BLG557E Theory of Computation

Tolga Ovatman

2018

Outline I

Outline II

Outline III

Computation Models

- Algorithm is a way to describe how we perform calculations. There is a need to make the algorithmic content of the world about us, mathematically explicit.
- Blaise Pascal's Pascaline and Gottfried Leibniz's Leibnitiana were early designs in 1650's that performs arithmetic operations.
- Charles Babbage's Analytical machine and Ada Lovelace's[1][2] first program to calculate Bernoulli numbers in 1850's.

[1]Lord Byron's daughter

[2]Student of DeMorgan

- David Hilbert's famous 20 problems in 1900's coined two discussions:
  1. Do there exist unsolvable problems in mathematics? Do there exist computational tasks for which there is no valid program?
  2. Is it possible to to capture mathematics in complete, consistent theories.
- Kurt Gödel's work on incompleteness theorem in 1930's
- Alan Turing and Alonzo Church's work on computability around 1940's.

Two important questions:

1. How can we define computation using a mathematical rigor?
2. Can we know what is computable and what is not? Can we measure computability?.

Definition

We say a set A is computable if there is an effective procedure for deciding whether $n \in A$ or $n \notin A$ for each $n \in \mathbb{N}$.

Example

Show that the set of even numbers is computable. As an answer we can provide with the algorithm: given n, divide n by two, if there is no remainder $n \in$ even numbers. Or the function $Rem(n/2)$.

Church-Turing Thesis

All effectively computable functions are partial recursive. A function is partial recursive iff it is Turing computable.

Models of Computation

- The Recursive Functions: Can we define a computation by inductive definitions of composition and recursion of function calls?
- The λ-Computable Functions: Can we define a computation by defining stateless substitutions defined as functions over values?
- Grammars-Languages/FSM: Can we use rewriting based or state based models to define computation using limited memory access?
- The Turing Computable Functions: Can we use a state based model having unlimited memory access?
- URM Computable Functions: Can we use an instruction based model having random memory access?

Spoiler

All of these computation models are equivalently strong, leading to the same class of functions: partial recursive functions.

λ-Calculus

λ-Calculus treats the functions as the main abstractions used to model the computation and the computation process is defined by applying the functions over arguments rather than applying step-by-step transformations over argument evaluations.

Definition

<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>

λ-Calculus

Example

$\lambda x.(x^2 - 2x + 5)$: A function definition
Notice the resemblance $func(x) = x^2 - 2x + 5$

$(\lambda x.(x^2 - 2x + 5))\ \ 2$: Application of the function over the value 2, evaluates to 5
Notice the resemblance $func(2) = 2^2 - 2 \cdot 2 + 5 = 5$ This kind of

application is also called β-reduction and expressed as :
$(\lambda x.E)N \triangleright E[x := N]$.

$(\lambda x.\lambda y.(\sqrt{x^2 + y^2}))\ \ 3\ \ 4$
$(\lambda xy.(\sqrt{x^2 + y^2}))\ \ 3\ \ 4$: A more compact form
Notice the resemblance $func(x, y) = x^2 + y^2$

λ-Calculus Syntax

Definition

As a convention, function application associates from the left, that is, the expression $E_1 E_2 E_3 \ldots E_n$ is evaluated by applying the expressions as $(((E_1 E_2) E_3) \ldots E_n)$

λ-Calculus: Free and Bound Variables

In λ calculus all names are local to definitions.

Identity Function

$\lambda x.x$ is the identity function
$(\lambda x.x)y$ applying identity function over y

Example

$\lambda x.xy$: x is bound, y is free
$(\lambda x.x)(\lambda y.yx)$: x is bound in the expression on the left. It is free in the expression on the right.

λ-Calculus: Substitution

Example

$(\lambda x.x)y = x[x := y] = y$: all occurrences of x are substituted by y in the expression
$(\lambda z.z)(\lambda x.x) = z[z := (\lambda x.x)] = \lambda x.x$: evaluates to identity function.

We should be careful when performing substitutions to avoid mixing up free occurrences of an identifier with bound ones.

Example

$(\lambda x.(\lambda y.xy))y \not\equiv (\lambda y.yy)$: the function to the left contains a bound y, whereas the y at the right is free
$(\lambda x.(\lambda t.xt))y \equiv (\lambda t.yt)$: Simply by renaming the bound y to t.

This kind of substitution is also called α-conversion and expressed as : $\lambda x.E[x] \triangleright \lambda z.E[z]$ if x is not free in E.

λ-Calculus

The following concepts can be modeled using λ-Calculus

1. Numerals can be modeled by counting the number of times a function has been applied on a variable
   1. (e.g. $\lambda fx.f(f(x)) \equiv 2$)
2. Arithmetic operations by introducing numerals as lambda functions
   1. (e.g. $\lambda xyz.x(yz)\ 2\ 2 \equiv \lambda z.2(2z) \equiv 2 \times 2$)
3. Conditionals and logical operations by introducing boolean values
   1. (e.g. $\lambda xy.x \equiv T$) and (e.g. $\lambda xy.y \equiv F$)
   2. (e.g. $\lambda xy.xy(\lambda uv.v) \equiv \wedge$)
   3. (e.g. $\lambda x.x(\lambda uv.v)(\lambda uv.u) \equiv \neg$)
   4. (e.g. $\lambda x.xFT \equiv$ if==0 expression)
4. Recursion by Y-combinator $\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$.

λ-Calculus Multiplication

Given

$\lambda fx.f(f(x)) \equiv 2$
$\lambda z.2(2z)$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned}
\lambda z.2(2z) &\equiv \lambda z.2(\ \ (\lambda fx.f(f(x)))z\ \ ) \\
&\equiv \lambda z.2(\lambda x.z(z(x))) \\
&\equiv \lambda z.(\lambda fu.f(f(u)))(\lambda x.z(z(x))) \\
&\equiv \lambda z.(\lambda u.[(\lambda x.z(z(x)))]([(\lambda x.z(z(x)))](u))) \\
&\equiv \lambda z.(\lambda u.[(\lambda x.z(z(x)))][z(z(u))]) \\
&\equiv \lambda z.(\lambda u.(z(z(z(z(u)))))) \\
&\equiv \lambda zu.(z(z(z(z(u)))))
\end{aligned}$$

λ-Calculus Logical and

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda xy.xy(\lambda uv.v) \equiv \wedge$
$T \wedge F$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} T \wedge F &\equiv [\lambda xy.xy(\lambda uv.v)][\lambda xy.x][\lambda xy.y] \\ &\equiv (\lambda xy.x)(\lambda xy.y)(\lambda uv.v) \\ &\equiv (\lambda xy.y) \end{aligned}$$

λ-Calculus Logical and

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda xy.xy(\lambda uv.v) \equiv \wedge$
$F \wedge T$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} F \wedge T &\equiv [\lambda xy.xy(\lambda uv.v)][\lambda xy.y][\lambda xy.x] \\ &\equiv (\lambda xy.y)(\lambda xy.x)(\lambda uv.v) \\ &\equiv (\lambda uv.v) \\ &\equiv (\lambda xy.y) \end{aligned}$$

λ-Calculus Logical and

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda xy.xy(\lambda uv.v) \equiv \wedge$
$T \wedge T$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} T \wedge T &\equiv [\lambda xy.xy(\lambda uv.v)][\lambda xy.x][\lambda xy.x] \\ &\equiv (\lambda xy.x)(\lambda xy.x)(\lambda uv.v) \\ &\equiv (\lambda xy.x) \end{aligned}$$

λ-Calculus Logical negation

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda x.x(\lambda uv.v)(\lambda uv.u) \equiv \neg$
$\neg F$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} \neg F &\equiv [\lambda x.x(\lambda uv.v)(\lambda uv.u)][\lambda xy.y] \\ &\equiv (\lambda xy.y)(\lambda uv.v)(\lambda uv.u) \\ &\equiv (\lambda uv.u) \\ &\equiv (\lambda xy.x) \end{aligned}$$

λ-Calculus Logical negation

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda x.x(\lambda uv.v)(\lambda uv.u) \equiv \neg$
$\neg T$

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned}\neg T &\equiv [\lambda x.x(\lambda uv.v)(\lambda uv.u)][\lambda xy.x] \\ &\equiv (\lambda xy.x)(\lambda uv.v)(\lambda uv.u) \\ &\equiv (\lambda uv.v) \\ &\equiv (\lambda xy.y)\end{aligned}$$

λ-Calculus if==0 expression

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda x.xFT \equiv \text{if}$
if==F

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} \text{if} == F &\equiv [\lambda x.xFT][\lambda xy.y] \\ &\equiv (\lambda xy.y)(\lambda xy.y)(\lambda xy.x) \\ &\equiv (\lambda xy.x) \end{aligned}$$

λ-Calculus if==0 expression

Given

$\lambda xy.x \equiv T$
$\lambda xy.y \equiv F$
$\lambda x.xFT \equiv \text{if}$
if==T

Note: Going to use [ ] to distinguish between expression

$$\begin{aligned} \text{if} == F &\equiv [\lambda x.xFT][\lambda xy.x] \\ &\equiv (\lambda xy.x)(\lambda xy.y)(\lambda xy.x) \\ &\equiv (\lambda xy.y) \end{aligned}$$

λ-Calculus loops

Given

$(\lambda x.xx)(\lambda x.xx)$ expression can be used to create a loop

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\equiv (\lambda x.xx)(\lambda x.xx)\\ &\equiv (\lambda x.xx)(\lambda x.xx)\\ &\equiv (\lambda x.xx)(\lambda x.xx)\end{aligned}$$

λ-Calculus recursion

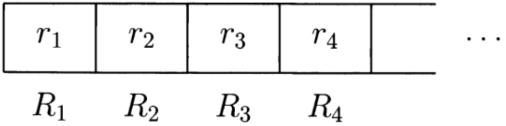
Given

$\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$ expression can be used to create recursion
Remember f(x) = E is expressed as $\lambda x.E$

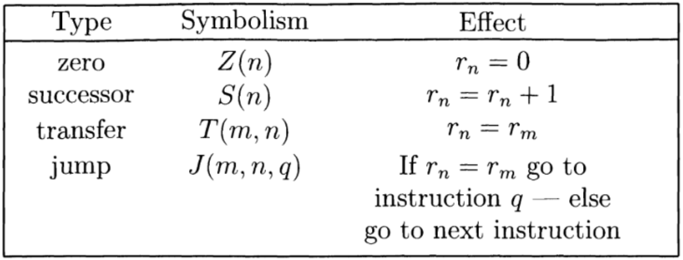
$$\begin{aligned}
[\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))]f &\equiv (\lambda x.f(xx))(\lambda x.f(xx)) \\
&\equiv f((\lambda x.f(xx))(\lambda x.f(xx)) \\
&\equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)) \\
&\equiv \lambda y.((\lambda x.y(xx))(\lambda x.y(xx))
\end{aligned}$$

Unlimited Register Machines

A URM has registers $R_1, R_2, \ldots$ which store natural numbers $r_1, r_2, \ldots$



A URM program is a finite list of instructions each having one of four basic types:

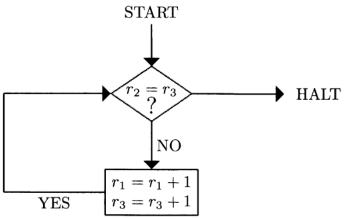


Type	Symbolism	Effect
zero	$Z(n)$	$r_n = 0$
successor	$S(n)$	$r_n = r_n + 1$
transfer	$T(m, n)$	$r_n = r_m$
jump	$J(m, n, q)$	If $r_n = r_m$ go to instruction q — else go to next instruction

Example URM Program

1. J(2,3,5)
2. S(1)
3. S(3)
4. J(1,1,1)

The computation of this program for input (7,2)



Instruction	R_1	R_2	R_3
1	7	2	0
2	7	2	0
3	8	2	0
4	8	2	1
1	8	2	1
2	8	2	1
3	9	2	1
4	9	2	2
1	9	2	2
Halt			

Definition

If f and g are URM-computable functions, then so is $f \circ g$

1. **P_g with input n may halt via a jump into the middle of P_f (or even past its end)**.
2. If we renumber the instructions of a P_f all the jump instructions $J(m, n, q)$ in P_f will need q modifications.
3. The initial index of P_g needs to be shifted.

Definition

P is in standard form if for every $J(m, n, q)$ in P we have $g \leq \ell(P) + 1$. It can be assumed that all programs P are in standard form.

Definition

If f and g are URM-computable functions, then so is $f \circ g$

1. P_g with input n may halt via a jump into the middle of P_f (or even past its end).
2. **If we renumber the instructions of a P_f all the jump instructions $J(m, n, q)$ in P_f will need q modifications**.
3. The initial index of P_g needs to be shifted.

Definition

The join of programs P and Q is the program $(P \mapsto Q)$, obtained by writing the instructions of Q after those of P, with each $J(m, n, q)$ in Q replaced by $J(m, n, \ell(P) + q)$

Definition

If f and g are URM-computable functions, then so is $f \circ g$

1. P_g with input n may halt via a jump into the middle of P_f (or even past its end).
2. If we renumber the instructions of a P_f all the jump instructions $J(m, n, q)$ in P_f will need q modifications.
3. **The initial index of P_g needs to be shifted**.

Definition

$Z[a; b]$, for $b \geq a$ is the program which cleans up all registers $R_a, \ldots, R_b$:

```
    1: Z(a)
    2: Z(a+1)
     ⋮
b-a+1: Z(b)
```

Definition

If f and g are URM-computable functions, then so is $f \circ g$

Let $\rho(P) =$ the largest index k of a register R_k used by P; and let $\rho = min\{\rho(P_f), \rho(P_g)\}$

P_g

$Z[2; \rho]$

P_f

computes the composition $f \circ g$.

Theorem

URM-computable functions are closed under composition, substitution and recursion.

Theorem

A function is URM computable iff, it is recursive.

Post Systems

Definition

A Post System Π is defined by:

$$\Pi = (C, V, A, P)$$

where

- C is a finite set of non-terminal constants(C_N) and terminal constants(C_T)
- V is a finite set of variables
- A is a finite set from C^*, called axioms
- P is a finite set of productions

Post Systems

The productions in a Post system must be of the form

$$x_1 V_1 x_2 \dots V_n x_{x+1} \to y_1 W_1 y_2 \dots W_m y_{m+1}$$

where,

- $x_i, y_i \in C^*$ and $V_i, W_i \in V$.
- Any variable V_i can appear at most once on the left.
- Each variable on the right must appear on the left: $\bigcup_{i=1}^{m} W_i \subseteq \bigcup_{i=1}^{n} V_i$

Theorem

A language is recursively enumerable iff there exists some Post system that generates it.

Post Systems

Example

Consider the post system with

- $C_T = \{1, +, =\}$
- $C_N = \emptyset$
- $V = \{V_1, V_2, V_3\}$
- $A = \{1 + 1 = 11\}$
- $P := V_1 + V_2 = V_3 \rightarrow V_1 1 + V_2 = V_3 1$
- $P := V_1 + V_2 = V_3 \rightarrow V_1 + V_2 1 = V_3 1$

Matrix Grammars

Matrix grammars are one of different rewriting systems.

- Matrix grammars differ from phrase-structure grammars in how the productions can be applied.
- For matrix grammars, the set of productions consists of subsets $P_1, P_2, \ldots$, each of which is an ordered sequence.
- Whenever the first production of some set P_i is applied, we **must** next apply the rest of the production sequence in order

Example

Consider the matrix grammar

- $P_1 : S \to S_1S_2$
- $P_2 : S_1 \to aS_1, S_2 \to bS_2c$
- $P_3 : S_1 \to \lambda, S_2 \to \lambda$

A derivation with this grammar is:
$S \to S_1S_2 \to aS_1bS_2c \to aaS_1bbS_2c \to aabbcc$

Other Rewriting Systems

- L-Systems: In each step of a derivation left-hand side string is replaced with right-hand side string. Productions start with an initial production. e.g.($a \rightarrow aa$)
- Markov structures: Markov structures start with an initial string from a language and work until empty string (or terminal production rule) is reached. e.g($ab \rightarrow \lambda, ba \rightarrow \lambda$)

Other Computation Models

- Parallel Computation and Process Modeling
- Biocomputing
- Agents
- Quantum Computing

References

- Linz, P. (2006). An introduction to formal languages and automata. Jones Bartlett Learning. Chapter 13 - Other Models of Computation
- Cooper, S. B. (2003). Computability theory. CRC Press. Chapter 2 - Models of Computability and the Church-Turing Thesis
- Rojas, R. (2015). A tutorial introduction to the lambda calculus. arXiv preprint arXiv:1503.09060.

Languages and Grammars

Finite State Machine(FSM)

An FSM has a mathmematical model defined by a quintuple $(S, I, O, \delta, \omega)$, where:

- S: Set of states.
- I: Input alphabet (a finite, non-empty set of symbols)
- O: Output alphabet (a finite, non-empty set of symbols)
- δ: The transition function defined on $I \times S \to S$
- ω: The output function defined on either $S \to O$ or $I \times S \to O$

Formal Languages

Language

Language may refer either to the specifically human capacity for acquiring and using complex systems of communication, or to a specific instance of such a system of complex communication.

Formal Languages

Formal Language

In mathematics, computer science, and linguistics, a formal language is a set of strings of symbols. The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed which are called words.

Definitions

- Alphabet: A finite, non-empty set of symbols or characters. Denoted as Σ
- Word: A finite array or string from Σ
- Word Length: Number of symbols in a word.
- Empty string: A word of length 0. Denoted as Λ or ϵ.

Word concatenation

Concatenation is performed by joining two character strings end-to-end.

$$(x = a_1a_2 \ldots a_n) \wedge (y = b_1b_2 \ldots b_m) \Rightarrow xy = a_1a_2 \ldots a_nb_1b_2 \ldots b_m(x\&y)$$

Identity element of concatenation is Λ
$x = \Lambda \Rightarrow xy = y$
$y = \Lambda \Rightarrow xy = x$

We can build a monoid set of words(Σ^*) from the alphabet using concatenation operation having an empty string as an identity element and holding associativity property.

Reverse of a string

$(baba)^R = abab$
$(ana)^R = ana$
We can define the operation using induction:

1. $|w| = 0 \Rightarrow w^R = w = \Lambda$
2. Assume we have two strings w and u.
   $|u| = n$ $\qquad$ $|w| = n + 1$
   $n \in \mathbb{N}$. Following these assumptions
   $(w = ua) \wedge (a \in \Sigma) \Rightarrow w^R = au^R$

For example:

$$|w| = 1 \Rightarrow w = \Lambda a$$
$$w^R = a\Lambda = w$$
$$|w| = 2 \Rightarrow w = ua(u = b)$$
$$w^R = au^R = au = ab$$
$$|w| = 3 \Rightarrow w = ua, u = cb$$
$$w^R = au^R = abc$$

Reverse of a string

Theorem

$(wx)^R = x^R \cdot w^R \wedge |x| = n, |w| = m \wedge m, n \in \mathbb{N}$

Proof.

Basis step:
$|x| = 0 \Rightarrow x = \Lambda$
$(wx)^R = (w\Lambda)^R = w^R = \Lambda w^R = \Lambda^R w^R = x^R w^R$

Inductive step: $|x| \leq n \Rightarrow (wx)^R = x^R w^R$

For $|x| = n + 1$
$(x = ua) \wedge (|u| = n) \wedge (a \in \Sigma) \wedge (x^r = au^r)$
$(wx)^R = (w(ua))^R = ((wu)a)^R = a(u^R w^R) = au^R w^R = x^R w^R$

$\square$

For example (snow ball)$^{\mathrm{R}}$= (ball)$^{\mathrm{R}}$(snow)$^{\mathrm{R}}$

Σ^+ denotes the set of non-empty strings over the Σ alphabet

i $a \in \Sigma \Rightarrow a \in \Sigma^+$

ii $(x \in \Sigma^+ \wedge a \in \Sigma) \Rightarrow ax \in \Sigma^+$

iii Σ^+ doesn't contain any elements other than the ones that can be constructed by applying i and ii finitely.

For example:
$\Sigma = \{a, b\} \Rightarrow \Sigma^+ = \{a, b, aa, ba, ab, bb, aaa, aab, \ldots\}$.

Σ^* denotes the set of all strings over the Σ alphabet

- **i** $\Lambda \in \Sigma^*$
- **ii** $(x \in \Sigma^* \wedge a \in \Sigma) \Rightarrow ax \in \Sigma^*$
- **iii** Σ^* doesn't contain any elements other than the ones that can be constructed by applying i and ii finitely.

For example:

$\Sigma = \{a, b\} \Rightarrow \Sigma^* = \{\Lambda, a, b, aa, ba, ab, bb, aaa, aab, \ldots\}$
$\Sigma = \{0, 1\} \Rightarrow \Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$

Assuming $(x \in \Sigma^*) \wedge (\forall n \in \mathbb{N})$, a string's nth power($x^n$) can be formally defined as :

1. $x^0 = \Lambda$
2. $x^{n+1} = x^n \cdot x = (x^n \& x)$

For example:
$\Sigma = \{a, b\}$ $x = ab$
$x^0 = \Lambda,\ \ x^1 = ab,\ \ x^2 = abab,\ \ x^3 = ababab$

or
$\{a^n b^n | n \geq 0\}$ defines the set: $\{\Lambda, ab, aabb, aaabbb, \ldots\}$

Let Σ be a finite alphabet.

A language over Σ is a subset of Σ^*.

The rules of choosing a subset from Σ^* can be performed by using a grammar.

For example:
$\{a^m b^n | m, n \in \mathbb{N}\}$ is a language defined over $\{a, b\}$.

This language cannot contain any other symbols than a and b.

In this language b always succeeds a. For instance ba doesn't belong to this language.

Multiplication of Languages

Let A and B be languages defined over Σ. Cartesian product $A \times B$ produces a new language.
$AB = \{xy | x \in A \wedge y \in B\}$
$AB = \{z | z = xy \wedge x \in A \wedge y \in B\}$

For example:
Let $\Sigma = \{a, b\}$ be the alphabet
Let $A = \{\Lambda, a, ab\}$ and $B = \{a, bb\}$ be the languages
$AB = \{a, bb, aa, abb, aba, abbb\}$
$BA = \{a, aa, aab, bb, bba, bbab\}$
$AB \neq BA$

Theorem

Let $\varnothing$ denote empty language; A,B,C,D denote different languages defined over the alphabet Σ.

1. $A\varnothing = \varnothing A = \varnothing$
2. $A\{\Lambda\} = \{\Lambda\}A = A$
3. $(AB)C = A(BC)$
4. $(A \subset B) \wedge (C \subset D) \Rightarrow AC \subset BD$
5. $A(B \cup C) = AB \cup AC$
6. $(B \cup C)A = BA \cup CA$
7. $A(B \cap C) \subset AB \cap AC$
8. $(B \cap C)A \subset BA \cap CA$

Proof.

$$A\varnothing = \varnothing A = \varnothing$$

$A\varnothing = \{xy | x \in A \wedge y \in \varnothing\}$ doesn't hold since $\forall y; y \notin \varnothing$.

Since there doesn't exist any x, y couples satisfying the condition
$A\varnothing = \varnothing$ □

Proof.

$$(A \subset B) \land (C \subset D) \Rightarrow AC \subset BD$$

We are going to use a direct proof. We assume
$A \subset B \land C \subset D$ and $z \in AC$

$$\begin{aligned} z = xy \Leftrightarrow x \in A \land y \in C \\ (A \subset B \land x \in A \Rightarrow x \in B) \land (C \subset D \land y \in C \Rightarrow y \in D) \\ (x \in B \land y \in D) \Leftrightarrow xy \in BD \\ (xy \in AC \Rightarrow xy \in BD) \Leftrightarrow AC \subset BD \end{aligned}$$

$\square$

Proof.

$$A(B \cap C) \subset AB \cap AC$$

$$\begin{aligned} A(B \cap C) &\Rightarrow \forall z(z = xy \land x \in A \land y \in B \cap C) \\ x \in A \land y \in B \cap C &\Leftrightarrow x \in A \land y \in B \land y \in C \\ x \in A \land y \in B \land y \in C &\Leftrightarrow x \in A \land y \in B \land x \in A \land y \in C \\ x \in A \land y \in B \land x \in A \land y \in C &\Leftrightarrow xy \in AB \land xy \in AC \\ xy \in AB \land xy \in AC &\Rightarrow xy \in (AB \cap AC) \\ \forall z(z \in A(B \cap C) &\Rightarrow z \in AB \cap AC) \end{aligned}$$

□

Let's give a counter example to show that $AB \cap AC \subseteq A(B \cap C)$ might not hold all the time.

$A = \{a^n | n \in \mathbb{N}\}$
$B = \{a^n b^n | n \in \mathbb{N}\}$
$C = \{b^n | n \in \mathbb{N}\}$

Let's take $z = a^5 b^2$. $z \in AB \cap AC$
However $B \cap C = \{\Lambda\}$ therefore
$z \notin A(B \cap C)$.

$z = a^5 b^2 = a^3 a^2 b^2$
$z \in AB \cap AC$
$z \notin A(B \cap C)$ because $B \cap C = \{\Lambda\}$

A^n : Let language A be defined over Σ

- **i** $A^0 = \{\Lambda\}$
- **ii** $A^{n+1} = A^n A; \forall n \in \mathbb{N}$

For example:
$\Sigma = \{a, b\}$
$A = \{\Lambda, a, b\}$
$A^1 = A$
$A^2 = \{\Lambda, a, b, aa, ab, ba, bb\}$

Theorem

Let A and B denote different languages defined over the alphabet Σ.

1. $A^m A^n = A^{m+n}$
2. $(A^m)^n = A^{mn}$
3. $A \subset B \Rightarrow A^n \subset B^n \ldots$

Proof.

$$A^m A^n = A^{m+n}$$

We are going to use induction.

For the base case, we use the previous definition: $n = 1 : A^m A^1 = A^{m+1}$

Inductive step: $A^m A^{n+1} = A^m A^n A^1$
Concat op. is associative: $A^m(A^n A^1) = (A^m A^n)A^1 = A^{m+n} A^1$ □

Proof.

$$(A^m)^n = A^{mn}$$

We are going to use induction.

For the base case $n = 1 : A^m = A^m$

Inductive step: $(A^m)^{n+1} = (A^m)^n(A^m)^1 = A^{mn+m} = A^{m(n+1)}$

$\square$

Proof.

$$A \subset B \Rightarrow A^n \subset B^n \ldots$$

Let's remember $A^2 = A \times A, B^2 = B \times B$ and
$A \subset B \Rightarrow (\forall x (x \in A \rightarrow x \in B))$

For $n = 2$:
$A^2 = \{xy | x \in A \wedge y \in A\}$
$A \subset B \Rightarrow (x \in A \rightarrow x \in B)$
$A \subset B \Rightarrow (y \in A \rightarrow y \in B)$

$(\forall xy (xy \in A^2 \rightarrow xy \in B^2)) \Rightarrow A^2 \subset B^2$
Proof continues similarly for $n := n + 1$ $\square$

Star closure (Kleene Star, Kleen Closure)

Positive Closure

$A^+ : \bigcup_{n=1} A^n = A \cup \ldots$

Star Closure

$A^* : \bigcup_{n=0} A^n = A^0 \cup A \cup \ldots$

Theorem

Let A and B denote different languages defined over the alphabet Σ and let $n \in \mathbb{N}$.

