots\}$$

Assigned as input the binary string x, decide whether $x \in L_p$

Decision problems

Many mathematical problems are not decision problems, but require instead a computation that constructs an output **result**

Think about search problems, optimization problems, etc.

We can reformulate these problems as decision problems

Example :

- given matrices A, B, construct the matrix $C = A \times B$
- associated decision problem : given a triple $\langle A, B, C \rangle$, decide whether $C = A \times B$

Decision problems

The general (non-decision) problem **is no easier** than the associated decision problem

You can solve the decision problem if you have a subroutine for the general problem

Example : Algorithm for decision problem using the algorithm for the general problem as a subroutine (**reduction** technique)

- input $\langle A, B, C\rangle$
- use subroutines on A, B to produce $C' = A \times B$
- if $C' = C$ answer yes, otherwise answer no

If you have enough computational resources to solve the general problem, then you can also solve the decision problem