1. $A^* = \Lambda \cup A^+$. *Derived from the definition*
2. $A^n \subseteq A^*, n \geq 0$. *Derived from the definition*
3. $A^n \subseteq A^+, n \geq 1$. *Derived from the definition*
4. $A \subseteq AB^*$ *Proof tip:* $A^0 \subseteq B^* \Rightarrow A \subseteq AB^*$

Theorem

Let A and B denote different languages defined over the alphabet Σ and let $n \in \mathbb{N}$.

5 $A \subseteq B^*A$

6 $A \subseteq B \Rightarrow A^* \subseteq B^*$. *Can be proven by using $A \subseteq B \Rightarrow A^n \subseteq B^n$. Once true for all n, then it is true for union of all n.*

7 $A \subseteq B \Rightarrow A^+ \subseteq B^+$

8 $AA^* = A^*A = A^+$

9 $\Lambda \in A \Leftrightarrow A^+ = A^*$

10 $(A^*)^* = A^*A^* = A^*$

11 $(A^*)^+ = (A^+)^* = A^*$

12 $A^*A^+ = A^+A^* = A^+$

13 $(A^*B^*)^* = (A \cup B)^* = (A^* \cup B^*)^*$.

Proof.

$$AA^* = A^*A = A^+$$

$AA^* = A^+ \Rightarrow A(A^0 \cup A^1 \cup \ldots) = A \cup A^2 \cup \ldots = A^+$ □

Proof.

$$\Lambda \in A \Leftrightarrow A^+ = A^*$$

First let's show $\Lambda \in A \Rightarrow A^+ = A^*$

$\Lambda \in A \Rightarrow A \cup \{\Lambda\} = A$
$A \cup A^0 = A$

$A^+ = A \cup \{\bigcup_{n=2} A^n\} = A^0 \cup A \cup \{\ldots\} = A^*$ □

Proof.

$$\Lambda \in A \Leftrightarrow A^+ = A^*$$

Secondly $A^+ = A^* \Rightarrow \Lambda \in A$
If $A \cup A^2 \cup \ldots = A^0 \cup A \cup \ldots$
$A^0 \subseteq A \cup A^2 \cup \ldots$
This inclusion can be interpreted in two different ways:

- **i** $\Lambda \in A \Rightarrow A^0 \subseteq A$
- **ii** $\Lambda \notin A \Rightarrow \exists i, \Lambda \in A^i, i \in (\mathbb{N}^+ - 1)$

However $\Lambda \notin A \Rightarrow \forall x \in A^i(|x| \geq i)$
on the contrary $|\Lambda| = 0 \Rightarrow \Lambda \notin A^i \wedge i \geq 2$.
We have a contradiction proving ii wrong. Therefore i must be true.

□

Proof.

$$(A^*B^*)^* = (A \cup B)^*$$

We need to prove both $(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$

$$\begin{gathered}
A \subseteq A \cup B \wedge B \subseteq A \cup B \\
A^* \subseteq (A \cup B)^* \wedge B^* \subseteq (A \cup B)^* \\
A^*B^* \subseteq (A \cup B)^* \\
(A^*B^*)^* \subseteq ((A \cup B)^*)^* \\
(A^*B^*)^* \subseteq (A \cup B)^*
\end{gathered}$$

□

Proof.

$$(A^*B^*)^* = (A \cup B)^*$$

We need to prove both $(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$
$A \subseteq A^* \subseteq A^*B^*$ and $B \subseteq B^* \subseteq A^*B^*$

It is possible to write statements above since A^* and B^* contains Λ
$(A \cup B \subseteq A^*B^*) \Rightarrow (A \cup B)^* \subseteq (A^*B^*)^*$
$(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$.

$\square$

Theorem

Let A and B denote subsets of Σ^* and let $\Lambda \notin A$.
The only solution of $X = AX \cup B$ is $X = A^*B$

Formal Grammar

A formal grammar is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.

In our context
Σ denotes the alphabet
Σ^* is the set of all words that can be constructed using Σ
Grammar is the set of formation rules to construct a subset of Σ^*
For example:
To construct arithmetic expressions following can be used:
$\mathbb{Z}\ \ +\ \ -\ \ \times\ \ /\ (\ \ )$. All the well-formed arithmetic expressions are meaningful except division by zero.
$(((2-1)/3)+4\times 6)$ is a well-formed and meaningful arithmetic expression. $2+(3/(5-(10/2)))$ is well-formed as well but not meaningful because of division by zero.

The Syntax of Grammars

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s. A grammar G(also called a phase structure grammar) is formally defined as the tuple $(N, \Sigma, n_0, \mapsto)$:

- N is a set of *nonterminal symbols*[a].
- Σ is a set of *terminal symbols* that is disjoint from N
- n_0 is the start symbol
- $\mapsto$ is the set of production rules.
  If we call $V = N \cup \Sigma$, $\mapsto$ is a relation defined over V^*.
  It has the structure: $V^*NV^* \rightarrow V^*$
  where $*$ is a Kleene star operator.

[a] A terminal symbol cannot be replaced with another set of symbols

Formal grammars are also sometimes denoted as phase structure grammars in the literature.

The Semantics of Grammars

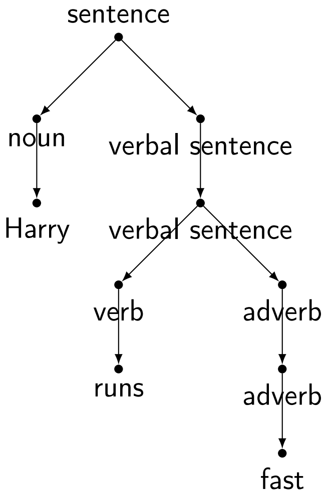
The operation of a grammar can be defined in terms of relations on strings:

- The direct derivability relation over a grammar $G = (N, \Sigma, n_0, \mapsto)$ on strings V^* is denoted as $\Rightarrow_G$ and formally defined as follows:
  $x \Rightarrow_G y \leftrightarrow \exists l, w, r \in V^* : (x = lwr) \wedge (w \rightarrow w' \in \mapsto) \wedge (y = lw'r)$
- The reflexive transtive closure of $\Rightarrow_G$ gives us the reachability relation ${\Rightarrow^*}_G$.
  The reachability of direct derivability denotes the words than can be derived in a finite number of steps.
  Therefore for any $\sigma \in \Sigma^*$ the statement $n_0 {\Rightarrow^*}_G \sigma$ denotes a well-formed string derived by the grammar of a language L,
  in other words $L(G) = \{\sigma | n_0 \Rightarrow^* \sigma\}$.

Example 1

S =Harry,Sally,runs,swims,fast,often,long
N =sentecte,verbal sentence,noun,verb,adverb
n_0 =sentence
Grammar:
sentence $\mapsto$ noun verbal sentence
noun $\mapsto$ Harry
noun $\mapsto$ Sally
verbal sentence $\mapsto$ verb adverb
adverb $\mapsto$ fast
adverb $\mapsto$ often
adverb $\mapsto$ long
verb $\mapsto$ runs
verb $\mapsto$ swims

We can apply these rules in one of two ways to derive a well-formed syntax. We can either always expand the leftmost expression first or



$S =$Harry,Sally,runs,swims,fast,often,long

$N =$sentence,verbal

sentence,noun,verb,adverb

$n_0 =$sentence

Grammar:

sentence $\mapsto$ noun, verbal sentence

noun $\mapsto$ Harry

noun $\mapsto$ Sally

verbal sentence $\mapsto$ verb, adverb

adverb $\mapsto$ fast

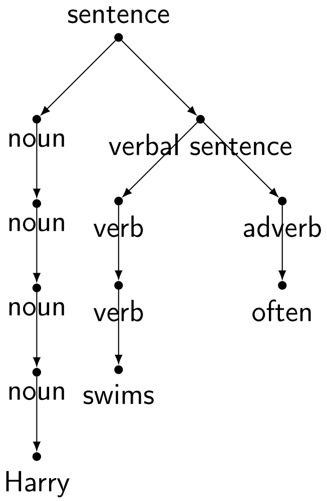
adverb $\mapsto$ often

adverb $\mapsto$ long

verb $\mapsto$ runs

verb $\mapsto$ swims

We can apply these rules in one of two ways to derive a well-formed syntax. We can either always expand the rightmost expression first



$S =$Harry,Sally,runs,swims,fast,often,long

$N =$sentence,verbal

sentence,noun,verb,adverb

$n_0 =$sentence

Grammar:

sentence $\mapsto$ noun, verbal sentence

noun $\mapsto$ Harry

noun $\mapsto$ Sally

verbal sentence $\mapsto$ verb, adverb

adverb $\mapsto$ fast

adverb $\mapsto$ often

adverb $\mapsto$ long

verb $\mapsto$ runs

verb $\mapsto$ swims

and derive a canonical parse tree.

Parsing

Parsing, or, syntactic analysis, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given formal grammar.

Example 2

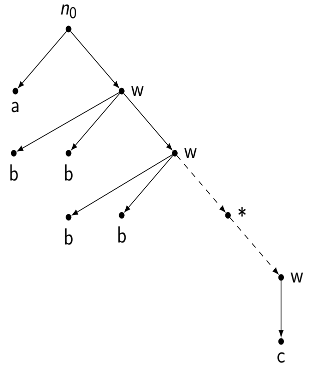
$S = a, b, c$

$N = n_0, w$

$\mapsto = \{n_0 \rightarrow aw | w \rightarrow bbw | w \rightarrow c\}$

Example 2

$S = a, b, c$
$N = n_0, w$
$\mapsto = \{n_0 \to aw | w \to bbw | w \to c\}$



$L(G) = \{a\} \cdot \{bb\}^* \cdot \{c\}$
$= a(bb)^* c$
$\{a(bb)^n c | n \in \mathbb{N}\}$

Example 3

$S = a, b, c$
$N = n_0, w$
$\mapsto = \{n_0 \to an_0b | n_0b \to bw | abw \to c\}$

$$\begin{aligned} n_0 &\Rightarrow an_0b \\ an_0b &\Rightarrow a(an_0b)b \Rightarrow a^n n_0 b^n \\ a^n(n_0b)b^{n-1} &\Rightarrow a^n bwb^{n-1} \\ a^{n-1}(abw)b^{n-1} &\Rightarrow a^{n-1}cb^{n-1} \end{aligned}$$

Or in a general form: $L(G) = \{a^m cb^m | m \in \mathbb{N}\}$

Example 4

$S = a, b, c$

$N = n_0, A, B, C$

$\mapsto = \{$

$n_0 \rightarrow A$

$A \rightarrow aABC$

$A \rightarrow abC$

$CB \rightarrow BC$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

$\}$

n_0

$\underline{\mathrm{A}}$

$\mathrm{a\underline{A}BC}$

$\mathrm{aa\underline{A}BCBC}$

$\mathrm{aaab\underline{C}\underline{B}CBC}$

$\mathrm{aaa\underline{b}\underline{B}C\underline{C}\underline{B}C}$

$\mathrm{aaabb\underline{C}\underline{B}CC}$

$\mathrm{aaab\underline{b}\underline{B}CCC}$

$\mathrm{aaabb\underline{b}\underline{C}CC}$

$\mathrm{aaabbb\underline{c}\underline{C}C}$

$\mathrm{aaabbbc\underline{c}\underline{C}}$

$\mathrm{aaabbbccc}$

$a^3b^3c^3$

$L(G) = \{a^k b^k c^k | k \in \mathbb{N}^+\}$

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 0 grammars

Type 0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine.
These languages are also known as the recursively enumerable languages.
Example 3 conforms to a Type 0 grammar.

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 1 grammars

Type 1 grammars (context-sensitive grammars) have transformation rules s.t. for $w_1 \rightarrow w_2$ rule $|w_1| \leq |w_2|$ should hold.
Conetext sensitivity follows the rules having the form $lwr \rightarrow lw'r$
The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton.
Example 4 conforms to a Type 1 grammar.

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 2 grammars

Type 2 grammars (context-free grammars) generate the context-free languages.
These are defined by rules of the form $A \rightarrow \gamma$ with a nonterminal(A) and a string of terminals and nonterminals(γ).
These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton.
Context-free languages are the theoretical basis for the syntax of most programming languages.
Example 1 conforms to a Type 2 grammar.

Chomsky Hierarchy

Type 3 grammars

Type 3 grammars (regular grammars) generate the regular languages.
Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a number of terminals, possibly followed by a single nonterminal.
These languages are exactly all languages that can be decided by a finite state automaton.
Additionally, this family of formal languages can be obtained by regular expressions.
Regular languages are commonly used to define search patterns.
Example 2 conforms to a Type 3 grammar.

Only Type-2 and Type-3 grammars have syntax trees.
$n_0 \rightarrow \Lambda$ rule can be added to any type to support empty sentences
Type 3 $\subseteq$ Type 2 $\subseteq$ Type 1 $\subseteq$ Type 0

Chomsky Hierarchy

Type	Language (Grammars)	Form of Productions in Grammar	Accepting Device
0	Recursively enumerable (unrestricted)	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*$, α contains a variable)	Turing machine
1	Context-sensitive	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert$, α contains a variable)	Linear-bounded automaton
2	Context-free	$A \to \alpha$ $(A \in N, \alpha \in (N \cup \Sigma)^*)$	Pushdown automaton
3	Regular	$A \to aB, A \to \Lambda$ $(A, B \in N, a \in \Sigma)$	Finite automaton

Regular Expression

Regular expressions provide a concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for "regular expression" include regex and regexp.
A regular expression can be defined over an alphabet Σ by induction:

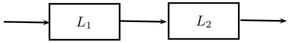
1. Each and every element of Λ and Σ is a regular expression
   $L(\Lambda) = \{\Lambda\}; L(a) = \{a\}; \forall a \in \Sigma$
2. Concetanation operation$(\cdot)$ on two regular expressions produce another regular expression
   $L(\alpha\beta) = L(\alpha)L(\beta)$
3. Alternation operation$(\vee)$ on two regular expressions produce another regular expression
   $L(\alpha \vee \beta) = L(\alpha) \vee L(\beta)$
4. Kleene star operation$(*)$ on a regular expression produce another regular expression.
   $L(\alpha^*) = L^*(\alpha)$

Theorem

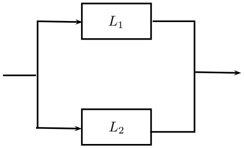
Let L be a language described over S s.t. $L \subseteq S^*$. L is called a regular language if it conforms to a regular grammar G. In other words $L = L(G)$.
Regular expressions describe regular languages in formal language theory. There exists an isomorphism between the described language and the regular expression. They have the same expressive power as regular grammars.

Syntax diagrams can be used define regular expressions. The following three diagrams describe basic properties of regular expressions.

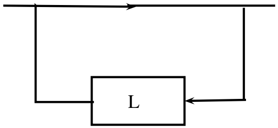
Concetanation ($\alpha_1 \alpha_2$)



Alteration ($\alpha_1 \vee \alpha_2$)

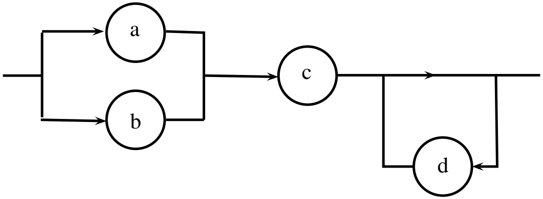


Kleene star ($(\alpha_1)^*$)

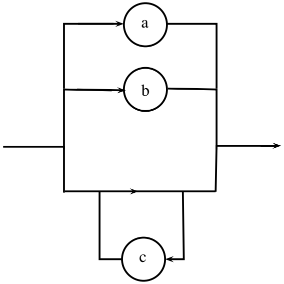


Following are some additional examples on syntax diagrams.

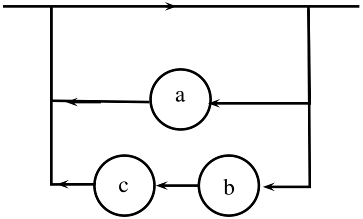
$(a \vee b)cd^*$



$a \vee b \vee c^*$



$(a \vee bc)^*$



For further practice check out the eclipse plugin Regexper project[3].

Finite Automata

Definitions

Automaton

An automaton is an abstract model of a machine that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input. The most general and powerful automata is the Turing machine.

Definitions

Deterministic Finite Automata

A DFA accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. *Deterministic* refers to the uniqueness of the computation.

The major objective of automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems, in which signals are sampled periodically.

Formal Definition of a DFA

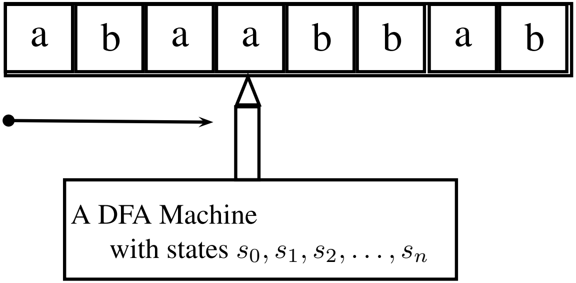
A deterministic finite state machine is a quintuple $M = (\Sigma, S, s_0, \delta, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- s_0: An initial state, an element of S.
- δ: The state-transition function $\delta : S \times \Sigma \rightarrow S$
- F: The set of final states where $F \subseteq S$.

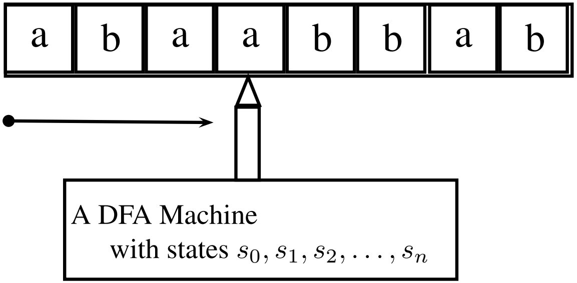
This machine is a Moore machine where each state produces the output in set $Z = \{0, 1\}$ corresponding to the machine's accepting/rejecting conditions.

DFA as a machine

Consider the physical machine below with an *input tape*. The tape is divided into cells, one next to the other. Each cell contains a symbol of a word from some finite alphabet. A machine that is modeled by a DFA, reads the contents of the cell successively and when the last symbol is read, the word is said to be accepted if the DFA is in an accepted state.



DFA as a machine



A run can be seen as a sequence of compositions of transition function with itself. Given an input symbol $\sigma \in \Sigma$ when this machine reads σ from the strip it can be written as $\delta(s, \sigma) = s' \in S$.

Configuration

A computation history is a (normally finite) sequence of configurations of a formal automaton. Each configuration fully describes the status of the machine at a particular point.
$s, \omega \in S \times \Sigma^*$

Configuration derivation is performed by a relation $\vdash_M$. If we denote the tuples in $\vdash_M$ as (s, ω) and (s', ω'), the relation can be defined as:

a $\omega = \sigma\omega' \wedge \sigma \in \Sigma$

b $\delta(s, \sigma) = s'$

A transition defined by this relation is called *derivation in one step* and denoted as $(s, \omega) \vdash_M (s', \omega')$. Following definitions can be defined based on this:

- Derivable configuration: $(s, \omega) \vdash_M^* (s', \omega')$ where $\vdash_M^*$ is the reflexive transitive closure of $\vdash_M$
- Recognized word: $(s_0, \omega) \vdash_M^* (s_i, \Lambda)$ where $s_i \in F$.
- Execution: $(s_0, \omega_0) \vdash (s_1, \omega_1) \vdash (s_2, \omega) \vdash \ldots \vdash (s_n, \Lambda)$ where Λ is the empty string.
- Recognized Language: $L(M) = \{\omega \in \Sigma^* | (s_0, \omega) \vdash_M^* (s_i, \Lambda) \wedge s_i \in F\}$

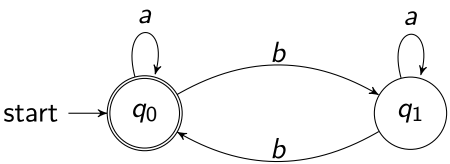
Language Recognizer

The reflexive transitive closure of $\vdash_M$ is denoted as ${\vdash_M}^*$. $(q,\omega){\vdash_M}^*(q',\omega')$ denotes that (q,ω) yields (q',ω') after some number of steps.
$(s,\omega){\vdash_M}^*(q,\Lambda)$ denotes that $\omega \in \Sigma^*$ is recognized by an automaton if $q \in F$. In other words $L(M) = \{\omega \in \Sigma^* | (s,\omega){\vdash_M}^*(q_i,\Lambda) \wedge q_i \in F\}$

Example 1

$S = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$s_0 = q_0$
$F = \{q_0\}$

q	σ	$\delta(q,\sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

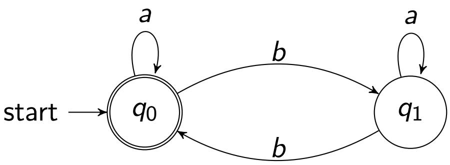


$$(q_0, aabba) \vdash_M (q_0, abba)$$
$$(q_0, abba) \vdash_M (q_0, bba)$$
$$(q_0, bba) \vdash_M (q_1, ba)$$
$$(q_1, ba) \vdash_M (q_0, a)$$
$$(q_0, a) \vdash_M (q_0, \Lambda)$$

Example 1

$S = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$s_0 = q_0$
$F = \{q_0\}$

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0



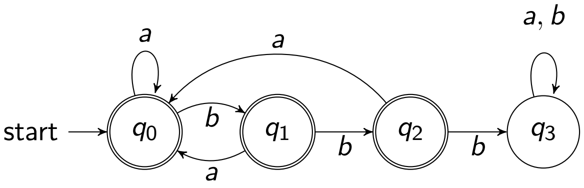
$L(M) = (a \vee ba^*b)^*$. We can write the grammar as:

$V = S \cup \Sigma$
$I = \Sigma = \{a, b\}$
$s_0 = q_0 = n_0$
$< q_0 > ::= \Lambda | a < q_0 > | b < q_1 > | a$
$< q_1 > ::= b < q_0 > | a < q_1 > | b$

Example 2

$L(M) = \{\omega | \omega \in \{a, b\}^* \wedge \omega$ should not include three successive b's$\}$
$S = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, s_0 = q_0, F = \{q_0, q_1, q_2\}$

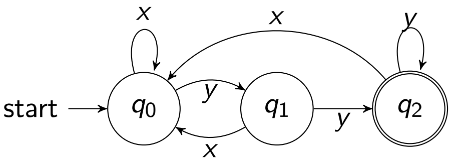
q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_2
q_2	a	q_0
q_2	b	q_3
q_3	a	q_3
q_3	b	q_3



We have a dead state q_3 where the automaton is not able to change state once it visits the dead state. $L(M) = [(\Lambda \vee b \vee bb)a]^*(\Lambda \vee b \vee bb)$

Example 3

$S = \{q_0, q_1, q_2\}, \Sigma = \{x, y\}, s_0 = q_0 = n_0, F = \{q_2\}$



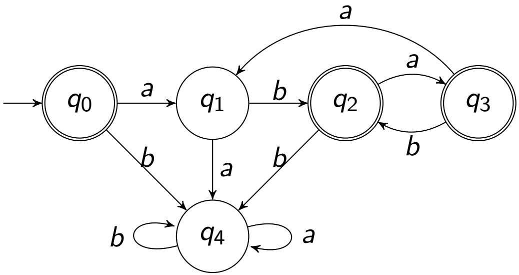
$< q_0 >::= x < q_0 > \mid y < q_1 >$
$< q_1 >::= y < q_2 > \mid \ y \ \mid x < q_0 >$
$< q_2 >::= y \ \mid y < q_2 > \mid x < q_0 >$

$L(M) = ((x \vee yx)^* yy^+ x)^* (x \vee yx)^* yy^+$
$L(M) = ((\Lambda \vee y \vee yy^+)x)^* yy^+ = (y^* x)^* yy^+$
$L(M) = (x \vee yx \vee yy^+ x)^* yyy^*$

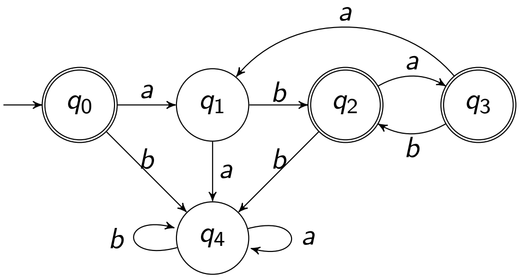
Non-deterministic Finite Automata(NFA)

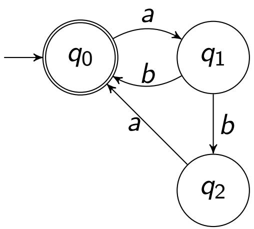
In an NFA, given an input symbol it is possible to jump into several possible next states from each state.

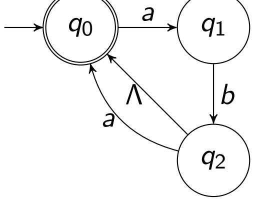
A DFA recognizing $L = (ab \vee aba)^*$ can be diagramatically shown as:

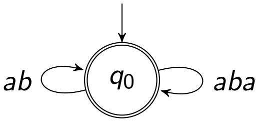


We may construct three different NFAs recognizing the same language.









Formal Definition of an NFA

A non-deterministic finite state automata is a quintuple
$M = (\Sigma, S, s_0, \Delta, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- s_0: An initial state, an element of S.
- Δ: The state-transition relation $\Delta \subseteq S \times \Sigma^* \times S$
- F: The set of final states where $F \subseteq S$.

A configuration is defined as a tuple in set $S \times \Sigma^*$. Considering the definition of derivation in one step:
$(q, \omega) \vdash_M (q', \omega') \Rightarrow \exists u \in \Sigma^*(\omega = u\omega' \wedge (q, u, q') \in \Delta)$
For deterministic automata $\Delta \subseteq S \times \Sigma^* \times S$ relation becomes a function $S \times \Sigma \to S$. For (q, u, q') triples $|u| = 1 \wedge (\forall q \in S \wedge \forall u \in \Sigma)\exists! q' \in S$
The language that an NFA recognizes is
$L(M) = \{\omega | (s, \omega) {\vdash_m}^* (q, \Lambda) \wedge q \in F\}$

An example NFA

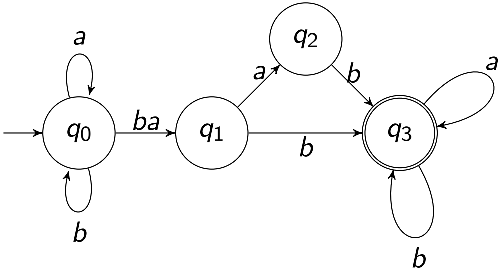
Build an NFA that recognizes languages including bab or baab as substrings.

$S = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$
$s_0 = q_0$
$F = \{q_3\}$
$\Delta = \{(q_0, a, q_0), (q_0, b, q_0), (q_0, ba, q_1), (q_1, b, q_3), (q_1, a, q_2),$
$(q_2, b, q_3), (q_3, a, q_3), (q_3, b, q_3)\}$

$M = (S, \Sigma, \Delta, s_0, F)$
$< q_0 >::= a < q_0 > |b < q_0 > |ba < q_1 >$
$< q_1 >::= b < q_3 > | \ b \ |a < q_2 >$
$< q_2 >::= b < q_3 > |b$
$< q_3 >::= a \ | \ b \ | \ a < q_3 > \ | \ b < q_3 >$

An example NFA

$< q_0 > ::= a < q_0 > | b < q_0 > | ba < q_1 >$
$< q_1 > ::= b < q_3 > | \ b \ | a < q_2 >$
$< q_2 > ::= b < q_3 > | b$
$< q_3 > ::= a | b | a < q_3 > | b < q_3 >$



A possible derivation may follow the path:
$(q_0, aaabbbaabab) \vdash$
$(q_0, aabbbaabab) \vdash$
$(q_0, abbbaabab) \vdash$
$(q_0, bbbaabab) \vdash$
$(q_0, bbaabab) \vdash$
$(q_0, baabab) \vdash (q_1, abab) \vdash$
$(q_2, bab) \vdash (q_3, ab) \vdash$
$(q_3, b) \vdash (q_3, \Lambda)$

Lemma

$M = (S, \Sigma, \Delta, s_0, F) \wedge q, r \in S \wedge x, y \in \Sigma^*$
$\exists p \in S \wedge (q, x) \vdash_M {}^*(p, \Lambda) \wedge (p, y) \vdash_M {}^*(r, \Lambda) \Rightarrow (q, xy) \vdash_M {}^*(r, \Lambda)$

Definition

Regular Grammar: All the production rules are of type-3.
Regular Language: Languages that can be recognized by regular grammars.
Regular Expression: $\varnothing, \{\Lambda\}, \{a|a \in \Sigma\}, A \vee B, A.B, A^*$
Regular set : The sets which can be represented by regular expressions are called regular sets.

Regular grammars can be represented by NFAs.

Definition

Regular Grammar: All the production rules are of type-3.
Regular Language: Languages that can be recognized by regular grammars.
Regular Expression: $\varnothing, \{\Lambda\}, \{a|a \in \Sigma\}, A \vee B, A.B, A^*$
Regular set : The sets which can be represented by regular expressions are called regular sets.

Regular grammars can be represented by NFAs.

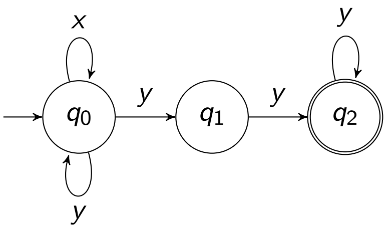
a) Non-terminal symbols are assigned to states
b) Initital state corresponds to initial symbol
c) Accepting states sorresponds to the rules that end with terminal symbols
d) If Λ should be recognized, initial state is an accepting state.

Languages recognized by finite automata(Regular Languages) are closed under union, concatanation and Kleene star operations.

Theorem (Kleene Theorem)

Every regular language can be recognized by a finite automaton and every finite automaton defines a regular language.
$M = (S, \Sigma, \Delta, s_0, F) \Leftrightarrow G = (N, \Sigma, n_0, \mapsto), L = L(G)$ *a grammar of type-3.*
$S = N \wedge F \subseteq N$
$s_0 = n_0$
$\Delta = \{(A, \omega, B) : (A \mapsto \omega B) \in \mapsto \wedge (A, B \in N) \wedge \omega \in \Sigma^*\} \cup \{(A, \omega, f_i) : (A \mapsto \omega) \in \mapsto \wedge A \in N \wedge f_i \in F \wedge \omega \in \Sigma^*\}$

Example



$< q_0 > ::= x < q_0 > \mid y < q_0 > \mid y < q_1 >$
$< q_1 > ::= y \mid y < q_2 >$
$< q_2 > ::= y < q_2 > \mid y$

Theorem (Kleene Theorem)

For every NFA an equivalent DFA can be constructed.

For the NFA $M = (S, \Sigma, \Delta, s_0, F)$ our aim is to...

(a) In $(q, u, q') \in \Delta$ there shouldn't be any $u = \Lambda$ and $|u| > 1$

(b) A transition should be present for all symbols in all states

(c) There shouldn't be more than one transitions for each configuration.

Theorem (Kleene Theorem (cont.))

Any language is regular which is constructed by applying closed language operations on regular languages. e.g. if we know L_1 and L_2 are regular languages, any language built by applying a closed operation on them produces a new regular language.

Theorem (Kleene Theorem (cont.))

Regular languages recognized by a finite automaton is closed under the following operations

(a) *Union*

(b) *Concatanation*

(c) *Kleene star*

(d) *Complement*

(e) *Intersection*

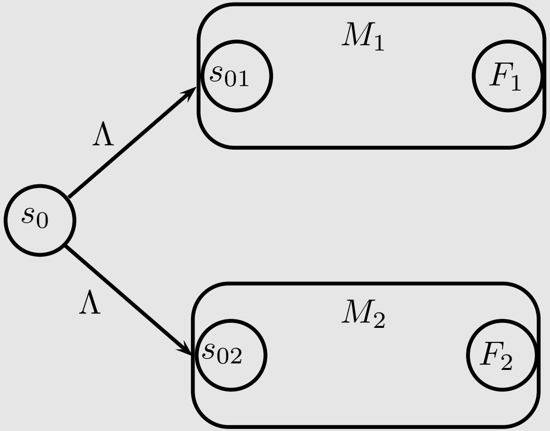
Union

$M_1 = (S_1, \Sigma, \Delta_1, s_{01}, F_1) \leftarrow L(M_1)$ Non-deterministic
$M_2 = (S_2, \Sigma, \Delta_2, s_{02}, F_2) \leftarrow L(M_2)$ Non-deterministic

$M_{=}(S, \Sigma, \Delta, s_0, F) \leftarrow L(M_1) \cup L(M_2)$ Non-deterministic

$S = S_1 \cup S_2 \cup \{s_0\}$ $F = F_1 \cup F_2$ $\Delta = \Delta_1 \cup \Delta_2 \cup \{(s_0, \Lambda, s_{01}), (s_0, \Lambda, s_{02})\}$

Union

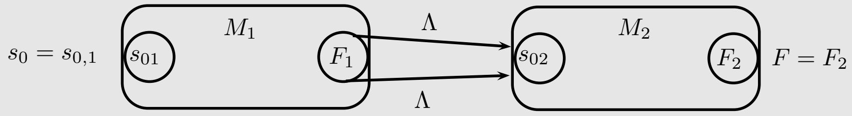


Concatanation (Non-deterministic)

$L(M_1).L(M_2) = L(M)$

$S = S_1 \cup S_2$
$s_0 = s_{01}$
$F = F_2$
$\Delta = \Delta_1 \cup \Delta_2 \cup (F_1 \times \{\Lambda\} \times \{s_{02}\})$

Concatanation

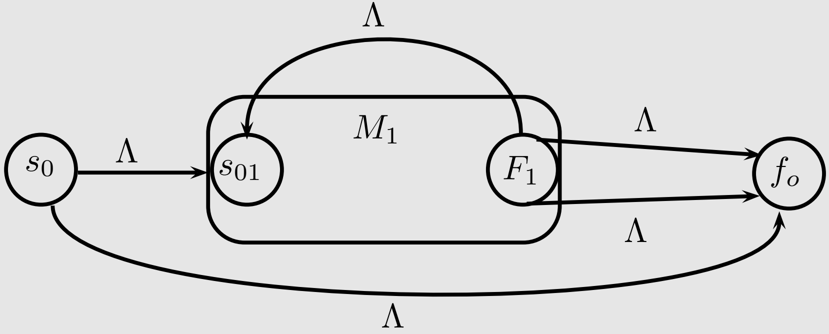


Kleene Star

$L(M_1)^* = L(M)$

$S = S_1 \cup \{s_0\}$
$F = \{f_o\}$
$\Delta = \Delta_1 \cup (F_1 \times \{\Lambda\} \times \{s_{01}\}) \cup (s_0, \Lambda, s_{01}) \cup (F_1 \times \{\Lambda\} \times F) \cup (s_0, \Lambda, f_o)$

Kleene Star



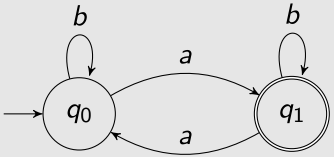
Complement (deterministic)

$\overline{L(M_1)} = L(M)$

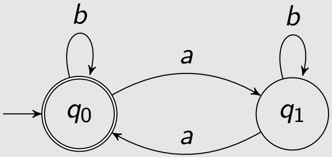
$F = \{S_1 - F_1\}$

Languages that contain odd number of a's in $\Sigma = \{a, b\}$

Complement



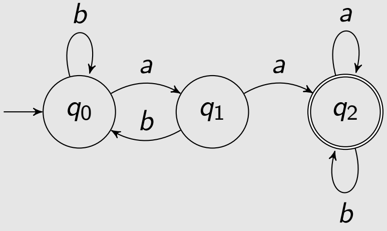
$q_1 = b^* a(b \vee ab^* a)^*$

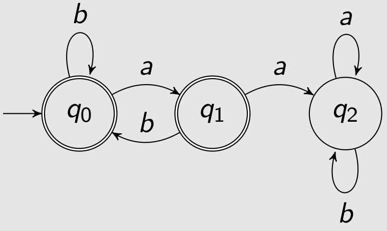


$q_0 = (b \vee ab^* a)^*$

Languages that contain at least one *aa* couple in $\Sigma = \{a, b\}$

Complement





$q_2 = (b \vee ab)^* aa(a \vee b)^*$

$q_0 \vee q_1 = (b \vee ab)^*(a \vee \Lambda)$

Intersection

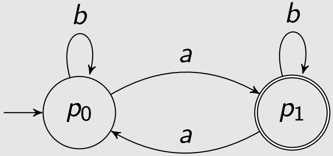
$L(M_1) \wedge L(M_2) = L(M)$

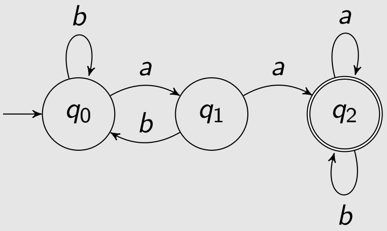
$S \subseteq S_1 \times S_2$
$s_0 = (s_{01}, s_{02})$

$F \subseteq F_1 \times F_2$ where
$[(p_1, p_2) \in F] \iff [[p_1 \in F_1] \wedge [p_2 \in F_2]]$

$\Delta \subseteq \Delta_1 \times \Delta_2$ where
$[((p_1, p_2), \sigma, (q_1, q_2)) \in \Delta] \iff [[(p_1, \sigma, q_1) \in \Delta_1] \wedge [(p_2, \sigma, q_2) \in \Delta_2]]$

Intersection





$$
\begin{aligned}
S &= \{p_0, p_1\} \\
s_0 &= \{p_0\} \\
F &= \{p_1\} \\
\Delta &= \{(p_0, b, p_0), (p_0, a, p_1), \\
&\quad (p_1, b, p_1), (p_1, a, p_0)\}
\end{aligned}
$$

$$
\begin{aligned}
S &= \{q_0, q_1, q_2\} \\
s_0 &= \{q_0\} \\
F &= \{q_2\} \\
\Delta &= \{(q_0, b, q_0), (q_0, a, q_1), \\
&\quad (q_1, b, q_0), (q_1, a, q_2), \\
&\quad (q_2, a, q_2), (q_2, b, q_2)\}
\end{aligned}
$$

Intersection

M_1 :

$$S = \{p_0, p_1\}$$
$$s_0 = \{p_0\}$$
$$F = \{p_1\}$$
$$\Delta = \{(p_0, b, p_0), (p_0, a, p_1), (p_1, b, p_1), (p_1, a, p_0)\}$$

M_2 :

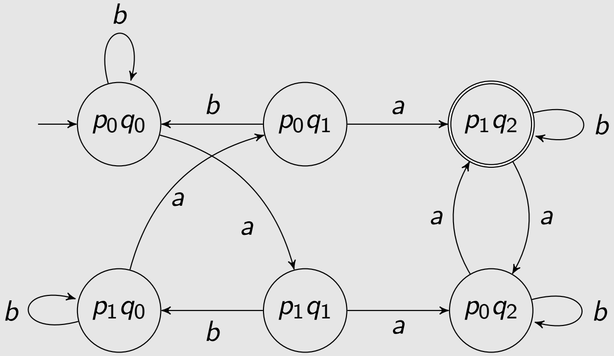
$$S = \{q_0, q_1, q_2\}$$
$$s_0 = \{q_0\}$$
$$F = \{q_2\}$$
$$\Delta = \{(q_0, b, q_0), (q_0, a, q_1), (q_1, b, q_0), (q_1, a, q_2), (q_2, a, q_2), (q_2, b, q_2)\}$$

$M_1 \cap M_2$:

$$S = \{(p_0, q_0), (p_0, q_1), (p_0, q_2), (p_1, q_0), (p_1, q_1), (p_1, q_2)\}$$
$$s_0 = \{(p_0, q_0)\}$$
$$F = \{(p_1, q_2)\}$$
$$\begin{aligned}\Delta = \{&[(p_0, q_0), b, (p_0, q_0)], [(p_0, q_1), b, (p_0, q_0)], [(p_0, q_2), b, (p_0, q_2)]\\ &[(p_1, q_0), b, (p_1, q_0)], [(p_1, q_1), b, (p_1, q_0)], [(p_1, q_2), b, (p_1, q_2)]\\ &[(p_0, q_0), a, (p_1, q_1)], [(p_0, q_1), a, (p_1, q_2)], [(p_0, q_2), a, (p_1, q_2)]\\ &[(p_1, q_0), a, (p_0, q_1)], [(p_1, q_1), a, (p_0, q_2)], [(p_1, q_2), a, (p_0, q_2)]\}\end{aligned}$$

Intersection

$M_1 \cap M_2$:



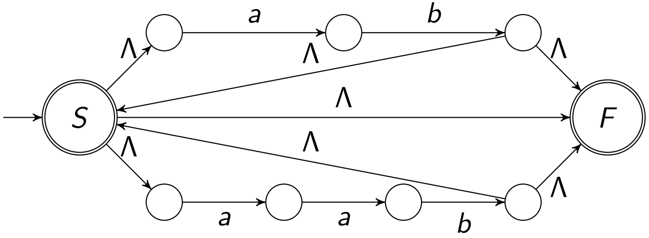
$$\begin{aligned}\Delta =\{&[(p_0,q_0),b,(p_0,q_0)],[(p_0,q_1),b,(p_0,q_0)],[(p_0,q_2),b,(p_0,q_2)]\\ &[(p_1,q_0),b,(p_1,q_0)],[(p_1,q_1),b,(p_1,q_0)],[(p_1,q_2),b,(p_1,q_2)]\\ &[(p_0,q_0),a,(p_1,q_1)],[(p_0,q_1),a,(p_1,q_2)],[(p_0,q_2),a,(p_1,q_2)]\\ &[(p_1,q_0),a,(p_0,q_1)],[(p_1,q_1),a,(p_0,q_2)],[(p_1,q_2),a,(p_0,q_2)]\}\end{aligned}$$

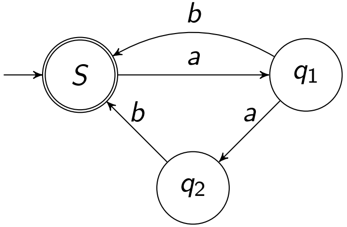
Theorem (Kleene Theorem (cont.))

Regular languages are closed under complement and intersection operations.

$$\begin{aligned} L_1 \cap L_2 &= \overline{\overline{L_1} \cup \overline{L_2}} \\ &= \Sigma^* - (\Sigma^* - L_1) \cup (\Sigma^* - L_2) \end{aligned}$$

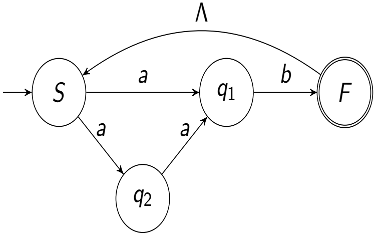
$(ab \vee aab)^*$

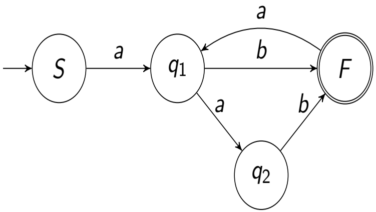




$< S >::= \Lambda \mid a < q_1 >$
$< q_1 >::= b < S > \mid a < q_2 >$
$< q_2 >::= b < S >$

$(ab \vee aab)^+ = (ab \vee aab)(ab \vee aab)^*$


$$\begin{tikzcd}
\arrow[r] & S \arrow[rr, "a"] \arrow[dr, "a"'] & & q_1 \arrow[r, "b"] & F \arrow[lll, bend right, "\Lambda"'] \\
& & q_2 \arrow[ur, "a"'] & &
\end{tikzcd}$$

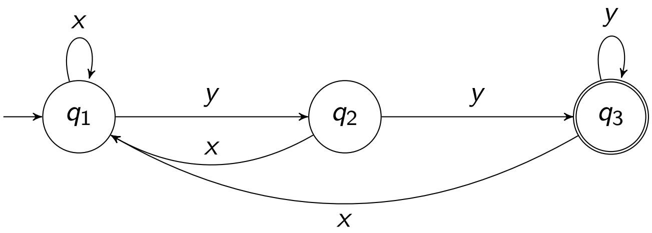

$$\begin{tikzcd}
\arrow[r] & S \arrow[r, "a"] & q_1 \arrow[rr, "b"] \arrow[dr, "a"'] & & F \arrow[ll, bend right, "a"'] \\
& & & q_2 \arrow[ur, "b"'] &
\end{tikzcd}$$

$< S >::= a < q_1 >$
$< q_1 >::= b|b < F > |a < q_2 >$
$< q_2 >::= b|b < F >$
$< F >::= a < q_1 >$

Systematic way to find the regular language recognized by a DFA

Remember the theorem that states the one and only solution to the equation $X = XA \cup B \quad \wedge \quad \Lambda \notin A$ is $X = BA^*$.
Let's rewrite the statement using regular expressions:
$x = xa \vee b \quad \wedge \quad \Lambda \neq a \Rightarrow x = ba^*$
We shall use this theorem in finding the regular language recognized by a DFA

Example



$q_1 = q_1x \vee q_2x \vee q_3x \vee \Lambda$
$q_2 = q_1y$
$q_3 = q_2y \vee q_3y$
We can use the theorem for q_3
$(q_3) = (q_3)y \vee q_2y$ than we have $q_3 = q_2yy^* \Rightarrow q_3 = q_1yy^+$

Example

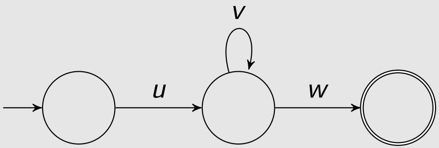
$(q_3) = (q_3)y \vee q_2y$ than we have $q_3 = q_2yy^* \Rightarrow q_3 = q_1yy^+$
Using this equality:
$q_1 = q_1x \vee q_1yx \vee q_1yy^+x \vee \Lambda$
$q_1 = q_1(x \vee yx \vee yy^+x) \vee \Lambda$
$q_1 = (x \vee yx \vee yy^+x)^* = (y^*x)^*$
$q_3 = (y^*x)^*yy^+$

Pumping Lemma

Theorem

If L is a regular language with unrestricted word length than in this language we can build any word longer than n using substrings u, v, w in following form: uv^iw. The following conditions should apply

- i $|uv| \leq n$
- ii $v \neq \Lambda$
- iii $\forall x \in L \quad x = uv^iw \ \wedge \ i \geq 0$



Proof

Each regular language L, can be recognized by a deterministic finite automaton.

- $M = \{S, \Sigma, \delta, s_0, F\}$
- $|S| = n$

$X = \sigma_1\sigma_2 \dots \sigma_\ell$, is a word of L and $|X| = \ell > n$

$(s_0, \sigma_1\sigma_2 \dots \sigma_\ell) \vdash_M (s_1, \sigma_2 \dots \sigma_\ell) \vdash_M \dots \vdash_M (s_{\ell-1}, \sigma_\ell) \vdash_M (s_\ell, \Lambda)$ ve $s_\ell \in F$

If $\ell > n \wedge |S| = n$ than according to the pigeonhole principle $\exists(i, j) : s_i = s_j \wedge 0 \leq i < j \leq n$[4] $\wedge\, i \neq j$

[4]If there is more than one couple, the couple with minimum $|i - j|$ should be selected.

Proof

In this case for our couple of states s_i, s_j
$\sigma_1\sigma_2\ldots\sigma_i = u \; \sigma_i\ldots\sigma_j = v$ ve $\sigma_j\ldots\sigma_\ell = w$

X can be written as $X = uvw$; this makes $X_0 = uw$ and
$X_m = uv^m w \;\; (m \in \mathbb{N}^+)$ strings to be recognized by M

If $|X_0| = |uw| \geq n$ than proof can be repeated by using uw instead of X. The diagrammatic representation of M is a connected graph which makes the longest length of a path at most n.

$$(s_0, uv^m w) \vdash_M^* (s_i, v^m w) \vdash_M^* (s_i, v^{m-1} w) \vdash_M^* (s_i, w) \vdash_M^* (s_\ell, \Lambda)$$

Proving Using Pumping Lemma

A proof using the pumping lemma that L cannot be accepted by a finite automaton is a proof by contradiction.

1. We assume, for the sake of contradiction, that L can be accepted by M, an FA with n states.
2. We try to select a string in L with length at least n so that statements in Theorem lead to a contradiction.

If we don't get a contradiction, we haven't proved anything, and so we look for a string x that will produce one.

There doesn't exist any language which doesn't satisfy "Pumping lemma".[5]

[5] One way implication

Example 1

$$L(M) = a^n b^n | n \in \mathbb{N}^+$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Choose $x = a^n b^n$. Then $x \in L$ and $|x| \geq n$.

By pumping lemma

- $x = uvw$ and $|uv| \leq n$
- We can get a contradiction by using any number i other than 1 for $uv^i w$ and $v = a^m$

For example, the string $uv^2 w$, is $a^{n+m} b^n$, obtained by inserting m additional a's into the first part of x. This is a contradiction, because the pumping lemma says $uv^2 w \in L$, but $n + m \neq n$.

Example 2

$$L(M) = a^{i^2} | i \geq 0$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Choose $x = a^{n^2}$

By pumping lemma

- $x = uvw$ and $0 < |v| \leq n$
- $n^2 = |uvw| < |uv^2w| = n^2 + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$
- $|uv^2w|$ must be i^2 for some integer i, but there is no integer i whose square is strictly between n^2 and $(n+1)^2$

Example 3

$$L(M) = \{a^s \mid s \text{ is a prime number } \}$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Let's choose $x = a^p a^q a^r$ and $p, q \geq 0 \;\; r > 0$
- Assume $p + q + r$ is prime

By pumping lemma

- $x = uvw$ ve $0 < |v| \leq n$
- $uv^2w = a^p a^q a^r$
- For $x = uv^{(p+q+r+1)}w$ and $|x| = p + (p + q + r + 1)q + r = (q + 1)(p + q + r)$ is not a prime number.

Pushdown Automata

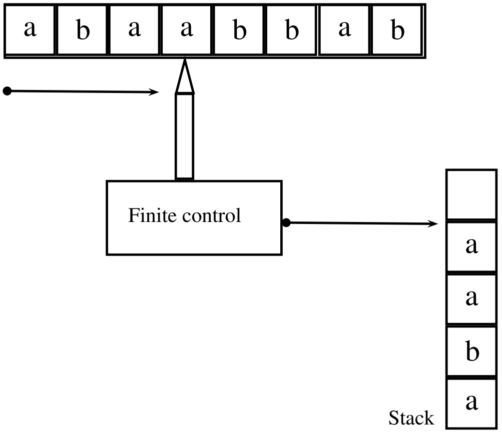
Context-Free Languages

It is not possible to design finite automata for every context-free language. For instance the recognizer for the language $\omega\omega^R | \omega \in \Sigma^*$ should contain a memory. We can design a pushdown automaton for every context-free language.

Pushdown Automata

A pushdown automaton is similar in some respects to a finite automaton but has an auxiliary memory that operates according to the rules of a stack. The default mode in a pushdown automaton (PDA) is to allow nondeterminism, and unlike the case of finite automata, the nondeterminism cannot always be eliminated.

Pushdown Automata



PDAs are not deterministic. Input strip is only used to read input while the stack can be written and read from.

Formal Definition of a PDA

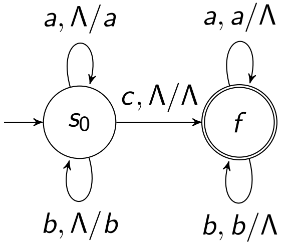
A pushdown automaton (PDA) is a 6-tuple $M = (S, \Sigma, \Gamma, \delta, s_0, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- Γ: Stack alphabet
- $s_0 \in S$: An initial state, an element of S.
- δ: The state-transition relation
  $\delta \subseteq (S \times \Sigma \cup \{\Lambda\} \times \Gamma \cup \{\Lambda\}) \times (S \times \Gamma^*)$
- F: The set of final states where $F \subseteq S$.

An example

$(\omega c \omega^R | \omega \in \{a, b\}^*)$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$S = \{s_0, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$
$\delta = \{[(s_0, a, \Lambda), (s_0, a)], [(s_0, b, \Lambda), (s_0, b)], [(s_0, c, \Lambda), (f, \Lambda)],$
$[(f, a, a), (f, \Lambda)], [(f, b, b), (f, \Lambda)]\}$

state	tape	stack	trans. rule
s_0	abb **c** bba	Λ	[(s_0,a,Λ),(s_0,a)]
s_0	bb **c** bba	a	[(s_0,b,Λ),(s_0,b)]
s_0	b **c** bba	ba	[(s_0,b,Λ),(s_0,b)]
s_0	**c** bba	bba	[(s_0,c,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



An example

state	tape	stack	trans. rule
s	abb **c** bba	Λ	[(s,a,Λ),(s,a)]
s	bb **c** bba	a	[(s,b,Λ),(s,b)]
s	b **c** bba	ba	[(s,b,Λ),(s,b)]
s	**c** bba	bba	[(s,c,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	

$G = (N, \Sigma, n_0, \mapsto)$
$N = \{S\}$
$\Sigma = \{a, b, c\}$
$n_0 = S$
$< S >::= a < S > a \mid b < S > b \mid c$

Definitions

Push: To add a symbol to the stack $[(p, u, \Lambda), (q, a)]$

Pop: To remove a symbol from the stack $[(p, u, a), (q, \Lambda)]$

Configuration: An element of $S \times \Sigma^* \times \Gamma^*$. For instance (q, xyz, abc) where a is the top of the stack, c is the bottom of the stack. Instantaneous

description (to yield in one step):
Let $[(p, u, \beta), (q, \gamma)] \in \delta$ and $\forall x \in \Sigma^* \wedge \forall \alpha \in \Gamma^*$

$(p, ux, \beta\alpha) \vdash_M (q, x, \gamma\alpha)$
Here u is read from the input tape and β is read from the stack while γ is written to the stack.

Definitions

$(p, ux, \beta\alpha) \vdash_M (q, x, \gamma\alpha)$

Let ${\vdash_M}^*$ be the reflexive transitive closure of $\vdash_M$ and let $\omega \in \Sigma^*$ and s_0 be the initial state. For M automaton to accept ω string:

$(s, \omega, \Lambda){\vdash_M}^*(p, \Lambda, \Lambda)$ and $p \in F$
$C_0 = (s, \omega, \Lambda)$ and $C_n = (p, \Lambda, \Lambda)$ where $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_{n-1} \vdash_M C_n$

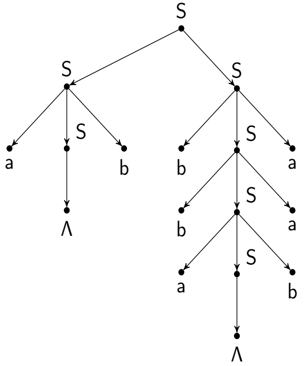
This operation is called *computation* of automaton M, this computation invloves n steps.

Let $L(M)$ be the set of string accepted by M.
$L(M) = \{\omega | (s, \omega, \Lambda){\vdash_M}^*(p, \Lambda, \Lambda) \wedge p \in F\}$

Example 1

$\omega \in \{\{a,b\}^*|\#(a) = \#(b)\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta = \{[(s,\Lambda,\Lambda),(q,c)],[(q,a,c),(q,ac)],[(q,a,a),(q,aa)],$
$[(q,a,b),(q,\Lambda)],[(q,b,c),(q,bc)],[(q,b,b),(q,bb)],\ [(q,b,a),(q,\Lambda)],[(q,\Lambda,c),(f,\Lambda)]\}$

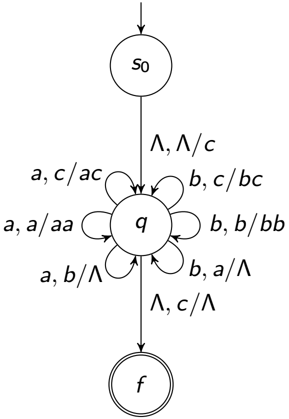
state	tape	stack	trans. rule
s	abbbabaa	Λ	[(s,Λ,Λ),(q,c)]
q	abbbabaa	c	[(q,a,c),(q,ac)]
q	bbbabaa	ac	[(q,b,a),(q,Λ)]
q	bbabaa	c	[(q,b,c),(q,bc)]
q	babaa	bc	[(q,b,b),(q,bb)]
q	abaa	bbc	[(q,a,b),(q,Λ)]
q	baa	bc	[(q,b,b),(q,bb)]
q	aa	bbc	[(q,a,b),(q,Λ)]
q	a	bc	[(q,a,b),(q,Λ)]
q	Λ	c	[(q,Λ,c),(f,Λ)]
f	Λ	Λ	



Example 1

$\omega \in \{\{a,b\}^*|\#(a)=\#(b)\}$
$M=(S,\Sigma,\Gamma,\delta,s_0,F)$
$\delta=\{[(s,\Lambda,\Lambda),(q,c)],[(q,a,c),(q,ac)],[(q,a,a),(q,aa)],$
$[(q,a,b),(q,\Lambda)],[(q,b,c),(q,bc)],[(q,b,b),(q,bb)],\ [(q,b,a),(q,\Lambda)],[(q,\Lambda,c),(f,\Lambda)]\}$

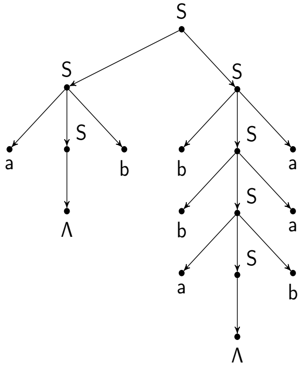
state	tape	stack	trans. rule
s	abbbabaa	Λ	[(s,Λ,Λ),(q,c)]
q	abbbabaa	c	[(q,a,c),(q,ac)]
q	bbbabaa	ac	[(q,b,a),(q,Λ)]
q	bbabaa	c	[(q,b,c),(q,bc)]
q	babaa	bc	[(q,b,b),(q,bb)]
q	abaa	bbc	[(q,a,b),(q,Λ)]
q	baa	bc	[(q,b,b),(q,bb)]
q	aa	bbc	[(q,a,b),(q,Λ)]
q	a	bc	[(q,a,b),(q,Λ)]
q	Λ	c	[(q,Λ,c),(f,Λ)]
f	Λ	Λ	



Example 1

$\omega \in \{\{a, b\}^* | \#(a) = \#(b)\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta = \{[(s, \Lambda, \Lambda), (q, c)], [(q, a, c), (q, ac)], [(q, a, a), (q, aa)],$
$[(q, a, b), (q, \Lambda)], [(q, b, c), (q, bc)], [(q, b, b), (q, bb)], [(q, b, a), (q, \Lambda)], [(q, \Lambda, c), (f, \Lambda)]\}$

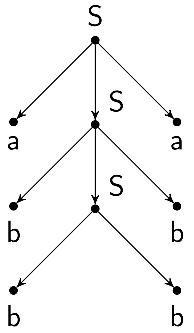
$G = (N, \Sigma, n_0, \mapsto)$
$N = \{S\}$
$\Sigma = \{a, b\}$
$n_0 = S$
$< S >::= a < S > b \mid b < S > a \mid < S >< S >$
$\mid \Lambda$



Example 2

$\omega \in \{xx^R | x \in \{a, b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s, a, \Lambda), (s, a)], [(s, b, \Lambda), (s, b)], [(s, \Lambda, \Lambda), (f, \Lambda)], [(f, a, a), (f, \Lambda)], [(f, b, b), (f, \Lambda)]\}$

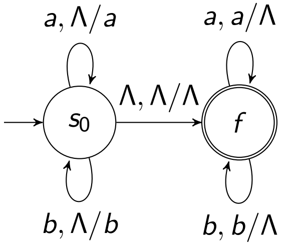
state	tape	stack	trans. rule
s	abbbba	Λ	[(s,a,Λ),(s,a)]
s	bbbba	a	[(s,b,Λ),(s,b)]
s	bbba	ba	[(s,b,Λ),(s,b)]
s	bba	bba	[(s,Λ,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



Example 2

$\omega \in \{xx^R | x \in \{a, b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s, a, \Lambda), (s, a)], [(s, b, \Lambda), (s, b)], [(s, \Lambda, \Lambda), (f, \Lambda)], [(f, a, a), (f, \Lambda)], [(f, b, b), (f, \Lambda)]\}$

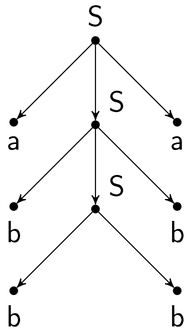
state	tape	stack	trans. rule
s	abbbba	Λ	[(s,a,Λ),(s,a)]
s	bbbba	a	[(s,b,Λ),(s,b)]
s	bbba	ba	[(s,b,Λ),(s,b)]
s	bba	bba	[(s,Λ,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



Example 2

$\omega \in \{xx^R | x \in \{a, b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s, a, \Lambda), (s, a)], [(s, b, \Lambda), (s, b)], [(s, \Lambda, \Lambda), (f, \Lambda)], [(f, a, a), (f, \Lambda)], [(f, b, b), (f, \Lambda)]\}$

$G = (N, \Sigma, n_0, \mapsto)$
$N = \{S\}$
$\Sigma = \{a, b\}$
$< S > ::= a < S > a \mid b < S > b \mid aa \mid bb$



Deterministic PDA

Deterministic PDA

1) $\forall s \in S \wedge \forall \gamma \in \Gamma$ if $\delta(s, \Lambda, \gamma) \neq \varnothing \Rightarrow \delta(s, \sigma, \gamma) = \varnothing ; \forall \sigma \in \Sigma$
2) If $a \in \Sigma \cup \{\Lambda\}$ then $\forall s, \forall \gamma$ and $\forall a$ $\text{Card}(\delta(s, a, \gamma)) \leq 1$

- (1) If there exists a transition accepting lambda from the input string no other transitions should be present for any other input. (2) There should be a unique transition for any (state,symbol,stack symbol) tuple
- For nondeterministic PDA, the equivalence problem to deterministic PDA is proven to be undecidable[6].
- For instance $\omega\omega^R$ can be accepted by a non-deteministic PDA but there doesn't exist any deterministic PDA that accepts this language.

[6] An undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct yes-or-no answer

Chomsky Hierarchy

Type	Language (Grammars)	Form of Productions in Grammar	Accepting Device
0	Recursively enumerable (unrestricted)	$\alpha \rightarrow \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*$, α contains a variable)	Turing machine
1	Context-sensitive	$\alpha \rightarrow \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert$, α contains a variable)	Linear-bounded automaton
2	Context-free	$A \rightarrow \alpha$ $(A \in N, \alpha \in (N \cup \Sigma)^*)$	Pushdown automaton
3	Regular	$A \rightarrow aB, A \rightarrow \Lambda$ $(A, B \in N, a \in \Sigma^*)$	Finite automaton

- Questions about the strings generated by a context-free grammar G are sometimes easier to answer if we know something about the form of the productions.
- Sometimes this means knowing that certain types of productions never occur, and sometimes it means knowing that every production has a certain simple form.
- For example, suppose we want to know whether a string x is generated by G, and we look for an answer by trying all derivations.
  If we don't find a derivation that produces x, how long do we have to keep trying?

Definition

A context-free grammar is said to be in *Chomsky normal form*(CNF) if every production is of one of these two types:

- $A \rightarrow BC$ (where B and C are non-terminals)
- $A \rightarrow \sigma$ (where σ is a terminal symbol)

Theorem

For every context-free grammar G, there is another CFG G_1 in Chomsky normal form such that $L(G_1) = L(G) - \{\Lambda\}$.

CNF Transformation

Following algorithm that can be used to construct the CNF grammar G_1 from a Type-2 grammar G:

1. Eliminate unit productions **and then** Λ productions.
2. Break-down productions whose right side has at least two terminal symbols.
3. Replace each production having more than two non-terminal occurrences on the right by an equivalent set of double-non-terminal productions.

Example CNF Transformation

CNF Transformation

CNF grammar G_1 from a Type-2 grammar G:

A Type-2 Grammar

$S = a, b, c$
$N = S, T, U, V, W$
$\mapsto = \{$
S $\rightarrow$ TU | V
T $\rightarrow$ aTb | Λ
U $\rightarrow$ cU | Λ
V $\rightarrow$ aVc | W
W $\rightarrow$ bW | Λ
$\}$
which generates the language $a^i b^j c^k | i = j$ or $i = k$.

CNF Transformation

1. Eliminate unit productions **and then** Λ productions.
   Unit production ex.: $V \rightarrow W$
   Λ production ex.: $T \rightarrow \Lambda$

A Type-2 Grammar

S → TU | V
T → aTb | Λ
U → cU | Λ
V → aVc | W
W → bW | Λ

Unit productions eliminated

S → TU | aVc | bW | Λ
T → aTb | Λ
U → cU | Λ
V → aVc | bW | Λ
W → bW | Λ

CNF Transformation

1. Eliminate unit productions **and then** Λ productions.
   Unit production ex.: $V \rightarrow W$
   Λ production ex.: $T \rightarrow \Lambda$

A Type-2 Grammar

S → TU | aVc | bW
T → aTb | Λ
U → cU | Λ
V → aVc | bW | Λ
W → bW | Λ

Λ productions eliminated

S → TU | aVc | bW | aTb | ab | cU | c | b | ac
T → aTb | ab
U → cU | c
V → aVc | bW | b | ac
W → bW | b

CNF Transformation

2. Break-down productions whose right side has at least two terminal symbols. Introduce for every terminal symbol σ a variable X_σ and a production rule $X_\sigma \rightarrow \sigma$

A Type-2 Grammar

S $\rightarrow$ TU | aTb | aVc | cU
S $\rightarrow$ ab | ac | c | b | bW
T $\rightarrow$ aTb | ab
U $\rightarrow$ cU | c
V $\rightarrow$ aVc | ac | bW | b
W $\rightarrow$ bW | b

Non-single-terminals eliminated

S $\rightarrow$ TU | X_aTX_b | X_aVX_c | X_cU
S $\rightarrow$ X_a X_b | X_a X_c | c | b | X_bW
T $\rightarrow$ X_aTX_b | X_a X_b
U $\rightarrow$ X_cU | c
V $\rightarrow$ X_aVX_c | X_a X_c | X_bW | b
W $\rightarrow$ X_bW | b
$X_a \rightarrow$ a
$X_b \rightarrow$ b
$X_c \rightarrow$ c

CNF Transformation

3 Replace each production having more than two non-terminal occurrences on the right by an equivalent set of double-non-terminal productions.

A Type-2 Grammar

$S \rightarrow TU \mid X_aTX_b \mid X_cU \mid X_aVX_c$
$S \rightarrow X_a\ X_b \mid X_a\ X_c \mid c \mid b \mid X_bW$
$T \rightarrow X_aTX_b \mid X_a\ X_b$
$U \rightarrow X_cU \mid c$
$V \rightarrow X_aVX_c \mid X_a\ X_c \mid X_bW \mid b$
$W \rightarrow X_bW \mid b$
$X_a \rightarrow a$
$X_b \rightarrow b$
$X_c \rightarrow c$

Non-double-non-terminals elim.

$S \rightarrow TU \mid X_a\ Y_1 \mid X_cU \mid X_a\ Y_2$
$Y_1 \rightarrow TX_b$
$Y_2 \rightarrow VX_c$
$S \rightarrow X_a\ X_b \mid X_a\ X_c \mid X_bW \mid b \mid c$
$T \rightarrow X_a\ Y_1 \mid X_a\ X_b$
$U \rightarrow X_cU \mid c$
$V \rightarrow X_a\ Y_2 \mid X_a\ X_c \mid X_bW \mid b$
$W \rightarrow X_bW \mid b$
$X_a \rightarrow a$
$X_b \rightarrow b$
$X_c \rightarrow c$

Consider the following languages. Can you design a PDA to recgonize them.

Example 1

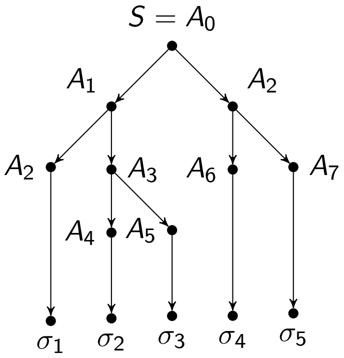
$L(G_1) = \{a^n b^n c^n | n \geq 0\}$

Example 2

$L(G_2) = \{xx \mid x \in \{a, b\}^*\}$

Some languages require more capable memory architectures than a stack to be recognized!

Pumping Lemma for Context-Free Languages



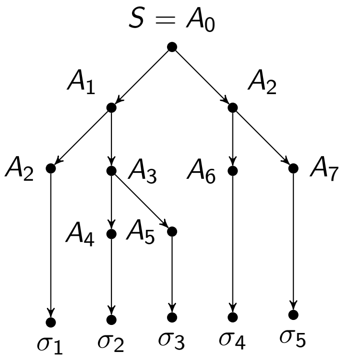
Chomsky Normal Form

The parse tree of a CNF defined context free language is a binary tree. When the leaves of the parse tree is traversed from left to right a word of the language is formed.

Theorem

Gor every context fee grammar G, there is another grammar in CNF such as $L(G_1) = L(G) - \Lambda$. In G_1 grammar rules there exists either a couple of non-terminals or a single terminal.

Pumping Lemma for Context-Free Languages



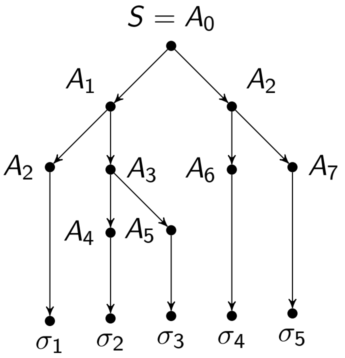
Theorem

Let's assume we produce the string $u = \sigma_1\sigma_2 \ldots \sigma_i \ldots \sigma_n \quad (\sigma_i \in \Sigma)$ for a parse tree produced by a CNF grammar such as G_{CNF}.

If we write u using only the non-terminals that directly produce terminals $u = A_1A_2 \ldots A_n$

e.g. $S = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ ya da
e.g. $S = A_2A_4A_5A_6A_7$

Pumping Lemma for Context-Free Languages



The parse tree of the grammar can be a complete binary tree for the most extreme case. In such a situation all the paths that navigate to the leaves of this tree are equals to the depth of this tree[7]. In a complete binary tree if the depth is p than the number of leaves is equal to 2^p.

In a CNF grammar's parse tree each leaf is a single child of a non-terminal parent increasing the tree's depth by one ($h = p + 1$). On the other hand the length of the produced word is the number of leaves in a complete binary tree of depth p, which is ($|w| = 2^{h-1} = 2^p$).

Tree depth: The longest path from a leaf to the r

Pumping Lemma for Context-Free Languages

In that case we can at most produce a word of length 2^p from a tree of depth $p+1$. In this tree

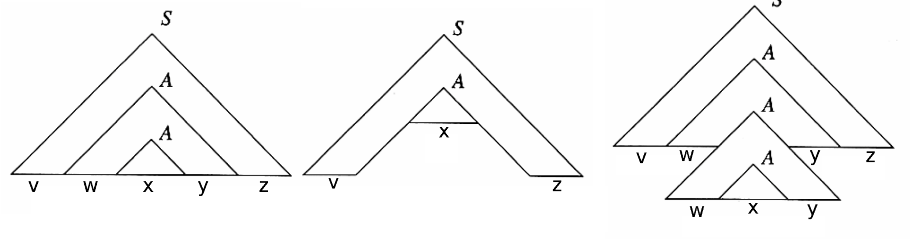
1. If each inner node does not correspond to a different non-terminal, in other words there exists a rule repetition, there exists substrings in the word that can be repeated (pumped).
2. If each inner node corresponds to a different non-terminal then there will be a repetition when producing a word of length larger than 2^p.

Theorem

Let L be a context-free language. For all strings(u) that satisfy $u \in L$ and $|u| \geq n$, string u can be written as $u = vwxyz$ where v, w, x, y, and z satisfies the following:

1. $|wy| > 0$
2. $|wxy| \leq n = 2^p$
3. For all $m \geq 0$, $vw^m xy^m z \in L$.

Pumping Lemma for Context-Free Languages



Example 1

$$L = \{a^m b^m c^m | m \geq 0\}$$

Assumptions:

- L has a CFG
- $u = vwxyz$ and $|u| \geq n$.
- n corresponds to the number of non-terminals that produce this string.
- $|wxy| \leq n$

$$\begin{array}{ccccc} a^{n-1} & a & b^{n-1} & b & c^n \\ v & w & x & y & z \end{array} \qquad \begin{array}{c} vw^2xy^2z = a^{n+1}b^nc^n \\ vxz = a^{n-1}b^{n-2}c^n \end{array}$$

Example 2

$$L = \{xx \mid x \in \{a,b\}^*\}$$

Assumptions:

- L has a CFG
- $u = a^n b^n a^n b^n$ and $|u| \geq n$
- n corresponds to the number of non-terminals.
- $|wxy| \leq n$

1.
$$\begin{array}{cccccc} a^n b^3 & b & b^{n-4} & a & a^{n-1} b^n & a^n b^3 b^{n-4} a^{n-1} b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$

2. or
$$\begin{array}{cccccc} a^n b & b & b^{n-3} & b & a^n b^n & a^n b^{n-2} a^n b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$

3. or
$$\begin{array}{cccccc} a^{n-1} & a & b^{n-3} & b & b^2 a^n b^n & a^{n-1} b^{n-1} a^n b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$

Example 3

$$L = \{x \in \{a, b, c\}^* \mid \#(a) < \#(b) \text{ and } \#(a) < \#(c)\}$$

Assumptions:

- L has a *CFG*
- $u = a^n b^{n+1} c^{n+1}$ and $|u| \geq n$
- n corresponds to the number of non-terminals.
- $|wxy| \leq n$

1. $$\begin{array}{cccccc} a^{n-1} & a & b^{n-3} & b & b^3 c^{n+1} & a^{n+1} b^{n+2} c^{n+1} \\ v & w & x & y & z & vw^2xy^2z \end{array}$$
2. or $$\begin{array}{cccccc} a^n b^5 & b & b^{n-5} & c & c^n & a^n b^n c^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$

Classical automata (a.k.a. language recognizers(DFA,NFA,PDA)) sometimes is incapable of recognizing very simple language(e.g. $a^n b^n c^n : n \geq 0$).
There exists more general language recognizers which perform transformation between chains of tokens. Turing machine is an example to this kind of language recognizers.

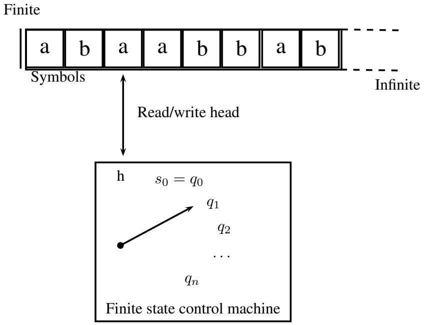
Church-Turing Thesis

In computability theory, the Church-Turing thesis is a combined hypothesis about the nature of functions whose values are effectively calculable; or, in more modern terms, functions whose values are algorithmically computable. In simple terms, the Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.
Informally the Church-Turing thesis states that if some method (algorithm) exists to carry out a calculation, then the same calculation can also be carried out by a Turing machine (as well as by a recursively definable function, and by a λ-function).

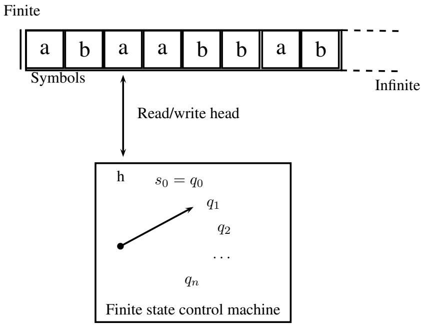
Classical automata (a.k.a. language recognizers(DFA,NFA,PDA)) sometimes is incapable of recognizing very simple language(e.g. $a^n b^n c^n : n \geq 0$).
There exists more general language recognizers which perform transformation between chains of tokens. Turing machine is an example to this kind of language recognizers.

Church-Turing Thesis

The Church-Turing thesis is a statement that characterizes the nature of computation and cannot be formally proven. Even though the three processes mentioned above proved to be equivalent, the fundamental premise behind the thesis - the notion of what it means for a function to be effectively calculable - is "a somewhat vague intuitive one". Thus, the "thesis" remains a conjecture.



Read/write head reads the corresponding symbol on the tape and according to the state of the control machine it either writes a new symbol, or it moves left(L) or right(R). If the machine attempts to move further left from the leftmost symbol on the tape than this situation is denoted as the "machine hangs". Head may move as much as possible towards right.



Symbols:

- h: End of computation. Machine halts. Halt is not included in input, output and state alphabet.
- #: blank symbol
- L,R: Symbols indicating left and right movement. Those are not included in input or output alphabet.
- Input symbols are written on the leftmost side by convention.

Formal Definition of a TM

A turing machine(TM) is a quadruple $M = (S, \Sigma, \delta, s_0)$, where:

- S: A finite, non-empty set of states. Usually $h \notin S$.
- Σ: Input/output alphabet. $\# \in \Sigma$ but $L, R \notin \Sigma$
- $s_0 \in S$: An initial state, an element of S.
- δ: The state-transition function $S \times \Sigma \rightarrow S \cup \{h\} \times (\Sigma \cup \{L, R\})$

$q \in S \wedge a \in \Sigma \wedge \delta(q, a) = (p, b)$
$q \rightarrow p$

i) (Read)$a \rightarrow b \in \Sigma$ (write on tape, in place of a)
ii) $b = L$ (head left)
iii) $b = R$ (head right)

Formal Definition of a TM

We can match modern computers with Turing Machines; RAM can be matched with the tape, computer programs can be matched with the state table, microprocessor(excluding I/O untis) can be matched with control unit of the TM. In order to improve understandibility of the state table, following notion can be used:

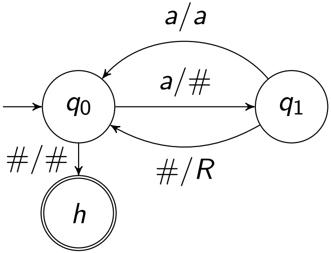
$q, \sigma, q', \sigma', HM$ where

- q stands for current state of the machine
- q' stands for the next state
- σ stands for read symbol
- σ' stands for symbol to be written (same with σ for no write)
- HM stands for head move where $HM \in -1 : L, 0 : None, 1 : R$

Example 1

A machine that erases all the symbols from left to right.
$S = \{q_0, q_1\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \#)$
q_0	$\#$	$(h, \#)$
q_1	a	$(q_0, a)^1$
q_1	$\#$	(q_0, R)


$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a/\#"] \arrow[d, "\#/\#"'] & q_1 \arrow[l, bend right, "a/a"'] \arrow[l, bend left, "\#/R"] \\ & h & \end{tikzcd}$$

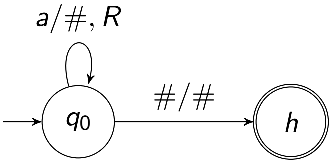
^{1}:This row is to conform to the formal definition of a function

Example 1

A machine that erases all the symbols from left to right.
$S = \{q_0, q_1\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

Now the same machine with a different notion and fewer lines

q	σ	$\delta(q, \sigma), HM$
q_0	a	$(q_0, \#), 1$
q_0	$\#$	$(h, \#), 0$

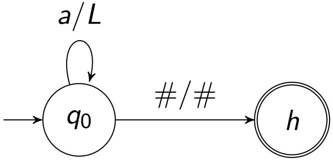


Example 2

A machine that moves towards left and recognizes *a* symbols, halting with the first # symbol

$S = \{q_0\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

q	σ	$\delta(q, \sigma)$
q_0	a	(q_0, L)
q_0	$\#$	$(h, \#)$



Configuration of a TM

A configuration is an element of the following set:
$S \cup \{h\} \times \Sigma^* \times \Sigma \times (\Sigma^*(\Sigma - \{\#\}) \cup \{\Lambda\})$

For instance(assume the head is on the underlined symbol):
(q, aba, a, bab) or $(q, aba\underline{a}bab)$
$(h, \#\#, \#, \#a)$ or $(h, \#\#\underline{\#}\#a)$
(q, Λ, a, aba) or $(q, \Lambda\underline{a}aba)$ or $(q, \underline{a}aba)$
$(q, \#a\#, \#, \Lambda)$ or $(q, \#a\#\underline{\#}\Lambda)$ or $(q, \#a\#\underline{\#})$

Following situation doesn't conform with configuration definition:
$(q, baa, a, bc\#)$ or $(q, baa\underline{a}bc\#)$

Configuration of a TM

Yields in one step for TM

$(q_1, \omega_1, a_1, u_1) \vdash_M (q_2, \omega_2, a_2, u_2)$
Here $\delta(q_1, a_1) = (q_2, b)$ and $b \in \Sigma \cup \{L, R\}$

1) $b \in \Sigma, \omega_2 = \omega_1, u_2 = u_1, a_2 = b$ (a_2 is written over a_1)
2) $b = L, \omega_1 = \omega_2 a_2; (a_1 = \# \wedge u_1 = \Lambda \Rightarrow u_2 = \Lambda) \vee (a_1 \neq \# \vee u_1 \neq \Lambda \Rightarrow u_2 = a_1 u_1)$. Left movement: If the head is on a blank and no more symbols to the right blank is erased. If the head is on a symbol or some symbols exist on the right the situation is preserved.
3) $b = R, \omega_2 = \omega_1 a_1; (u_1 = a_2 u_2) \vee (u_1 = \Lambda \Rightarrow u_2 = \Lambda \wedge a_2 = \#)$. Right movement: If there are no more symbols on the right blank is written.

Example 3

$\omega, u \in \Sigma^*; a, b \in \Sigma$ (u cannot end with $\#$)

$$\delta(q_1, a) = (q_2, b) \Rightarrow (q_1, \omega\underline{a}u) \vdash_M (q_2, \omega\underline{b}u)$$

$$\begin{aligned}\delta(q_1, a) = (q_2, L) \Rightarrow\ & i)(q_1, \omega b\underline{a}u) \vdash_M (q_2, \omega\underline{b}au) \\ & ii)(q_1, \omega b\underline{\#}) \vdash_M (q_2, \omega\underline{b})\end{aligned}$$

$$\begin{aligned}\delta(q_1, a) = (q_2, R) \Rightarrow\ & i)(q_1, \omega\underline{a}bu) \vdash_M (q_2, \omega a\underline{b}u) \\ & ii)(q_1, \omega\underline{a}) \vdash_M (q_2, \omega a\underline{\#})\end{aligned}$$

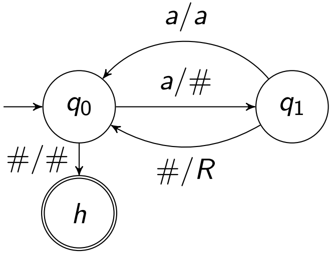
n steps computation

A computation of length n

A computation of length n or in other words n steps computation can be defined as:
$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots C_{n-1} \vdash_M C_n$

Example 1: A machine that erases all the symbols from left to right.

q	σ	$\delta(q,\sigma)$
q_0	a	$(q_1, \#)$
q_0	$\#$	$(h, \#)$
q_1	a	(q_0, a)
q_1	$\#$	(q_0, R)



$(q_0, \underline{a}aaa) \vdash_M (q_1, \underline{\#}aaa) \vdash_M (q_0, \#\underline{a}aa) \vdash_M (q_1, \#\underline{\#}aa) \vdash_M (q_0, \#\#\underline{a}a) \vdash_M$
$(q_1, \#\#\underline{\#}a) \vdash_M (q_0, \#\#\#\underline{a}) \vdash_M (q_1, \#\#\#\underline{\#}) \vdash_M (q_0, \#\#\#\#\underline{\#}) \vdash_M (h, \#\#\#\#\underline{\#})$

Turing Computable Function

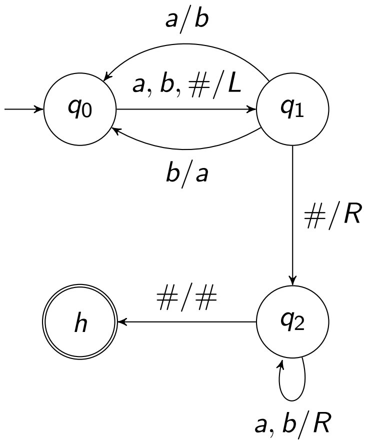
Turing Computable Function

Turing computable functions can perform transformations over character strings.
$M = (S, \Sigma, \delta, s_0)$
Let's distinguish Σ_I as input alphabet and Σ_O as output alphabet;
$f(\omega) = u \wedge \omega \in {\Sigma_I}^* \wedge u \in {\Sigma_O}^* \subseteq \Sigma^*$ and
$(s_0, \#\omega\underline{\#}) \vdash_M^* (h, \#u\underline{\#}), \# \notin \Sigma_I \wedge \# \notin \Sigma_O$

Example 4

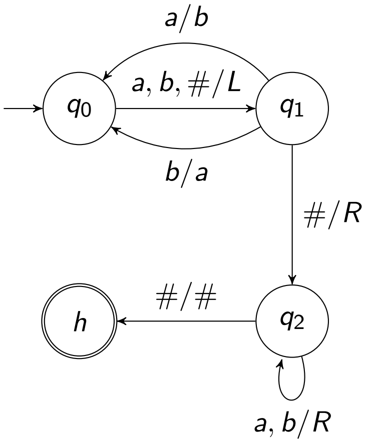
A machine that inverses strings (writes b in place of a ; a in place of b)
$S = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b, \#\}$
$s_0 = q_0$

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, L)
q_0	b	(q_1, L)
q_0	$\#$	(q_1, L)
q_1	a	(q_0, b)
q_1	b	(q_0, a)
q_1	$\#$	(q_2, R)
q_2	a	(q_2, R)
q_2	b	(q_2, R)
q_2	$\#$	$(h, \#)$


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "{a, b, \#/L}"] & q_1 \arrow[l, bend right, "a/b"'] \arrow[l, bend left, "b/a"] \arrow[d, "\#/R"] \\
& h & q_2 \arrow[l, "\#/\#"'] \arrow[loop below, "{a, b/R}"]
\end{tikzcd}$$

Example 4

A machine that inverses strings (writes b in place of a ; a in place of b)



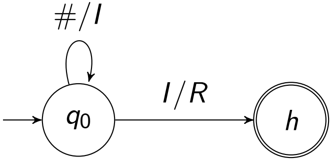
$$
\begin{aligned}
&(q_0, \#aab\underline{\#}) \vdash_M (q_1, \#aa\underline{b}) \vdash_M \\
&(q_0, \#aa\underline{a}) \vdash_M (q_1, \#a\underline{a}a) \vdash_M \\
&(q_0, \#a\underline{b}a) \vdash_M (q_1, \#\underline{a}ba) \vdash_M \\
&(q_0, \#\underline{b}ba) \vdash_M (q_1, \underline{\#}bba) \vdash_M \\
&(q_2, \#\underline{b}ba) \vdash_M (q_2, \#b\underline{b}a) \vdash_M \\
&(q_2, \#bb\underline{a}) \vdash_M (q_2, \#bba\underline{\#}) \vdash_M \\
&(h, \#bba\underline{\#})
\end{aligned}
$$

Example 5

Let's represent a number n with n I symbols like $III \dots I$. Let's build a machine that computes $f(n) = n + 1$

$S = \{q_0\}$
$\Sigma = \{I, \#\}$
$s_0 = q_0$

q	σ	$\delta(q, \sigma)$
q_0	I	(h, R)
q_0	$\#$	(q_0, I)



a) $(q_0, \#II\underline{\#}) \vdash_M (q_0, \#II\underline{I}) \vdash_M (h, \#III\underline{\#})$

b) $(q_0, \#I^n\underline{\#}) \vdash_M (h, \#I^{n+1}\underline{\#})$

c) $(q_0, \#\#) \vdash_M (q_0, \#\underline{I}) \vdash_M (h, \#I\underline{\#})$

Turing Decidable Machine

Turing Decidable Machine

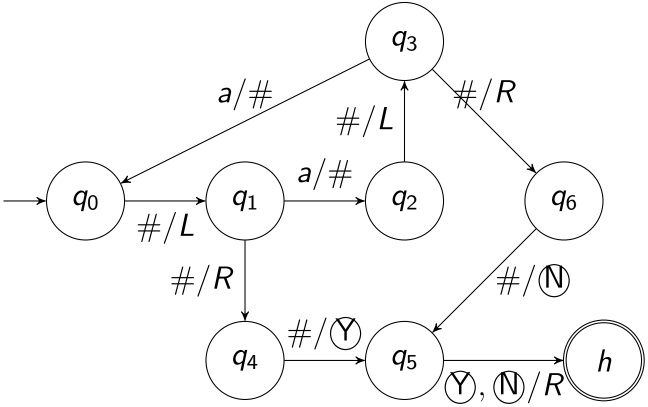
Let Σ_0 be our alphabet and $\# \notin \Sigma_0$ and L be a language $L \subseteq {\Sigma_0}^*$
If we can compute the function x_L like:

$$\forall \omega \in \Sigma_0^*, F_L(\omega) = \begin{cases} \textcircled{Y} \Rightarrow \omega \in L \\ \textcircled{N} \Rightarrow \omega \notin L \end{cases}$$

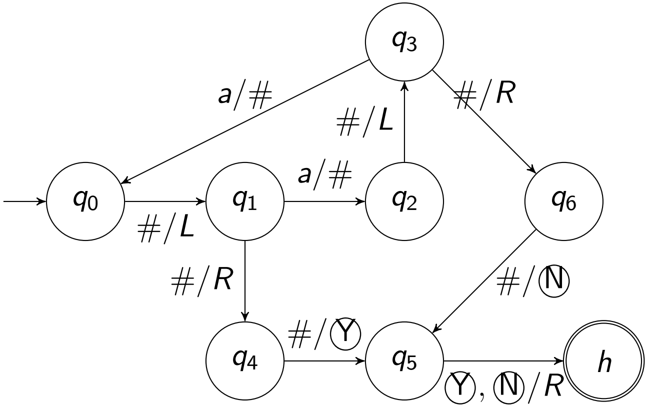
Turing Decidable Machine

Example:
$\Sigma_0 = \{a\}; L = \{\omega \in {\Sigma_0}^* | \ ||\omega|| \text{even number}\}$
$S = \{q_0, \ldots, q_6\}$
$\Sigma = \{a, \textcircled{Y}, \textcircled{N}, \#\}$
$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	$\#$	(q_1, L)
q_1	a	$(q_2, \#)$
q_1	$\#$	(q_4, R)
q_2	$\#$	(q_3, L)
q_3	a	$(q_0, \#)$
q_3	$\#$	(q_6, R)
q_4	$\#$	$(q_5, \textcircled{Y})$
q_5	$\textcircled{Y}$	(h, R)
q_5	$\textcircled{N}$	(h, R)
q_6	$\#$	$(q_5, \textcircled{N})$

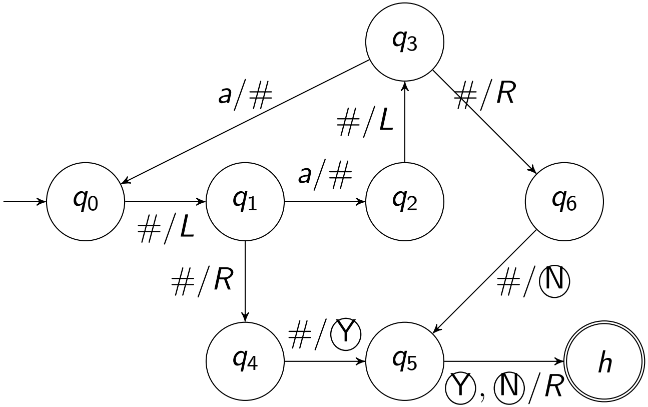


Turing Decidable Machine



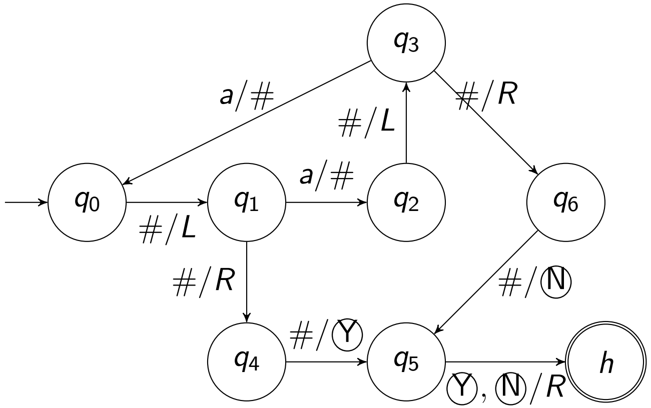
a) $(q_0, \#aa\underline{\#}) \vdash_M (q_1, \#a\underline{a}) \vdash_M (q_2, \#a\underline{\#}) \vdash_M$
$(q_3, \#\underline{a}) \vdash_M (q_0, \#\underline{\#}) \vdash_M (q_1, \underline{\#}) \vdash_M$
$(q_4, \#\underline{\#}) \vdash_M (q_5, \#\underline{\textcircled{Y}}) \vdash_M (h, \#\textcircled{Y}\underline{\#})$

Turing Decidable Machine



b) $(q_0, \#aaa\underline{\#}) \vdash_M (q_1, \#aa\underline{a}) \vdash_M (q_2, \#aa\underline{\#}) \vdash_M$
$(q_3, \#a\underline{a}) \vdash_M (q_0, \#a\underline{\#}) \vdash_M (q_1, \#\underline{a}) \vdash_M$
$(q_2, \#\underline{\#}) \vdash_M (q_3, \underline{\#}) \vdash_M (q_6, \#\underline{\#}) \vdash_M$
$(q_5, \#\underline{\textcircled{N}}) \vdash_M (h, \#\textcircled{N}\underline{\#})$

Turing Decidable Machine



a) $(q_0, \#a^n\underline{\#}) \vdash_M{}^* (h, \#\textcircled{Y}\underline{\#}) \Rightarrow n$ is even

b) $(q_0, \#a^n\underline{\#}) \vdash_M{}^* (h, \#\textcircled{N}\underline{\#}) \Rightarrow n$ is odd

Example I

$$M \text{ decides } A = \{0^{2^n} | n \geq 0\}$$

For string w

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to step 1.

Example - DIY

M decides $A = \{\omega\#\omega | \omega \in \{0,1\}^*\}$

Example II

$$M \text{ decides } A = \{a^i b^j c^k | i \times j = k \text{ and } i, j, k \geq 1\}$$

For string w

1. Scan the input from left to right to determine whether it is a member of $a^+ b^+ c^+$ and reject if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, reject.
4. Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, accept; otherwise, reject.

Example III

M decides

$$A = \{\#x_1\#x_2\#\ldots\#x_\ell |\text{ each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$$

For string w

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, accept. If that symbol was a #, continue with the next step. Otherwise, reject.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so accept
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, reject.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so accept.
5. Go to step 3.

Example IV

Hilbert's Tenth Problem

Devise a process according to which it can be determined by a finite number of operations if a multivariable polynomial has integral(integer) roots.

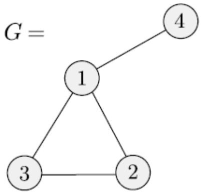
For now, let's consider a polynomial over a single variable and define the machine M as
On input p: where p is a polynomial over the variable x.

1. Evaluate p with x set successively to the values 0, 1, -1, 2, -2, 3, -3,... If at any point the polynomial evaluates to 0, accept.

M is a recognizer, but not a decider. We can make M a decider by defining a range that the roots can reside in.

For a single variable polynomial this range is inside $\pm k\frac{c_{\max}}{c_1}$, however for a multivariable polynomial it is not possilbe to calculate such a range.

Example V - Graph Connectivity



$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

M decides $A = \{\langle G \rangle | G$ is a connected undirected graph$\}$

On input $\langle G \rangle$, the encoding of graph G:

1. Select the first node of G and mark it.
2. Repeat the following step until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, accept; otherwise, reject.

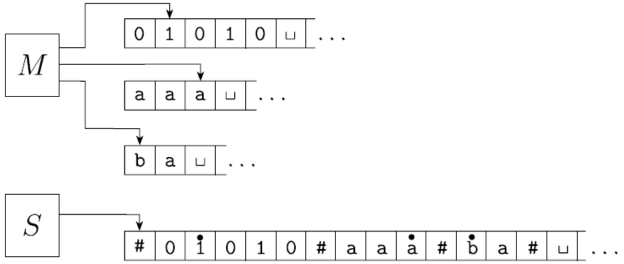
- It is possible to characterize the set of functions that can be computed by a Turing machine. Functions in this set are called Turing computable.
- Just about every reasonable model of computation anyone has come up with is Turing complete; that is, it can compute exactly the same set of functions as the Turing machine.
- Some of these models, like lamdba calculus, are very different from a Turing machine, so their equivalence is surprising.
- This observation led to the Church-Turing Thesis, which is essentially a definition of what it means to be computable.

Definition

From now on, we continue to speak of Turing machines, but our real focus from now on is on algorithms.
That is, the Turing machine merely serves as a precise model for the definition of algorithm.
We do not spend much time on the low-level programming of Turing machines.
We need only to be comfortable enough with Turing machines to believe that they capture all algorithms.

Recursively Enumerable Languages

Multi-tape Turing Machine



Theorem

Every multitape Turing machine has an equivalent single-tape Turing machine. Two machines are equivalent if they recognize the same language.

Non-deterministic Turing Machine

Definition

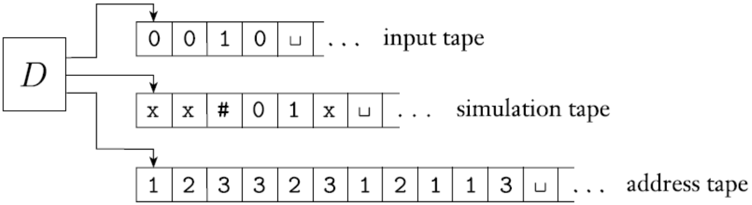
A non-deterministic Turing Machine acts using a transition relation rather than a transition function over the possible set of state pairs.

Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Idea

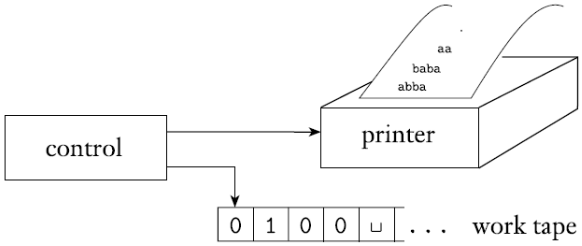
We can simulate any nondeterministic TM N with a deterministic TM D. The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate.


D
0 0 1 0 ␣ . . . input tape
x x # 0 1 x ␣ . . . simulation tape
1 2 3 3 2 3 1 2 1 1 3 ␣ . . . address tape

Enumerators

Definition

Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. TM do not get inputs, instead it **generates** strings belonging to a language one by one.



When we start a Turing machine on an input it may **accept**, **reject**, or **loop**. Looping may entail any simple or complex behavior that never leads to a halting state.

Definition

A language is Turing-recognizable[a] if some Turing machine is able **accept** the words in a language.

[a]recursively enumerable language

Definition

A language is Turing-decidable[a] or simply decidable if some Turing machine is able to decide it. Turing machines that halt on all inputs and never loop are called deciders because they always make a decision to **accept** or **reject**.

[a]recursive language

Corollary

Every decidable language is Turing-recognizable.

Theorem

A language is Turing-recognizable if and only if some enumerator enumerates it.

Proof I.

First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. On input w, M does:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept.

M accepts those strings that appear on E's list. □

Theorem

A language is Turing-recognizable if and only if some enumerator enumerates it.

Proof II.

If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*. E Repeats the following for i=1,2,3,...

1. Run M for i steps on each input, $s_1, s_2, s_3, \ldots$
2. If any computations accept, print out the corresponding s_j

□

The Universal Turing Machine

Definition

A universal Turing machine(T_U) is assumed to receive an input string of the form $\langle M, \omega \rangle$, where M is an arbitrary TM, ω is a string over the input alphabet of M, and $\langle\rangle$ is an encoding function whose values are strings in $\{0, 1\}^*$. The computation performed by T_U on this input string satisfies these two properties:

1. T_U accepts the string $\langle M, \omega \rangle$ iff M accepts ω.
2. If M accepts ω and produces output x, then T_U produces output $\langle x \rangle$.

Relative Power of TM Variants

All TM models with unrestricted access to unlimited memory turn out to be equivalent in power, so long as they satisfy reasonable requirements such as the ability to perform only a finite amount of work in a single step.

Turing Complete

A computational system that can compute every Turing-computable function is called Turing-complete. Alternatively, such a system is one that can simulate a universal Turing machine. To show that something is Turing complete, it is enough to show that it can be used to simulate some Turing complete system.

Turing Equivalent

A Turing-complete system is called Turing equivalent if it computes precisely the same class of functions as do Turing machines. Alternatively, a Turing-equivalent system is one that can simulate, and be simulated by a universal Turing machine.

Theorem

If L_1 and L_2 are both recursively enumerable languages over Σ, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also recursively enumerable

Theorem

If L_1 and L_2 are both recursive languages over Σ, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also recursive

Theorem

If L is a recursive language over Σ, then its complement $\overline{L}$ is also recursive

Theorem

If L is a recursively enumerable language over Σ, and its complement $\overline{L}$ is also recursively enumerable then L is recursive.

Unrestricted Grammars

Theorem

For every unrestricted grammar G, there is a Turing machine T with $L(T) = L(G)$.

Theorem

For every Turing machine T with input alphabet Σ, there is an unrestricted grammar G generating the language $L(T) \subseteq \Sigma^*$

Context-Sensitive Grammars

Definition

A context-sensitive grammar (CSG) is an unrestricted grammar in which no production is length-decreasing. In other words, every production is of the form $\alpha \to \beta$, where $|\beta| \geq |\alpha|$

Definition

A linear-bounded automaton (LBA) is a 5-tuple that is identical to a nondeterministic Turing machine, with the following exception. There are two extra tape symbols: [and] wrapping the initial input string on the tape. During its computation, an LBA is not permitted to replace either of these brackets or to move its tape head to the left of the [or to the right of the].

Context-Sensitive Grammars

Theorem

If $L \subseteq \Sigma^*$ is a context-sensitive language, then there is a linear-bounded automaton that accepts L.

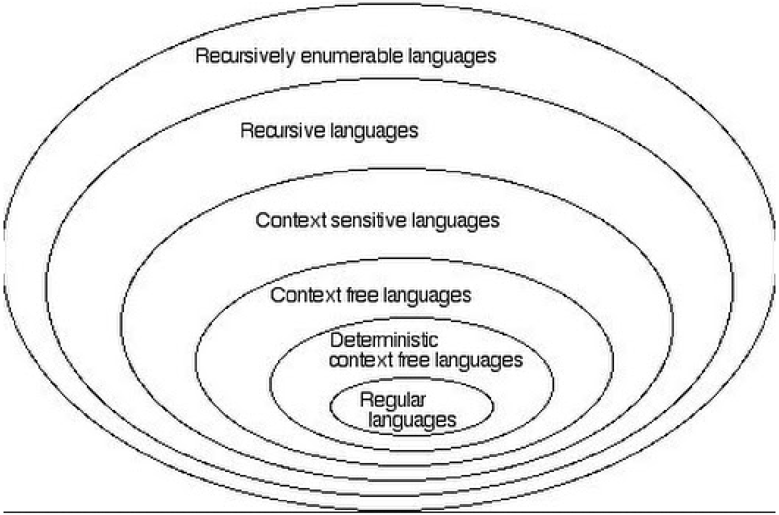
Theorem

If $L \subseteq \Sigma^*$ is accepted by a linear-bounded automaton M, then there is a context-sensitive grammar G generating $L - \{\Lambda\}$.

Theorem

Every context-sensitive language L is recursive.

Chomsky Hierarchy revisited


Recursively enumerable languages
Recursive languages
Context sensitive languages
Context free languages
Deterministic context free languages
Regular languages

References

- Sipser, M. (1996). Introduction to the Theory of Computation. ACM Sigact News, 27(1), 27-29. Chapter 3.2 - Variants of Turing Machines
- Martin, J. C. (1991). Introduction to Languages and the Theory of Computation (Vol. 4). NY: McGraw-Hill. Chapter 8 - Recursively Enumerable Languages

Decidability

Definition (The Acceptance Problem)

Testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept.

$$A_{DFA} = \{\langle B, \omega \rangle | B \text{ is a } DFA \text{ that accepts input string } \omega\}$$

Theorem

A_{DFA} is a decidable language

Proof.

$M =$ On input $\langle B, \omega \rangle$, where B is a *DFA* and ω is a string:

1. Simulate B on input ω
2. If the simulation ends in an accept state, accept . If it ends in a nonaccepting state, reject.

□

$$A_{NFA} = \{\langle B, \omega\rangle | B \text{ is a } NFA \text{ that accepts input string } \omega\}$$

Theorem

A_{NFA} is a decidable language

Proof.

$N =$ On input $\langle B, \omega\rangle$, where B is a NFA and ω is a string:

1. Convert NFA B to an equivalent DFA C
2. Run TM M from previous slide on input $\langle C, \omega\rangle$
3. If M accepts, accept ; otherwise, reject.

□

$$A_{Rex} = \{\langle R, \omega \rangle | R \text{ is a regular expression that generates string } \omega\}$$

Theorem

A_{Rex} is a decidable language

Proof.

$P =$ On input $\langle R, \omega \rangle$, where R is a regular expression and ω is a string:

1. Convert regular expression R to an equivalent *NFA* A
2. Run TM N from previous slide on input $\langle A, \omega \rangle$
3. If N accepts, accept ; otherwise, reject.

$\square$

Definition (Emptiness Testing)

Testing whether or not a finite automaton accepts any strings at all.

$$E_{DFA} = \{\langle A\rangle | A \text{ is a } DFA \text{ and } L(A) = \emptyset\}$$

Theorem

E_{DFA} is a decidable language

Proof.

```
$T$ = On input $\langle A \rangle$, where $A$ is a DFA:
  1 Mark the start state of $A$
  2 Repeat until no new states get marked:
      1 Mark any state that has a transition coming into it from any state that
        is already marked.
  3 If no accept state is marked, accept; otherwise, reject.
```

□

Definition (Automaton Equality)

Testing whether two DFAs recognize the same language is decidable.

$$EQ_{DFA} = \{\langle A, B\rangle | A \text{ and } B \text{ are } DFA\text{s and L(A)=L(B)}\}$$

Definition (Symmetric Difference of two Sets)

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

$L(C) = \emptyset$ iff $L(A) = L(B)$

Theorem

EQ_{DFA} is a decidable language

Proof.

$F =$ On input $\langle A, B\rangle$, where A and B are *DFA*s:

1. Construct *DFA* C as described
2. Run TM T from previous slides on input $\langle C\rangle$
3. If T accepts, accept. If T rejects, reject.

□

$$A_{CFG} = \{\langle G, \omega \rangle | G \text{ is a } CFG \text{ that generates string } \omega\}$$

Theorem

A_{CFG} is a decidable language

Proof.

$F =$ On input $\langle G, \omega \rangle$, where G is a CFG and ω is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, accept; if not, reject.

□

$$E_{CFG} = \{\langle G, \omega\rangle | G \text{ is a } CFG \text{ and } L(G) = \emptyset\}$$

Theorem

E_{CFG} is a decidable language

Proof.

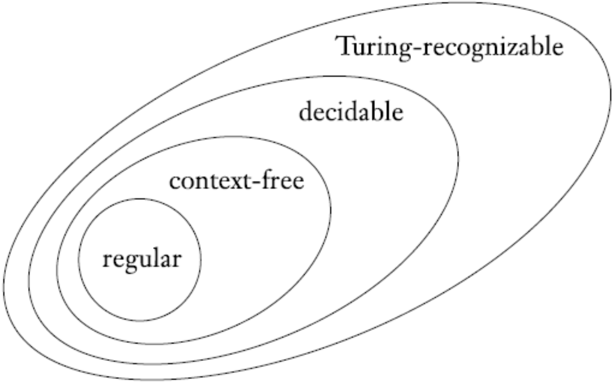
```
R = On input ⟨G⟩, where G is a CFG:
  1  Mark all terminal symbols in G.
  2  Repeat until no new variables get marked:
       1  Mark any variable A where G has a rule A → U_1 U_2 ... U_k and each
          symbol U_1, ..., U_k has already been marked.
  3  If the start variable is not marked, accept; otherwise, reject.
```

□

$$EQ_{CFG} = \{\langle G, H\rangle | G \text{ and } H \text{ are } CFG\text{s and } L(G) = L(H)\}$$

Eqaulity of CFGs

The class of context-free languages is not closed under complementation or intersection. EQ_{CFG} is not decidable. We will prove this in the future.


Turing-recognizable
decidable
context-free
regular

$$A_{TM} = \{\langle M, \omega\rangle | M \text{ is a } TM \text{ and } M \text{ accepts } \omega\}$$

Theorem

A_{TM} is undecidable.

A_{TM} is Turing-recognizable. Recognizers are more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.

Definition (Universal Turing Machine)

$U =$ On input $\langle M, \omega\rangle$, where M is a TM and ω is a string:

1. Simulate M on input ω.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject.

Countability

Set of even numbers

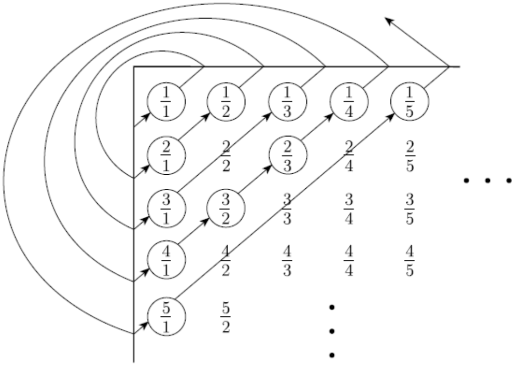
Let $\mathbb{N}$ be the set of natural numbers and let $\mathbb{E}$ be the set of even natural numbers. Georg Cantor observed that two sets have the same size if we can define a correspondence (bijection) between two sets.
Considering $f(n) = 2n$ we can conclude that $\mathbb{N}$ and $\mathbb{E}$ are the same size.

Definition

A set A is countable if either it is finite or it has the same size as $\mathbb{N}$

Set of rational numbers

Let $\mathbb{Q} = \{\frac{m}{n} | m, n \in \mathbb{N}\}$ be the set of positive rational numbers. To find a correspondence we make an infinite matrix containing all the positive rational numbers as below.



$\frac{1}{1}$ $\frac{1}{2}$ $\frac{1}{3}$ $\frac{1}{4}$ $\frac{1}{5}$
$\frac{2}{1}$ $\frac{2}{2}$ $\frac{2}{3}$ $\frac{2}{4}$ $\frac{2}{5}$
$\frac{3}{1}$ $\frac{3}{2}$ $\frac{3}{3}$ $\frac{3}{4}$ $\frac{3}{5}$
$\frac{4}{1}$ $\frac{4}{2}$ $\frac{4}{3}$ $\frac{4}{4}$ $\frac{4}{5}$
$\frac{5}{1}$ $\frac{5}{2}$

Diagonalization

Set of real numbers

Let $\mathbb{R}$ be the set of real numbers. Real numbers have decimal representations like $\pi = 3.1415926\ldots$ adn $\sqrt{2} = 1.4142135\ldots$. Cantor proved thath $\mathbb{R}$ is uncountable by introducing the diagonalization method.

Proof.

1. Suppose there is a bijection $f(n)$ between $\mathbb{N}$ and $\mathbb{R}$.
2. Construct a table between each natural number and rational number (can be in floating point representation).
3. We construct a real number x by selecting each digit as follows
4. For each natural number i in the table change the i^{th} digit in the real number to another integer. Do not choose 9's or 0's.

We have just constructed a number x which differs from every number in the so called bijection f. Contradiction. $\square$

Corollary

Some languages are not Turing-recognizable

Proof.

1. The set of all strings Σ^* is countable for any alphabet Σ.
2. The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$
3. The set of all languages $\mathbb{L}$ over alphabet Σ is uncountable.
    1. The set of all infinite binary sequences($\mathbb{B}$) is uncountable (use diagonalization).
    2. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$Each language $A \in \mathbb{L}$ has a unique *characteristic sequence* in $\mathbb{B}$ where the i^{th} bit of that sequence is 1 if $s_i \in A$.
    3. The function defined above is a bijection and hence $\mathbb{L}$ and $\mathbb{B}$ is correspondent. $\mathbb{L}$ is uncountable.
4. The set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine

□

$$A_{TM} = \{\langle M, \omega\rangle | M \text{ is a } TM \text{ and } M \text{ accepts } \omega\}$$

Theorem

A_{TM} is undecidable.

Proof.

1. We assume A_{TM} is decidable and obtain a contradiction. Let's assume that H is a decider for A_{TM} where
$$H(\langle M, \omega\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } \omega \\ \text{reject} & \text{if } M \text{ does not accept } \omega \end{cases}$$
2. Define a TM D where $D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } H \text{ rejects } \langle M, \omega\rangle \\ \text{reject} & \text{if } H \text{ accepts } \langle M, \omega\rangle \end{cases}$
3. Feed D unto itself to obtain a contradiction
$$D(\langle D\rangle) = \begin{cases} \text{accept} & \text{if } D \text{ rejects } \langle D\rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

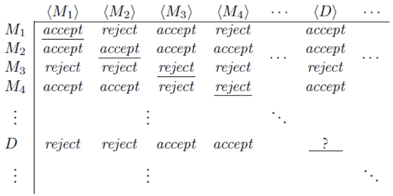
□

$$A_{TM} = \{\langle M, \omega\rangle | M \text{ is a } TM \text{ and } M \text{ accepts } \omega\}$$

Theorem

A_{TM} is undecidable.

- H accepts $\langle M, \omega\rangle$ exactly when M accepts ω
- D rejects $\langle M\rangle$ exactly when M accepts $\langle M\rangle$
- D rejects $\langle D\rangle$ exactly when D accepts $\langle D\rangle$



	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$	$\cdots$
M_1	$\underline{accept}$	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	$\underline{accept}$	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	$\underline{reject}$	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	$\underline{reject}$		*accept*	
$\vdots$		$\vdots$			$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		$\underline{\ ?\ }$	
$\vdots$		$\vdots$					$\ddots$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist

A Turing-unrecognizable Language

Theorem

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable. In other words, a language is decidable exactly when both it and its complement are Turing-recognizable.

Proof.

- Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable. If language A is decidable, both A and its complement $\overline{A}$ are Turing-recognizable.
- If both A and $\overline{A}$ are Turing-recognizable, let M_1 and M_2 be the recognizers. The following TM M is a decider for A. $M =$ On input ω
  1. Run both M_1 and M_2 on input ω in parallel.
  2. If M_1 accepts, accept; if M_2 accepts, reject.

$\square$

A Turing-unrecognizable Language

Corollary

If a Turing-recognizable language A_{TM} is not decidable then $\overline{A_{TM}}$ is not Turing-recognizable.

References

- Sipser, M. (1996). Introduction to the Theory of Computation. ACM Sigact News, 27(1), 27-29. Chapter 4 - Decidability

Computability

The necessary properties of a satisfactory formal system are

1. Completeness: It should be possible either to prove or disprove any proposition that can be expressed in the system.
2. Consistency: It should not be possible to both prove and disprove a proposition in the system.

A way of putting together the computation theory and its limits is starting from the axioms of set theory and build the system using only set relations, specifically functions.
However, later Kurt Gödel was able to express and prove the following result

Incompleteness Theory

If a formal system is consistent, then it is impossible to prove (within the system) that it is consistent. A formal system can be either complete or consistent at once.

Is it possible to distinguish between provable and unprovable propositions?

- Alan Turing: Turing machines and halting problem
- Alonzo Church: Recursive Function Theory

Primitive Recursion - Basic Functions

(a) Constant function[8] : For each $k \geq 1$ and each $a \geq 0$ the constant function $C_a^k : \mathbb{N}^k \to \mathbb{N}$ can be defined as: $C_a^k(X) = a$[9]

(b) Successor function $s : \mathbb{N} \to \mathbb{N}$ is defined as: $s(x) = x + 1$

(c) Projection function: For each $k \geq 1$ and each i where $0 \leq i \leq k$, the projection function $p_i^k : \mathbb{N}^k \to \mathbb{N}$ can be defined as: $p_i^k(x_1, x_2, \ldots, x_i, \ldots, x_k) = x_i$

[8] Zero function

[9] Capital variable represents a vector of variables

Primitive Recursion - Basic Operations

(d) Composition: $C(f, g_1, \ldots, g_m)$ denotes the unique function h such that : $h(X) = f(g_1(X), \ldots, g_m(X))$

(e) Primitive Recursion: If f has arity $k+2$ and g has arity k, for $k \geq 0$, then $R(g, f)$ denotes the unique $(k+1)$-ary function h such that:

$$h(0, Y) = g(Y);$$
$$h(Succ(x), Y) = f(h(x, Y), x, Y)$$

Primitive Recursion

Definition

A primitive recursive function is a function formed from by using the basic functions (constant, successor, projection) and by using only composition and primitive recursion.

Computation Tree

We may think of the basic functions invoked as leaves in a tree whose non-terminal nodes are labeled with C(Composition) and R(Primitive Recursion). Nodes labeled by C may have any number of daughters and nodes labeled by R always have two daughters. We may think of this tree as a program for computing the function defined this way.

Primitive Recursion

Definition

A primitive recursive function is a function formed from by using the basic functions (constant, successor, projection) and by using only composition and primitive recursion.

Primitive Recursion

We may think of the basic functions invoked as leaves in a tree whose non-terminal nodes are labeled with C(Composition) and R(Primitive Recursion). Nodes labeled by C may have any number of daughters and nodes labeled by R always have two daughters. We may think of this tree as a program for computing the function defined this way.

Theorem

Every primitive recursive function is total and computable.

Addition, Multiplication and Subtraction

Primitive Recursive Derivation

In general, we can show that a function f is primitive recursive by constructing a primitive recursive derivation: a sequence of functions $f_0, f_1, \ldots, f_j$ such that $f_j = f$ and each f_i in the sequence is an initial function, or obtained from earlier functions in the sequence by composition, or obtained from earlier functions by primitive recursion.

$$Add(x, y) = x + y$$

$$\begin{aligned} Add(x, 0) &= x = p_1^1(x) \\ Add(x, Succ(k)) &= Succ((x + k)) = s(Add(x, k)) \end{aligned}$$

Addition, Multiplication and Subtraction

Primitive Recursive Derivation

In general, we can show that a function f is primitive recursive by constructing a primitive recursive derivation: a sequence of functions $f_0, f_1, \ldots, f_j$ such that $f_j = f$ and each f_i in the sequence is an initial function, or obtained from earlier functions in the sequence by composition, or obtained from earlier functions by primitive recursion.

$$Mult(x, y) = x * y$$

$$\begin{aligned} Mult(x, 0) &= 0 = C_0^1(x) \\ Mult(x, k+1) &= x * k + x = Add(x, Mult(x, k)) \end{aligned}$$

Addition, Multiplication and Subtraction

$$Sub(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

We are going to define function predecessor in the inverse form of successor.

$$Pred(x) = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{if } x \geq 1 \end{cases}$$

or in other terms

$$\text{Pred(Succ(x))} = \text{x}$$

subtraction function can be written as

$$Sub(x, 0) = x$$
$$Sub(x, k + 1) = Pred(Sub(x, k))$$

The proper subtraction operation is often written as $\dot{-}$, and another name for it is the monus operation

Other Operations

Signature

$$sg(0) = 0$$
$$sg(x+1) = 1$$

Absolute Difference

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

Characteristic Function

$$\chi_P(x) = \begin{cases} 1 & \text{if } P(x) \text{is true} \\ 0 & \text{if } P(x) \text{is false} \end{cases}$$

Other Operations

Identitiy

$$\chi_{=}(x,y) = \overline{sg}(|x-y|)$$

Ordering

$$\chi_{>}(x,y) = sg(x \dot{-} y)$$

Min-Max

$$min(x,y) = x \dot{-} (x \dot{-} y)$$
$$max(x,y) = x + (y \dot{-} x)$$

Other Operations

Remainder

$$\begin{aligned} rm(0, y) &= 0 \\ rm(x+1, y) &= (rm(x, y) + 1) * (y > (rm(x, y) + 1)) \end{aligned}$$

Quotient

$$\begin{aligned} qt(0, y) &= 0 \\ qt(x+1, y) &= qt(x, y) + (y = rm(x, y) + 1) \end{aligned}$$

Other Operations

Divisibility

$$x|y = \overline{sg}(rm(x,y))$$

Exponentiation

$$Exp(x,0) = 1$$
$$Exp(x,y+1) = x * Exp(x,y)$$

Theorem

The two-place predicates LT, EQ, GT, LE, GE, and NE are primitive recursive. If P and Q are any primitive recursive n-place predicates, then $P \wedge Q$, $P \vee Q$, and $\neg P$ are primitive recursive.

Logical Operators

$$\begin{aligned}
\chi_{P_1 \wedge P_2} &= \chi_{P_1} * \chi_{P_2} \\
\chi_{P_1 \vee P_2} &= \chi_{P_1} + \chi_{P_2} \dot{-} \chi_{P_1 \wedge P_2} \\
\chi_{\neg P_1} &= 1 \dot{-} \chi_{P_1}
\end{aligned}$$

Finite Operations

Finite Sum

$$\sum_{x}(f_0, \ldots, f_{x-1})(y) = \sum_{z<x} f_z(y)$$

$$\sum_{z<0} f_z(y) = 0$$

$$\sum_{z<x+1} f_z(y) = \sum_{z<x} f_z(y) + f_x(y)$$

Finite Operations

Finite Product

$$\prod_{z<0} f_z(y) = 1$$

$$\prod_{z<x+1} f_z(y) = \prod_{z<x} f_z(y) * f_x(y)$$

Finite Operations

Definition by Cases

$$\sum_{z<k+1} f_z(x) \cdot P_z(x) = \begin{cases} f_1(x) & \text{if } P_1(x) \\ f_2(x) & \text{if } P_2(x) \\ \vdots & \\ f_k(x) & \text{if } P_k(x) \end{cases}$$

Bounded Quantifiers

Bounded Universal Quantifier

$$(\forall z < x)P_z(Y) = \prod_{z<x} P_z(Y)$$

Bounded Existential Quantifier

$$(\exists z < x)P_z(Y) = sg(\sum_{z<x} P_z(Y))$$

Bounded Quantifiers

Bounded Minimization

$$m_P(X,k) = \begin{cases} min\{y \mid 0 \leq y \leq k \text{ and } P(X,y)\} & \text{if this set is not empty} \\ k+1 & \text{otherwise} \end{cases}$$

$$m_P(X,k) = \overset{k}{\mu}\, y[P(X,y)]$$

The n^{th} Prime Number

The function to calculate the n^{th} Prime Number(PrNo) is primitive recursive.

1. For $n \geq 0$ let $PrNo(n)$ be the n^{th} prime number. E.g. $PrNo(0) = 2, PrNo(1) = 3, PrNo(2) = 5$ and so on.
2. $Prime(n) = (n \geq 2) \wedge \neg(\exists y \text{ s.t. } y \geq 2 \wedge y \leq n-1 \wedge Mod(n,y) = 0)$ function is true only if n is a prime.
3. If we can define a bound for every next prime, we can use bounded minimization operator to define PrNo by primitive recursion.
4. This bound exists. For every positive integer m it is proven that there is a prime number between m and m!.

Prime Number

$$PrNo(0) = 2$$
$$PrNo(k+1) = m_P(PrNo(k), PrNo(k)! + 1)$$

μ-Recursive Functions

Unbounded Minimization

$M_p(X,k) = min\{y \mid P(X,y) \text{ is true}\}$ $M_P(X,k) = \mu y[P(X,y)]$

Definition

The set of μ-recursive functions $\mathbb{M}$ is defined by including every initial function, every function derived from the initial functions by composition and primitive recursion and unbounded minimization of each function in the set.

Theorem

All μ-recursive partial functions are computable. For the undefined values in unbounded minimization operation the computation may continue forever and not give an output which is acceptable since the unbounded minimization is not defined for that case.

Gödel Numbering

- In the 1930s, the logician Kurt Gödel developed a method of "arithmetizing" a formal axiomatic system by assigning numbers to statements and formulas.
- We will use his approach to arithmetize Turing machines, to describe operations of TMs, and computations performed by TMs, in terms of purely numeric operations.
- This approach can be used to show that every Turing-computable function is μ-recursive

Gödel Numbering

Theorem

Every positive integer can be factored as a product of primes and this factorization is unique except for differences in the order of the factors.
E.g. $46 = 2^1 3^0 5^0 7^0 11^0 13^0 17^0 19^0 23^1$

Definition

For every $n \geq 1$ and every finite sequence $x_0, x_1, \ldots, x_{n-1}$ of n natural numbers, the Gödel number of the sequence is the number

$$gn(x_0, x_1, \ldots, x_{n-1}) = 2^{x_0} 3^{x_1} 5^{x_2} \ldots (PrNo(n-1))^{x_{n-1}}$$

For every $n \geq 1$, every positive integer is the Gödel number of at most one sequence of n integers

Gödel Numbering

- A TM move can be interpreted as a transformation of one TM configuration to another, and we need a way of characterizing a Turing machine configuration by a number
- The configuration of a TM is determined by the state, tape head position, and current contents of the tape, and we define the configuration number to be the number $gn(q, P, tn)$ where q is the number of the current state, P is the current head position, and tn is the current tape number
- The most important feature of the configuration number is that from it we can reconstruct all the details of the configuration.

Theorem

Every Turing-computable partial function from $\mathbb{N}^n$ to $\mathbb{N}$ is μ-recursive

Ackermann Function

- Ackermann function is an example of a function that is μ-recursive but not primitive recursive.

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x+1, 0) &= A(x, 1) \\ A(x+1, y+1) &= A(x, A(x+1, y)) \end{aligned}$$

Computability and Deciability

- Computability: A function f on a certain domain is computable if there exists a Turing machine that computes the value of f for all arguments in its domain and halts in a final state
- Problem: A set of inputs together with a question asked for each input.
- Decidability: A problem is decidable if there exists a Turing machine that can compute 'yes' or 'no' for a problem and halts in a final state.

References

- Martin, J. C. (1991). Introduction to Languages and the Theory of Computation (Vol. 4). NY: McGraw-Hill. Chapter 10 - Computable Functions
- Xavier, S. E. (2005). Theory Of Automata, Formal Languages And Computation (As Per Uptu Syllabus). New Age International. Chapter 6 - Computability

Reducibility

Undecidable Problems

Question

Are there any languages that cannot be accepted by any Turing Machine?

Answer

Yes. Because there are more languages than Turing machines to accept.
How many TM's are there? How many languages are there?

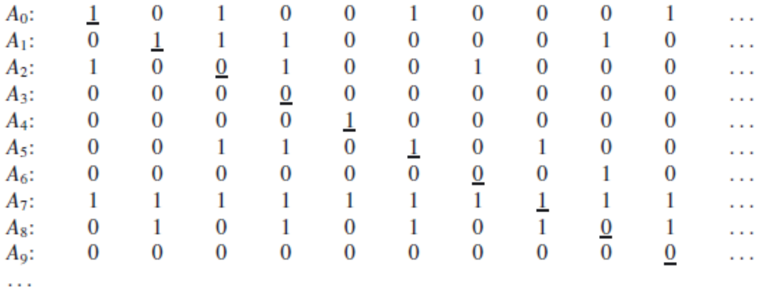
- Sets A and B are the same size if a bijection $f : A \to B$ can be defined.
- Set A is countable if a bijection $f : \mathbb{N} \to A$ can be defined.

Example

- The Set $\mathbb{N} \times \mathbb{N}$ is countable
- A Countable union of countable sets is countable
- The Set of Turing Machines with a finite input alphabet Σ is countable
- A Language for a finite alphabet Σ are countable sets
- The set of all language that can be defined for a finite alphabet Σ is uncountable. (Set of all subsets of Σ^* is uncountable)

$2^{\mathbb{N}}$ is uncountable

- There can be no list of subsets of $\mathbb{N}$ containing every subset of $\mathbb{N}$
- Let $A_i | i \in \mathbb{N}$ represent each subset from a set with cardinality $\mathbb{N}$
- Let's define $X = \{i \in \mathbb{N} \mid i \notin A_i\}$



A_0:	$\underline{1}$	0	1	0	0	1	0	0	0	1	...
A_1:	0	$\underline{1}$	1	1	0	0	0	0	1	0	...
A_2:	1	0	$\underline{0}$	1	0	0	1	0	0	0	...
A_3:	0	0	0	$\underline{0}$	0	0	0	0	0	0	...
A_4:	0	0	0	0	$\underline{1}$	0	0	0	0	0	...
A_5:	0	0	1	1	0	$\underline{1}$	0	1	0	0	...
A_6:	0	0	0	0	0	0	$\underline{0}$	0	1	0	...
A_7:	1	1	1	1	1	1	1	$\underline{1}$	1	1	...
A_8:	0	1	0	1	0	1	0	1	$\underline{0}$	1	...
A_9:	0	0	0	0	0	0	0	0	0	$\underline{0}$	...
...											

- Set X's characteristic function is obtained by inverting the underlined elements.

Reducibility

Definition

Reduction: A way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

- A: You want to find a way from main gate of ITU to BBF (Faculty of Computer)
- B: You want to find a path between two points in a small campus map.
- A reduces to B. We can use a solution to B to solve A.
- C: Find a path between two nodes in a graph.

Reducibility Examples

- A: Measuring the area of a rectangle
- B: Measuring rectangle's height and width.
- A: Solving a system of linear equations
- B: Inverting a matrix
- A: Determining if a NFA accepts a given string
- B: Determining if a FA accepts a given string
- A: Given two FAs M_1 and M_2, is $L(M_1) \subseteq L(M_2)$
- B: Find an FA M such that $L(M) = L(M_1) - L(M_2)$, is $L(M) = \emptyset$

Mapping Reducibility

Definition

Suppose P_1 and P_2 are decision problems. We say P_1 is reducible to P_2 if there is an algorithm that finds, if an arbitrary instance I is a yes-instance of P_1 if and only if $F(I)$ is a yes-instance of P_2.

If L_1 and L_2 are languages over alphabets Σ_1 and Σ_2, respectively, we say L_1 is reducible to L_2 if there is a Turing-computable function $f : {\Sigma_1}^* \rightarrow {\Sigma_2}^*$ such that for every $x \in {\Sigma_1}^*$: $x \in L_1 \Leftrightarrow f(x) \in L_2$

- When A is reducible to B, solving A cannot be harder than solving B.
- If A is reducible to B and B is decidable, A also is decidable
- If A is undecidable and reducible to B, B is undecidable

Previously on ToC

$$A_{TM} = \{\langle M, \omega \rangle | M \text{ is a } TM \text{ and } M \text{ accepts } \omega\}$$

or in other words

Accepting Problem: Given a TM T and a string ω, is $\omega \in L(T)$?

We can't just say "Execute T on input ω and see what happens", because T might loop forever on input ω.

Theorem

A_{TM} is undecidable.

The Halting Problem

$$HALT_{TM} = \{\langle M, \omega \rangle | M \text{ is a } TM \text{ and } M \text{ halts on } \omega\}$$

or in other words

Halting Problem: Given a TM T and a string ω, does T halt on input ω?

Theorem

$HALT_{TM}$ is undecidable.

The Halting Problem

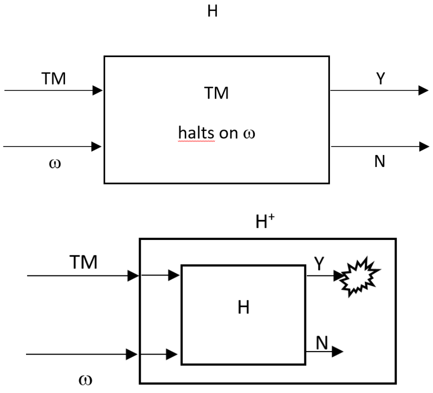
- We assume that $HALT_{TM}$ is decidable and use that assumption to show that A_{TM} is decidable, which is a contradiction.
- The key idea is to show that A_{TM} is reducible to $HALT_{TM}$

Proof.

1. Assume that TM R can decide $HALT_{TM}$.
2. We construct TM S to decide A_{TM}, with S operating as follows on input $\langle M, \omega \rangle$
    1. Run TM R on input $\langle M, \omega \rangle$
    2. If R rejects, reject
    3. If R accepts, simulate M on ω until it halts
    4. If M has accepted, accept; if M has rejected, reject
3. If R decides $HALT_{TM}$, then S decides A_{TM}.Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

□

The Halting Problem



What happens if we feed H$^+$ to H and . . .

- . . . H answers Yes, but it shouldn't because H$^+$ doesn't halt if the H inside the H$^+$ answers Yes

The Halting Problem

```
n = 4
while (n is the sum of two primes)
  n = n+2
```

- Goldbach's conjecture[10]: every even integer greater than 2 is the sum of two primes
- Can you write a program to decide if this program continues forever or not?

[10] Hasn't been proven yet

Undecidability Examples

Reduction to A_{TM} is a common method to prove undecidability except for A_{TM} itself which is proven using diagonalization. For example

Theorem

$E_{TM}\{\langle M\rangle \mid M$ is a TM and $L(M) = \emptyset\}$ is undecidable.

Theorem

$REGULAR_{TM}\{\langle M\rangle \mid M$ is a TM and $L(M)$ is a regular language$\}$ is undecidable.

Theorem

$EQ_{TM}\{\langle M_1, M_2\rangle \mid M_1$ and M_2 are TMs and $L(M_1) = L(M_2)\}$ is undecidable.

Undecidability Examples

Reduction to A_{TM} is a common method to proove undecidability except for A_{TM} itself which is proven using diagonalization. For example[11]

Theorem

A_{LBA} is decidable.

Proof idea: Number of configurations for an LBA is limited. We can say it doesn't accept if it takes longer than a limited number of steps.

[11] LBA:Linear Bounded Automaton

Undecidability Examples

Theorem

$ALL_{CFG} = \{\langle G \rangle \mid$ is a CFG and $L(G) = \Sigma^*\}$ is undecidable.

Theorem

$ISECT_{CFG} = \{\langle G_1, G_2 \rangle \mid G_1$ and G_2 are CFGs and $L(G_1) \cap L(G_2) \neq \emptyset\}$ is undecidable.

Rice's Theorem

Definition

A property R of Turing machines is called a language property if, for every Turing machine having property R, and every other TM T_1 with $L(T_1) = L(T)$, T_1 also has property R.

A language property of TMs is nontrivial if there is at least one TM that has the property and at least one that does not.

Theorem

If R is a nontrivial language property of TMs, then the decision problem

P_R*:Given a TM T, does T have property R?*

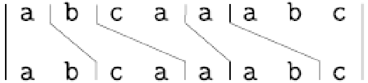
is undecidable

Post's Correspondence Problem

- We can describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side.
- The question is whether it is possible to make a horizontal line of one or more dominoes, with duplicates allowed, so that the string obtained by reading across the top halves matches the one obtained by reading across the bottom

$$\left\{ \left[\frac{\mathtt{b}}{\mathtt{ca}}\right], \left[\frac{\mathtt{a}}{\mathtt{ab}}\right], \left[\frac{\mathtt{ca}}{\mathtt{a}}\right], \left[\frac{\mathtt{abc}}{\mathtt{c}}\right] \right\}.$$

$$\left[\frac{\mathtt{a}}{\mathtt{ab}}\right]\left[\frac{\mathtt{b}}{\mathtt{ca}}\right]\left[\frac{\mathtt{ca}}{\mathtt{a}}\right]\left[\frac{\mathtt{a}}{\mathtt{ab}}\right]\left[\frac{\mathtt{abc}}{\mathtt{c}}\right]$$



Post’s Correspondence Problem

An example solvable problem:

	Up	Down
Tile 1	1	111
Tile 2	10111	10
Tile 3	10	0

An example unsolvable problem:

	Up	Down
Tile 1	10	101
Tile 2	011	11
Tile 3	101	011

Post's Correspondence Problem

Theorem

PCP is undecidable.

- We will reduce PCP to A_{TM} to show that it is undecidable.
- We will transform A_{TM} to PCP for this.
- Given $\langle M, \omega \rangle$ as an input to an A_{TM}, construct an instance of PCP
- By solving the transformed PCP, we are going to obtain an accepting history
- If we were able to obtain an accepting history without running the machine, than we would have been able to decide A_{TM}

Post's Correspondence Problem

We are going to generate tiles that represent segments of an execution history from a TM definition. Following is a step by step transformation.

1. The initial tile is determined as: $\{\vdash_I \mid \vdash_I q_o\omega\}$
2. Generate a tile from each Left or Right move
   1. Moving from state q to state r by replacing symbol a with b and moving the tape head to right corresponds to tile: $\{qa \mid br\}$
   2. Moving from state q to state r by replacing symbol a with b and moving the tape head to left corresponds to tile: $\{\sigma qa \mid r\sigma b\}$ where $\sigma \in \Sigma$. Generate a tile for each symbol in Σ
3. Generate identity tiles for each symbol in Σ. e.g. $\{a \mid a\}$, $\{b \mid b\}$, $\dots$, $\{\vdash \mid \vdash\}$
4. Generate accepting tiles : $\{q_a\sigma \mid q_a\}$, $\{\sigma q_a \mid q_a\}$ for each symbol in Σ.

Your Turn

- Model a Turing machine that accepts when a given string on tape contain odd number of 1's.
- Generate tiles to be used in PCP.
- Demonstrate your answer.

References

- Martin, J. C. (1991). Introduction to Languages and the Theory of Computation (Vol. 4). NY: McGraw-Hill. Chapter 5 - Reducibility
- Sipser, M. (1996). Introduction to the Theory of Computation. ACM Sigact News, 27(1), 27-29. Chapter 8.5 - Not every language is recursively enumerable, and Chapter 9 - Undecidable Problems

Advanced Topics in Computability

The Recursion Theorem

Is it possible to produce a self replicating machine?

- Suppose machine M produces product P
- M must contain the model of P inside
- M should be adding more complexity by containing production information
- M should be more complex than P
- A machine cannot produce itself because it is not more complex than itself.

By recursion theorem we are going to show that this inference is wrong!

Self Reference

- Let $\langle SELF \rangle$ be the self replicating machine's description.
- $\langle SELF \rangle$ is composed of two parts A and B such as $\langle AB \rangle$
- The job of A is to print out a description of B, and conversely the job of B is to print out a description of A
- Part A runs first and upon completion passes control to B
- Let function $q : \Sigma^* \rightarrow \Sigma^*$ be the function, where if ω is any string, $q(\omega)$ is the description of a Turing machine P_ω that prints out ω and then halts
- Part A is a Turing machine that prints out $\langle B \rangle$. We can obtain Part A by $q(\langle B \rangle)$.

- Our description of A depends on having a description of B.
- We might be tempted to define B by $q(\langle A \rangle)$, but that doesn't make sense.
- Doing so would define B in terms of A, which in turn is defined in terms of B.
- B computes A from the output that A produces
- If B could obtain $\langle B \rangle$ than $q(\langle B \rangle)$ would have given $\langle A \rangle$.
- B can obtain $\langle B \rangle$ by reading the freshly added string on the tape after A finishes.
- Then after B computes $q(\langle B \rangle) = \langle A \rangle$

In summary, when $SELF = AB$ runs

1. First A runs. It prints $\langle B \rangle$ on the tape.
2. B starts. It looks at the tape and finds its input, $\langle B \rangle$
3. B calculates $q(\langle B \rangle) = \langle A \rangle$ and combines that with $\langle B \rangle$ into a TM description $\langle SELF \rangle$
4. B prints this description and halts.

Theorem

- *To make a Turing machine that can obtain its own description and then compute with it, we need only make a machine, called T in the statement, that receives the description of the machine as an extra input.*
- *Then the recursion theorem produces a new machine R, which operates exactly as T does but with R's description filled in automatically.*

Recursion theorem states that the following h function is computable.

$$f(0, y) = g(y)$$
$$f(x + 1, y) = h(f(x, y), x, y)$$

Recursion theorem make the following possible: Recursive functions, computer viruses, reflective programming

Logical Functions

1. $\forall q \exists p \forall x, y [p < q \land (x, y > 1 \rightarrow xy \neq p)]$
2. $\forall a, b, c, n [(a, b, c > 0 \land n > 2) \rightarrow a^n + b^n \neq c^n]$
3. $\forall q \exists p \forall x, y [p > q \land (x, y > 1 \rightarrow (xy \neq p \land xy \neq p + 2))]$

Models and Theories

A model consists of a universe-domain and logical formulas. For example, for the following formula
$\forall x \forall y [R(x,y) \wedge R(y,x)]$, a model can be defined as $M = (\mathbb{N}, \leq)$.
Another example can be the formula $\forall y \exists x [R(x,x,y)]$ and the model $M = (\mathbb{R}, a + b == c)$

Definition

If M is a model, we let theory of M, written $Th(M)$, be the collection of true sentences in the language of the model.

Decidability of Logical Theories

Theorem

$Th((N),+)$ is decidable.

Theorem

$Th((N),+,\times)$ is undecidable.
Some true statement(s) in $Th((N),+,\times)$ is(are) not provable.

Turing Reducibility

Definition

An oracle for a language B is an external device that is capable of reporting whether any string ω is a member of B.

An oracle Turing machine is a modified Turing machine that has the additional capability of querying an oracle.

We write M^B to describe an oracle Turing machine that has an oracle for language B

Turing Reducibility

Consider an oracle for A_{TM}. Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input. It can also decide E_{TM}, in the following way, assuming $\langle M \rangle$ is provided as an input:

1. Construct the following TM, N:
   a. Run M in parallel on all strings in Σ^*
   b. If M accepts any of these srings, accept.
2. Query the oracle to determine whether $\langle N, 0 \rangle \in A_{TM}$
3. If the oracle answers NO, accept; if YES, reject.

E_{TM} is decidable relative to A_{TM}

Turing Reducibility

Definition

Language A is Turing reducible to language B if A is decidable relative to B.

If A is Turing reducible to B and B is decidable, then A is decidable.

An oracle Turing machine with an oracle for A_{TM} is very powerful. It can solve many problems that are not solvable by ordinary Turing machines. But even such a powerful machine cannot decide all languages

Information

Which string contains more information?

A=0101010101010101010101010101010101010101
B=1110010110100011101010000111010011010111

Sequence A contains little information because it is merely a repetition of the pattern 01 twenty times. In contrast, sequence B appears to contain more information.

We can represent a binary string x by a Turing machine M and its input ω, binary encoded on a tape as $\langle M, \omega \rangle$

Information

Definition

- Let x be a binary string.
- The minimal description of x, written $d(x)$, is the shortest string $\langle M, \omega \rangle$ where TM M on input ω halts with x on its tape.
- If several such strings exist, select the lexicographically first among them.
- The descriptive complexity of x, is written as $K(x) = |d(x)|$

Information

Definition

- Let x be a binary string.
- The minimal description of x, written $d(x)$, is the shortest string $\langle M, \omega \rangle$ where TM M on input ω halts with x on its tape.
- If several such strings exist, select the lexicographically first among them.
- The descriptive complexity of x, is written as $K(x) = |d(x)|$

Theorem

For a specific string x, $K(x)$ has an upper bound.

Compressibility

Definition

- Let x be a binary string.
- x is c-compressible if $K(x) \leq |x| - c$
- If x is not c-compressible, we say that x is incompressible by c
- If x is incompressible by 1, we say that x is incompressible.

Theorem

Incompressible strings of every length exist.

Compressibility

Theorem

Incompressible strings of every length exist.

Proof.

- The number of binary strings of length n is 2^n
- Each description is a binary string, so the number of descriptions of length less than n is at most the sum of the number of strings each length up to $n-1$ which is equal to $2^n - 1$
- The number of short descriptions is less than the number of strings of length n.
- Therefore, at least one string of length n is incompressible.

□

Compressibility

- The K measure of complexity is not computable
- No algorithm can decide in general whether strings are incompressible
- We have no way to obtain long incompressible strings and would have no way to determine whether a string is incompressible even if we had one.
- Any incompressible string of length n has roughly an equal number of 0s and 1s
- The length of an incompressible string's longest run of 0s is approximately $\log 2n$

Computability Examples

Cellular Automata

- A cellular automaton is a model of a world with very simple physics
- "Cellular" means that the space is divided into discrete chunks, called cells.
- Automata are governed by rules that determine how the system evolves in time.
- Time is divided into discrete steps, and the rules specify how to compute the state of the world during the next time step based on the current state.

Cellular Automata

- Consider a cellular automaton (CA) with a single cell. A 0-dimensional CA.
- The state of the cell is an integer represented with the variable x_i
- where the subscript i indicates that x_i is the state of the system during time step i
- As an initial condition, $x_0 = 0$
- As a rule, let's pick $x_i = x_{i-1} + 1$
- After each time step, the state of the CA gets incremented by 1
- We have a simple CA that performs a simple calculation: it counts.

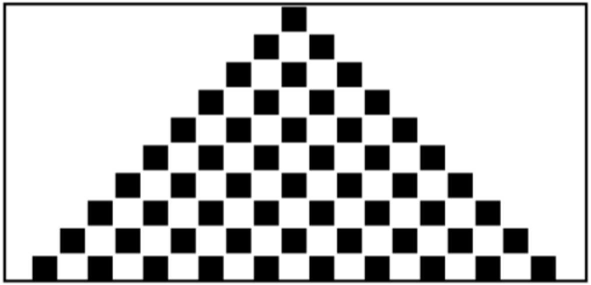
Cellular Automata

- Most CAs are deterministic, which means that rules do not have any random elements; given the same initial state, they always produce the same result.
- 1-dimensional CAs are more interesting. In the early 1980s Stephen Wolfram published a series of papers presenting a systematic study of 1-dimensional CAs.
- Cells are arranged in a contiguous space so that some of them are considered "neighbors".
- In a basic 1-dimensional CA, the cells have two states, denoted 0 and 1, so the rules can be summarized by a table that maps from the state of the neighborhood to the next state for the center cell.

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

Wolfram suggested reading the bottom row as a binary number. In this case $(00110010)_2 = 50$ so "Rule 50."

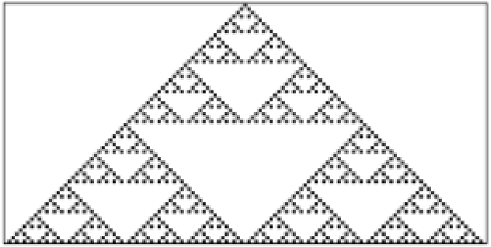
Cellular Automata



prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

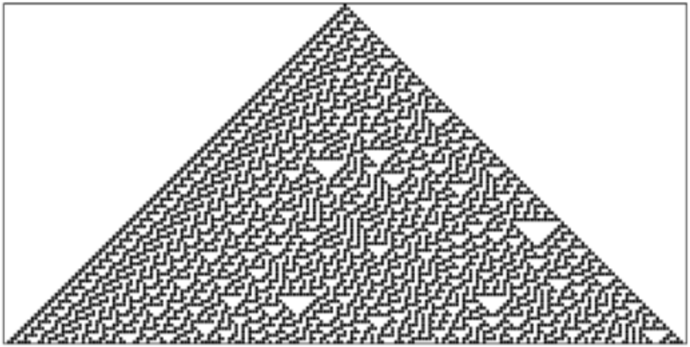
Wolfram suggested reading the bottom row as a binary number. In this case $(00110010)_2 = 50$ so "Rule 50."

Cellular Automata



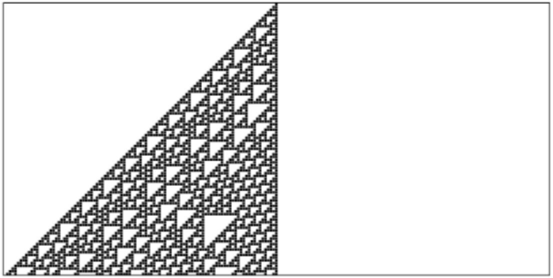
- Wolfram proposes that the behaviour of CAs can be grouped into four classes.
- Class 1 contains simplest CAs, the ones that evolve from almost any starting condition to the same uniform pattern.
- Rule 50 is an example of Class 2. It generates a simple recursive pattern. Another exmple is Rule 18.

Cellular Automata



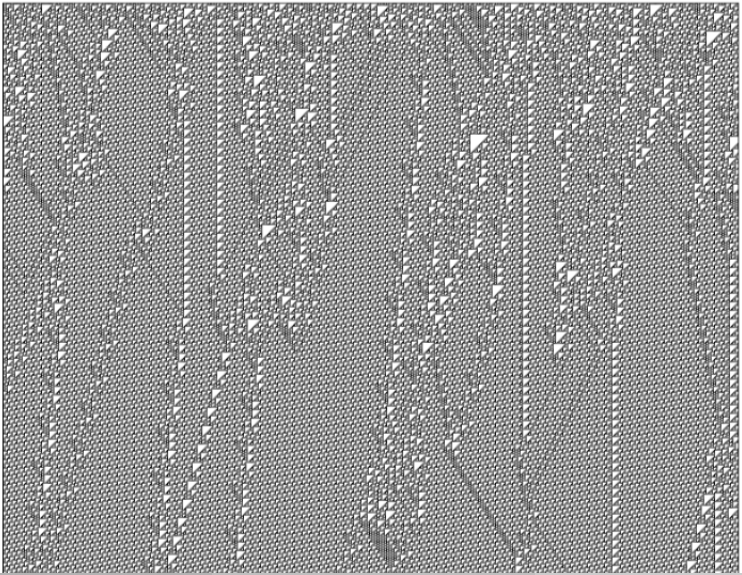
- Wolfram proposes that the behaviour of CAs can be grouped into four classes.
- Class 3 is a pattern that generate randomness. Rule 30 is an example.
- If you take the center column for Rule 30,it can be used as a pseudo-random number generator.

Cellular Automata



- Behaviour of Class 4 CAs are Turing complete, they can compute any computable function.
- This property is called universality.
- In 2002 Stephen Wolfram published "A New Kind of Science" where he presents his and others' work on cellular automata and describes a scientific approach to the study of computational systems.

Cellular Automata



Chaitin's Constant and Kolmogorov Complexity

- Descriptive complexity (K-measure) in the previous section is also called Kolmogorov Complexity.
- A Chaitin constant(Ω) is a real number that represents the probability that a randomly constructed program will halt.
- Knowing the first N bits of Ω, one could calculate the halting problem for all programs of a size up to N.
- A real number is called computable if there is an algorithm which, given n, returns the first n digits of the number.
- There is no algorithm that can compute arbitrarily many digits of Ω

Entscheidungsproblem

- Is there an effective algorithm which, given a set of axioms and a FOL[12] proposition, decides whether it is or is not provable from the axioms?
- In 1936, Alonzo Church and Alan Turing published independent papers showing that a general solution to the Entscheidungsproblem is impossible.
- Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Problem is ineffectively solvable.

[12] First Order Logic

Busy Beaver Problem

- Discovered by Tibor Radó in 1962.
- Consider a Turing Machine with one two-way infinite tape, a finite deterministic state control of n states, and alphabet $\{0, 1\}$
- There is no separate "blank" character—the tape initially holds all 0s
- At most, how long the machine may run, provided that it must stop eventually.
- The problem has been solved only up until 4 states.
- The lower bound for n=5 states is: The number of steps taken or $S(n) \geq 47176870$ and The number of 1s on the tape after machine halts or $\Sigma(n) \geq 4098$

Six State Busy Beaver

A beaver plants trees along a path in the following way:

- At any time, including the start, our beaver can decide to "Plant Two Trees" and "Think".
  - Plant at current empty spot or(if current is full) rightward empty spot.
  - If planted at right, plant to current spot; else plant to left spot and move left.
- Thinking stage involves the following
  - If he finds himself in sun with no tree on his right, "Plants Two Trees".
  - If he finds himself in sun with a tree to his right, he chops it down and "Thinks".
  - If he is in shade with no tree on his left, he plants there, and than "Plants Two Trees".
  - If he is in shade with a tree on his left, he steps left and chops it down, steps left again, and "Looks For His Shadow".

Six State Busy Beaver

A beaver plants trees along a path in the following way:

- When "Looking For His Shadow", if he sees it, he gets frightened and halts by planting a tree and burrowing under it.
- Else, he steps left again.
    - Then if in sun he steps left to "Plant Two Trees"
    - If under a tree he chops it down, moves left yet again, and "Thinks".

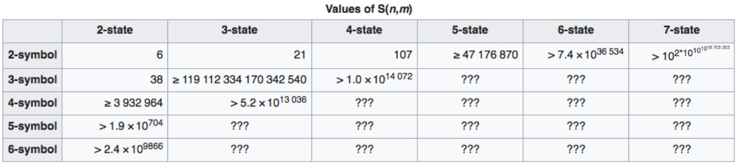
Algorithm above keeps on beavering until $\Sigma(6,2)=$95.524.079 trees in all and $S(6,2)=$8.69 trillion steps.

Current record for the 6-state 2-symbol busy beaver is
$\Sigma(6,2) = 3.5 \times 10^{18267}$ and $S(6,2) = 7.4 \times 10^{36534}$

The Busy Beaver

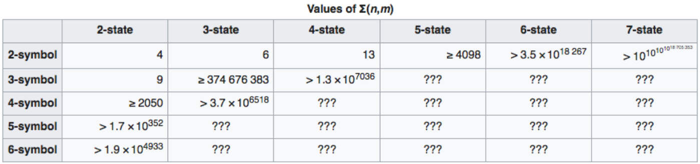
Theorem

In 1962, Radó proved that $\Sigma(n)$ and $S(n)$ are both Turing Uncomputable and they grow faster than any computable function.



Values of S(*n*,*m*)

	2-state	3-state	4-state	5-state	6-state	7-state
2-symbol	6	21	107	$\geq 47\,176\,870$	$> 7.4 \times 10^{36\,534}$	$> 10^{2*10^{10^{10^{18\,705\,353}}}}$
3-symbol	38	$\geq 119\,112\,334\,170\,342\,540$	$> 1.0 \times 10^{14\,072}$	???	???	???
4-symbol	$\geq 3\,932\,964$	$> 5.2 \times 10^{13\,036}$	???	???	???	???
5-symbol	$> 1.9 \times 10^{704}$	???	???	???	???	???
6-symbol	$> 2.4 \times 10^{9866}$	???	???	???	???	???



Values of Σ(*n*,*m*)

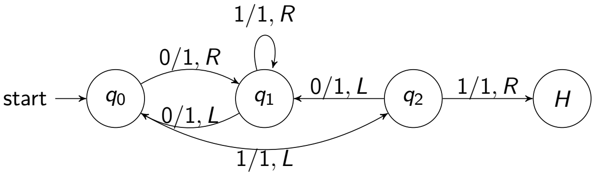
	2-state	3-state	4-state	5-state	6-state	7-state
2-symbol	4	6	13	≥ 4098	$> 3.5 \times 10^{18\,267}$	$> 10^{10^{10^{10^{18\,705\,353}}}}$
3-symbol	9	$\geq 374\,676\,383$	$> 1.3 \times 10^{7036}$	???	???	???
4-symbol	≥ 2050	$> 3.7 \times 10^{6518}$	???	???	???	???
5-symbol	$> 1.7 \times 10^{352}$	???	???	???	???	???
6-symbol	$> 1.9 \times 10^{4933}$	???	???	???	???	???

13

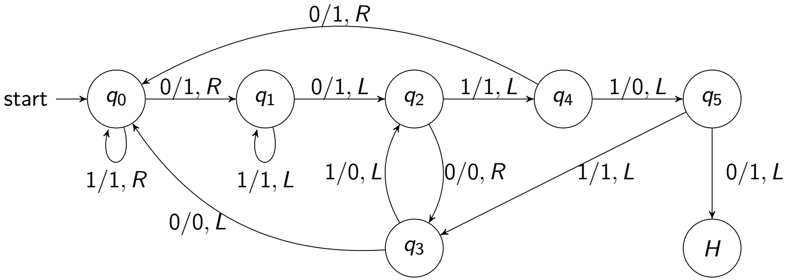
[13] Taken from wikipedia.

The Busy Beaver

The 3-state busy beaver:



The 6-state busy beaver by Heiner Marxen



Mortality Problem

Definition

Consider a finite set of $d \times d$ matrices. $S = \{M_1, M_2, \ldots, M_n\}$ with integer entries. We call S mortal iff there is a non-empty word $\omega \in \{1, 2, \ldots, n\}^*$ of length m, where product of matrices $M_{\omega_1} \cdots M_{\omega_m} = 0$

- The problem is undecidable for 3×3 matrices
- The decidability is unknown for 2×2 matrices

Word Problem

Definition

The word problem for any finite set of equations is the general question: given an equation, can it be deduced from the given equations or not?

For example, consider the equations below

$$
\begin{aligned}
ba &= abc \\
bc &= cba \\
ac &= ca
\end{aligned}
$$

We can obtain new equations by performing substitutions.
However a question like "Can we deduce for the three equations that bacabca = acbca?" is undecidable.

Time Complexity

Algorithm Complexity

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory.
- Computational complexity theory investigates the time, memory, or other resources required for solving computational problems.
- A possible way to achieve such a measurement is by using asymptotic complexity on worst-case and avarage-case running time analysis.
- Widely-known notations in complexity analysis are O, Θ and Ω.

Algorithm Complexity

Question

How much time does a single-tape Turing machine need to decide the language $A = \{0^k1^k | k \geq 0\}$?

A possible TM M_1

1. Scan across the tape and reject if a 0 is found to the right of a 1 $(O(n))$.
2. Repeat if both 0s and 1s remain on the tape $(O(n))$:
    - a. Scan across the tape, crossing off a single 0 and a single 1 $(O(n))$
3. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject . Otherwise, if neither 0s nor 1s remain on the tape, accept $(O(n))$.

The total time of M_1 on an input of length n is
$O(n) + O(n^2) + O(n) = O(n^2)$

Algorithm Complexity

A possible TM M_2

1. Scan across the tape and reject if a 0 is found to the right of a 1 $(O(n))$.
2. Repeat as long as some 0s and some 1s remain on the tape: $(O(log_2 n))$:
   a. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject $(O(n))$
   b. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.$(O(n))$
3. If no 0s and no 1s remain on the tape, accept. Otherwise, reject $(O(n))$.

The total time of M_2 on an input of length n is
$O(n) + O(nlog_2 n) + O(n) = O(nlog_2 n)$,

Algorithm Complexity

A possible TM M_3

1. Scan across the tape and reject if a 0 is found to the right of a 1 $(O(n))$.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2 $(O(n))$.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject $(O(n))$.
4. If all the 0s have now been crossed off, accept . If any 0s remain, reject $(O(n))$.

The total time of M_3 on an input of length n is $O(n)$,

Algorithm Complexity

Corollary

The type of computation model we use can effect the time complexity of a problem.

Theorem

Every $O(n)$ time multitape Turing machine has an equivalent $O(n^2)$ time single-tape Turing machine.

Theorem

Every $O(n)$ time nondeterministic single tape Turing machine has an equivalent $O(2^n)$ time deterministic single-tape Turing machine.

Polynomial Time Complexity

- On the one hand, we have a polynomial difference between single tape and multi-tape TMs.
- On the other hand, we have an exponential difference between deterministic and non-deterministic TMs.
- Let n be 1000, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe.
- Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called brute-force search.
- All reasonable deterministic computational models are polynomially equivalent. That is, any one of them can simulate another with only a polynomial increase in running time.

Polynomial Time Complexity

Definition

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k TIME(n^k)$$

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine
- P roughly corresponds to the class of problems that are realistically solvable on a computer

Path Problem in Graphs

Definition

A directed graph G contains nodes s and t. The PATH problem is to determine whether a directed path exists from s to t.

Theorem

$PATH \in P$

A polynomial time algorithm for PATH.

On input $\langle G, s, t \rangle$ with m nodes

1. Place a mark on node s.($O(1)$)
2. Repeat the following until no additional nodes are marked. ($O(m)$)
   a. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
3. If t is marked, accept. Otherwise, reject.($O(1)$)

Relative Prime Check

Theorem

$RELPRIME \in P$

A polynomial time algorithm for RELPRIME.

On input $\langle x, y \rangle$

```
1 Repeat until y=0
    a Assign $x \leftarrow x \bmod y$
    b SWAP $x$ and $y$
2 If $x = 1$ accept, otherwise reject.
```

Context Free Languages

Theorem

Every context-free language is a member of P

An Unsuccessful Attempt

1. Let L be a CFL generated by CFG G that is in Chomsky normal form.
2. Any derivation of a string ω has $2n - 1$ steps, where n is the length of ω
3. The decider for L works by trying all possible derivations with $2n - 1$ steps when its input is a string of length n.
4. The number of derivations with k steps may be exponential in k, so this algorithm may require exponential time.

Context Free Languages

Theorem

Every context-free language is a member of P

Dynamic Programming

- Use the accumulation of information about smaller subproblems to solve larger problems.
- We record the solution to any subproblem so that we need to solve it only once
- We do so by making a table of all subproblems and entering their solutions systematically as we find them.

Context Free Languages

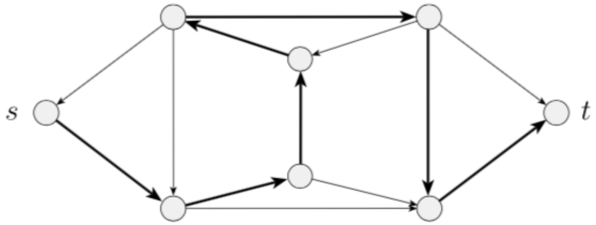
Proof.

```
1 For $w = \lambda$,if $S \rightarrow \lambda$ is a rule, accept; else, reject.
2 For $i = 1$ to $n$
    (a) For each variable A
        1 Test whether $A \rightarrow b$ is a rule, where $b = \omega_i$.
        2 If so, place A in table(i,i)
3 For $\ell = 2$ to $n$
    (a) For $i = 1$ to $n - \ell + 1$
        1 Let $j = i + \ell - 1$
        2 For $k = i$ to $j - 1$
            (a) For each rule $A \rightarrow BC$
                1 If table(i, k) contains B and table(k + 1, j) contains C, put A in table(i, j).
4 If S is in table (1, n), accept ; else, reject .
```

□

Harder Problems

- We can avoid brute-force search in many problems and obtain polynomial time solutions
- For some other certain problems polynomial time algorithms that solve them aren't known to exist.
- A good example is the Hamiltonian Path problem.



P-Verifiable Problems

- Verifying the existence of a Hamiltonian path(HAMPATH) may be much easier than determining its existence
- A natural number is composite if it is the product of two integers greater than 1. Compositeness is verifiable in polynomial time.
- Some problems may not be polynomially verifiable. For example, the complement of the HAMPATH problem.

Definition

NP is the class of languages that have polynomial time verifiers.

NP Problems

- The term NP comes from nondeterministic polynomial time.
- The following is a nondeterministic Turing machine (NTM) that decides the HAMPATH problem in nondeterministic polynomial time.

NTM for HAMPATH

On input $\langle G, s, t \rangle$

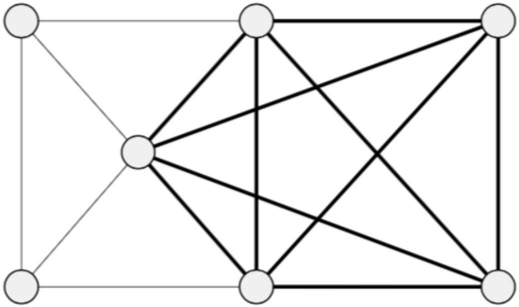
1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have been passed, so accept.

NP Problems

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Examples to NP Problems



CLIQUE

The clique problem is to determine whether a graph contains a clique of a specified size.

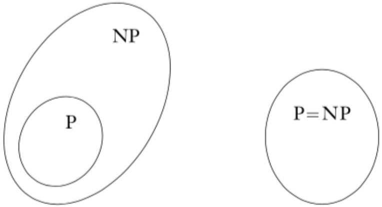
Examples to NP Problems

SUBSET-SUM

SUBSET-SUM$= \{\langle S, t\rangle | S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_\ell\} \subseteq \{x_1, \ldots, x_k\}$, we have $\Sigma_{y_i} = t\}$

- The complements of CLIQUE and SUBSET-SUM are not members of NP.
- Verifying that something is not present seems to be more difficult than verifying that it is present.

P versus NP



- We are unable to prove the existence of a single language in NP that is not in P yet.
- The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.
- The best deterministic method currently known for deciding languages in NP uses exponential time.
- We don't know whether NP is contained in a smaller deterministic time complexity class

NP-completeness

- Stephen Cook and Leonid Levin discovered certain problems in NP whose individual complexity is related to that of the entire class.
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable.
- These problems are called NP-complete.
- Proving that a problem is NP-complete is strong evidence of its nonpolynomiality.

Satisfiability Problem

$$\phi = (\neg x \wedge y) \vee (x \wedge \neg z)$$

- A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
- The satisfiability problem (SAT) is to test whether a Boolean formula is satisfiable
- Showing that $SAT \in P \Leftrightarrow P = NP$ links the complexity of the SAT problem to the complexities of all problems in NP

Satisfiability Problem

Polynomial Time Reducibility

Language A is polynomial time reducible to language B, if a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$ exists, where for every ω,

$$\omega \in A \Leftrightarrow f(\omega) \in B$$

The function f is called the polynomial time reduction of A to B.

Theorem

If A is P-Time Reducible to B and $B \in P$, then $A \in P$

3SAT

Conjunctive Normal Form

A Boolean formula is in conjunctive normal form, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6)$$

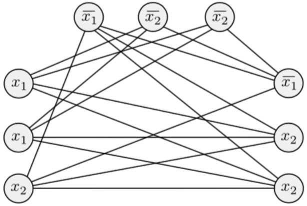
It is a 3cnf-formula if all the clauses have three literals, as in

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4)$$

Theorem

3SAT is P-Time Reducible to CLIQUE

3SAT



Proof

1. Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$
2. The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph
3. G is a k-partite graph where each cluster corresponds to one of the clauses in ϕ.
4. No edge is present between two nodes with contradictory clauses.

3SAT

Proof.

5 Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause

6 In each triple of G, we select one node corresponding to a true literal in the satisfying assignment.

7 Recently selected nodes form a k-clique.

8 They could not have contradictory labels because the associated literals were both true in the satisfying assignment.

9 Truth values are assigned to the variables of ϕ so that each literal labeling a clique node is made true.

10 This assignment to the variables satisfies ϕ

□

NP-Completeness

Definition

A language B is NP-complete if it satisfies two conditions:

1. B is in NP
2. every A in NP is P-Time Reducible to B.

Theorem

If B is NP-complete and $B \in P$, then $P = NP$.

Theorem

If B is NP-complete and B is P-Time Reducible to C for C in NP, then C is NP-complete.

Cook-Levin Theorem

- Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it.
- Establishing the first NP-complete problem is more difficult.
- We may prove SAT is NP-Complete to do so.

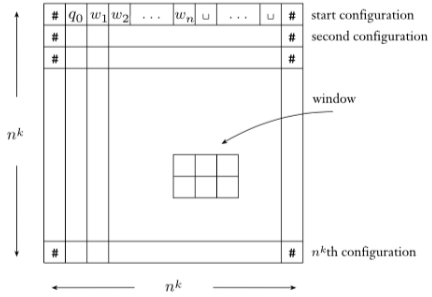
Theorem

SAT is NP-Complete

Proof Idea

- First, we show that SAT is in NP.
- Next, we take any language A in NP and show that A is P-Time reducible to SAT.

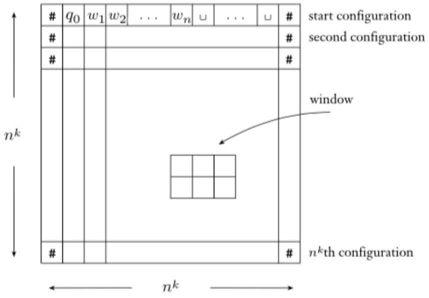
Cook-Levin Theorem



Proof

- Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k.
- We use a tableau for N on ω; an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input ω

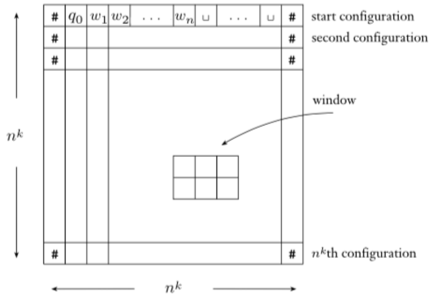
Cook-Levin Theorem



Proof

- The first row of the tableau is the starting configuration of N on ω, and each row follows the previous one according to N's transition function.
- A tableau is accepting if any row of the tableau is an accepting configuration.

Cook-Levin Theorem



Proof

- The problem of determining whether N accepts ω is equivalent to the problem of determining whether an accepting tableau for N on ω exists.

Cook-Levin Theorem

Proof

- Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ
- We represent the contents of the cells of the tableau with the variables of ϕ.
- If a boolean variable $x_{i,j,s}$ takes on the value 1, it means that cell[i, j] contains an s, symbol from the tape.
- Now we design ϕ so that a satisfying assignment to the variables does correspond to an accepting tableau for N on ω.
- The formula ϕ can be broken up to four parts as $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$

Cook-Levin Theorem

Proof.

- ϕ_{cell}: Becomes true when exactly one variable is true for each cell
- ϕ_{start}: Becomes true when the tableau has a valid starting configuration
- ϕ_{move}: Becomes true when configurations has valid transformations. It does so by mapping valid transitions in the transition table to 2×3 windows' in the tableau.
- ϕ_{accept}: Becomes true when the tableau has a valid accepting final row.

For each component of the formula a reduction that produces ϕ in polynomial time from the input ω can be constructed. $\square$

Theorem

3SAT is NP-complete.

The Vertex Cover Problem

If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size

Theorem

VERTEX-COVER is NP-complete. 3SAT can be reduced to Vertex cover.

Time Complexity Classes

coNP Class

Definition

coNP is the complexity class which contains the complements of problems found in NP.

- The complements of CLIQUE, HAMPATH, SUBSET-SUM and SAT are not members of NP.
- Verifying that something is not present seems to be more difficult than verifying that it is present.
- What about PRIMES and COMPOSITES?[14]

[14] Check AKS algorithm by Agrawal et al.

NP-hardness

Definition

Problem A is NP-hard when there exists a P-Time Reduction from every problem in NP; even though the problem itself is not in NP.

$$\text{NP-Hard} \cap \text{NP} = \text{NP-complete}$$

Example

Halting problem is NP-hard but not NP-complete.

A very rough generalization might assume that non-trivial optimization and scheduling problems fall into NP-hard category.

NP-hard Problems

Example

- For example, "Is there a Hamiltonian cycle with length less than k" is NP-complete.
- "What is the shortest hamiltonian tour?", is NP-hard

Example

- Is it possible to find *k* star-shaped polygons[a] whose union is equal to a given simple polygon is NP-complete.
- Finding the minimum number (least k) of star-shaped polygons whose union is equal to a given simple polygon, is NP-hard.

[a] A polygon P in which there exists an interior point p such that all the boundary points of P are visible(straight line segment between the points does not intersect any other edge) from p.

EXPTIME and NEXPTIME

EXPTIME Definition

The set of all decision problems that are solvable by a deterministic Turing machine in exponential runtime.

NEXPTIME Definition

The set of all decision problems that are solvable by a nondeterministic Turing machine in exponential runtime.

Examples of EXPTIME Problems

- Towers of Hanoi is an EXPTIME problem.
- Other examples of EXPTIME problems include the problem of evaluating a position in generalized chess, checkers or Go because games can last for a number of moves that is exponential in the size of the board

Examples of NEXPTIME Problems[15]

- Assume two regular expressions e_1 and e_2 involving only the operations of union, concatenation and squaring. Are those regular expressions represent different sets.
- A tile is an ordered quadruple $T = (N_T, S_T, E_T, W_T)$ of integers. Suppose we are given a finite collection $C(T_1, \ldots, T_m)$ of tiles satisfying $E_{T_i} = W_{T_{i+1}}$. Is there a tiling of an $n \times n$ square defined as $f : \{1, \ldots, n\} \times \{1, \ldots, n\} \leftarrow C$ where $\forall (i,j) : 1 \leq (i,j) \leq n$
  - $f(1,i) = T_i$
  - $j < n \leftarrow E_{f(i,j)} = W_{f(i,j+1)}$
  - $i < n \leftarrow N_{f(i,j)} = S_{f(i+1,j)}$

[15] Examples taken from "Handbook of Theoretical Computer Science", Jan van Leeuwen

AC^0

Circuit Complexity

Circuit complexity is a branch of computational complexity theory in which Boolean functions are classified according to the size or depth of Boolean circuits that compute them.

AC Hierarchy

In AC complexity class hierarchy each class, AC^i, consists of the languages recognized by Boolean circuits with depth $O(\log^i n)$ and a polynomial number of unlimited fan-in AND and OR gates.

AC^0

AC^0 Class

AC^0 is the smallest class in the AC hierarchy, and consists of all families of circuits of depth $O(1)$.

Examples

Integer addition and subtraction are computable in AC^0 but multiplication is not.

#P Classs

Definition

A function $f : \{0,1\}^* \leftarrow \mathbb{N}$ is in #P if there exists a polynomial $P : \mathbb{N} \leftarrow \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$:

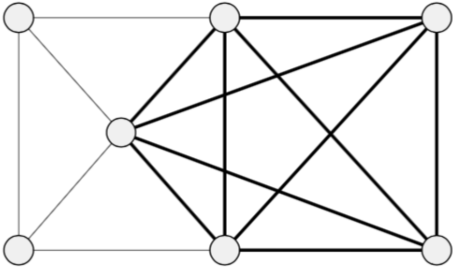
$$f(x) = |\{y \in \{0,1\}^{p(|x|)} : M(x,y) = 1\}|$$

Examples

- Given a nondeterministic Turing machine M that runs in polynomial time, and an input x for the machine; What is the number of different accepting computations of M on x?
- What is the number of Hamiltonian cycles in this graph?
- What is the number of different spanning trees in this graph?
- How many different variable assignments will satisfy a given general boolean formula?
- Given a directed graph G, the number of simple cycles in G.

Remarkable NP-Complete Problems

The Clique Problem



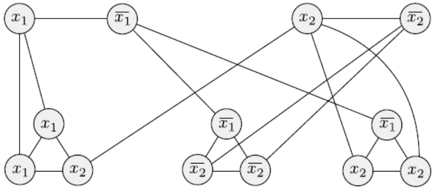
CLIQUE

The clique problem is to determine whether a graph contains a clique of a specified size.

Theorem

3SAT is P-time reducible to CLIQUE. CLIQUE is NP-complete.

The Vertex-Cover Problem



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

- We look for *gadgets* in that language that can simulate the variables and clauses in Boolean formulas
- Each variable gadget is a variable node connected to its negation.
- Each clause gadget is a triple of nodes that are labeled with the three variables of the clause
- These three nodes are connected to each other and to the nodes in the variable gadgets that have the identical labels.

The Vertex-Cover Problem

Proof

- The total number of nodes that appear in G is $2m + 3\ell$ where ϕ has m variables and ℓ clauses.
- We need to show that ϕ is satisfiable iff G has a vertex cover with $k = m + 2\ell$ nodes.
- We start with a satisfying assignment. We first put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover.
- Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover.
- They cover all edges because every variable gadget edge is clearly covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered.

The Vertex-Cover Problem

Proof.

- If G has a vertex cover with k nodes, we show that ϕ is satisfiable by constructing the satisfying assignment
- The vertex cover must contain one node in each variable gadget and two in every clause gadget
- We take the nodes of the variable gadgets that are in the vertex cover and assign TRUE to the corresponding literals. That assignment satisfies ϕ

$\square$

The Hamiltonian Path Problem

Theorem

HAMPATH is NP-complete.

- 3SAT is polynomial time reducible to HAMPATH.
- It is possible to convert 3cnf-formulas to graphs in which Hamiltonian paths correspond to satisfying assignments of the formula
- The graphs contain gadgets that mimic variables and clauses.
- The variable gadget is a diamond structure that can be traversed in either of two ways, corresponding to the two truth settings.
- The clause gadget is a node.
- Ensuring that the path goes through each clause gadget corresponds to ensuring that each clause is satisfied in the satisfying assignment.

The Hamiltonian Path Problem

Theorem

UHAMPATH is NP-complete.

- The reduction takes a directed graph G with nodes s and t, and constructs an undirected graph G' with nodes s' and t'
- Each node u of G, except for s and t, is replaced by a triple of nodes u^{in}, u^{mid} and u^{out} in G'.
- Nodes s and t in G are replaced by nodes s^{out} and t^{in}.
- An edge in G' either connects u^{mid} with u^{in} and u^{out} ...
- ... or an edge connects u^{out} with v^{in} if an edge goes from u to v in G.
- G has a Hamiltonian path from s to t iff G' has a Hamiltonian path from s^{out} to t^{in}

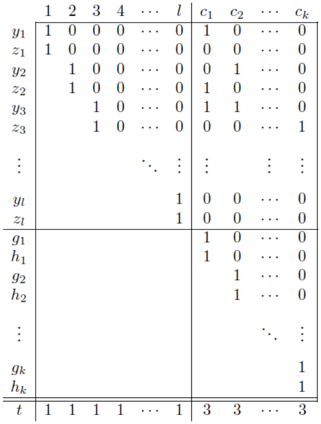
The Subset Sum Problem

Theorem

SUBSET-SUM is NP-complete.

- We construct an instance of the SUBSET-SUM problem that contains a subcollection T, summing to the target t if and only if a corresponding 3CNF formula is satisfiable.
- We represent variables by pairs of numbers and clauses by certain positions in the decimal representations of the numbers
- Variable x_i can be represented by y_i and z_i where either one of them must be in T.
- Each clause position contains a certain value in the target t, which imposes a requirement on the subset T

The Subset Sum Problem



	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

- S contains one pair of numbers, y_i, z_i, for each variable x_i in ϕ
- The digit of y_i in column c_j is 1 if clause c_j contains literal x_i, and the digit of z_i in column c_j is 1 if clause c_j contains literal $\bar{x}_i$.
- g_j and h_j in S consist of a single 1 followed by 0s.
- The target number t, the bottom row of the table, consists of ℓ 1s followed by k 3s.

The Knapsack Problem

Definition

The decision version of the Knapsack problem can be defined as:
Given n items with size $s_1, s_2, \ldots, s_n$ value $v_1, v_2, \ldots, v_n$, capacity B and value V, is there a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum\limits_{i \in S} s_i \leq B$ and $\sum\limits_{i \in S} v_i \geq V$

Theorem

Knapsack is NP-complete

Proof.

1) Knapsack is NP.
Compute and compare $\sum\limits_{i \in S} s_i \leq B$ and $\sum\limits_{i \in S} v_i \geq V$ which takes polynomial time in the size of input. $\square$

The Knapsack Problem

Theorem

Knapsack is NP-complete

Proof.

2) There is a polynomial reduction from subset sum problem to Knapsack.

- We create such a Knapsack problem that

$$s_i = v_i = num_i$$
$$B = V = SUM$$

- If the same integer set for both item capacities and values are used then
  - The same set that satisfies both conditions of the knapsack problem means there is a single value that also is an answer to subset sum
  - If there is no such a number than the answer to subset sum is also No.

□

The Set Partitioning Problem

Definition

The Set Partition Problem takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A}$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$

Theorem

SET-PARTITION is NP-complete

Proof.

1) SET-PARTITION is NP.
Taking the sum of each partition and comparison of sums takes polynomial time. □

The Set Partitioning Problem

Proof.

2) There is a polynomial reduction from subset sum problem to set partition.

- Let $\langle S, t\rangle$ be an instance of SUBSET-SUM and $s = \sum S$
- Let $S' = S \cup \{s - 2t\}$ be an instance of SET-PARTITION. $\sum S' = 2s - 2t$
- If there exists a set of numbers in S that sum to t, then the remaining numbers in S sum to $s - t$. Therefore, there exists a partition of S' into two such that each partition sums to $s - t$.
- Let's say that there exists a partition of S' into two sets such that the sum over each set is $s - t$. One of these sets contains the number $s - 2t$. Removing this number, we get a set of numbers whose sum is t, and all of these numbers are in S.

$\square$

The Bin Packing Problem

Definition

Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of positive integers, a bin capacity B and a target integer K, can we partition S into K subsets such that each subset sums to at most B.

Theorem

BIN-PACKING is NP-complete

Proof.

1) BIN-PACKING is NP.
Taking the sum of each partition and comparison of sums with B takes polynomial time. □

The Bin Packing Problem

Theorem

BIN-PACKING is NP-complete

Proof.

2) There is a polynomial reduction from set partition problem to bin packing.

- Let's assume we have a set partition problem over set $S = \{a_1, a_2, \ldots, a_n\}$.
- We create an instance of bin packing with $B = \sum_i a_i/2$ and target $K = 2$

$\square$

The Graph Coloring Problem

Definition

Given a graph $G(V, E)$, can we assign k colours to the vertices such as $c : V \rightarrow \{c_1, c_2, \ldots, c_k\}$ where adjacent vertices receive different colors.

The minimum colouring problem asks for the smallest k to properly colour G. The k-colouring problem asks whether G can be properly coloured using at most k colours.

Theorem

3-COL is NP-complete

Proof.

1) There is a polynomial reduction from 3-SAT problem to 3-COL.
Our verifier takes a graph with n vertices and checks every edge of the graph if the connecting nodes have the same colour. Verification is in P-time, specifically $O(n^2)$ $\square$

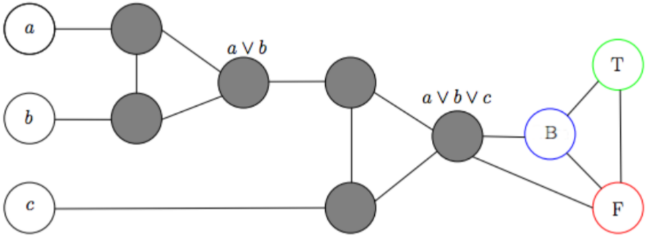
The Graph Coloring Problem

Proof

2) There is a polynomial reduction from 3-SAT problem to 3-COL.

- Let ϕ be a 3-SAT instance and $C_1, C_2, \ldots, C_m$ be the clauses of ϕ defined over the variables $\{x_1, x_2, \ldots, x_n\}$.
- We start by building a 3-connected graph (as a subgraph of G) labeled as $\{T, F, B\}$ corresponding to true, false and center nodes.
- We next add two vertices to our initial graph as $\{v_i, \overline{v_i}, B\}$ for every literal x_i.
- Notice that so far, this construction captures the truth assignment of the literals.
- We continue to build the graph by adding constraints to capture the satisfiability of the clauses of ϕ.

The Graph Coloring Problem



Proof.

2) There is a polynomial reduction from 3-SAT problem to 3-COL.

- OR-nodes can be colored true iff one of the inputs is coloured true.
- Our initial 3-SAT instance ϕ is satisfiable iff the graph G as constructed above is 3-colourable.

□

The Scheduling Problem

Definition

Given a set J of jobs of different lengths, a number of workers k, and a number t: Can the jobs in J be assigned to the k workers such that all jobs are finished within t units of time.

Theorem

JOB-SCHEDULING is NP-complete

Proof.

1) JOB-SCHEDULING is NP.
Taking the sum of each assignment, finding the maximum and comparing it with t takes polynomial time. $\square$

The Scheduling Problem

Theorem

JOB-SCHEDULING is NP-complete

Proof.

2) There is a polynomial reduction from SET-PARTITION problem to JOB-SCHEDULING.

- Given a set of numbers to partition, create one task for each number.
- Have two workers.
- See if the workers can complete the tasks in time at most half the total time required to do all jobs.

□

The Exact Cover Problem

Definition

Let U be a set of elements and $S \subseteq \mathbb{P}(U)$. An exact covering of U is a collection of sets $I \subseteq S$ such that every element of U belongs to exactly one set in I. Does S contain an exact covering of U

Theorem

EXACT-COVER is NP-complete

Proof.

1) EXACT-COVER is NP.
Checking if each set in solution I is contained in S can be done in P-time.
Checking if every element in every set of I is contained in U can be done in P-time. □

Theorem

EXACT-COVER is NP-complete

Proof.

2) There is a polynomial reduction from 3-COL problem to EXACT-COVER.

- For each node v in graph G, construct four elements in the universe U. An element v and three colour nodes R_v, G_v, and B_v.
- For each edge $\{u, v\}$ in graph G, construct three colour nodes R_{uv}, G_{uv} and B_{uv}.
- For each node v in graph G, construct a set belonging to S containing v and each R_{uv} for each edge in the graph. Repaeat this process for G and B colours as well.
- Add singleton sets containing each individual element except for elements corresponding to nodes.

□

The 0-1 Integer Programming Problem

Definition

A set of linear inequalities over Boolean variables $x_1, x_2, \ldots, x_n$ with rational coefficients. Is there an assignment of values that satisifies the inequalities.

Theorem

0-1-IP is NP-complete

Proof.

1) 0-1-IP is NP.
Checking if value assignment meets the constraints can be done in P-time. $\square$

Theorem

0-1-IP is NP-complete

Proof.

2) There is a polynomial reduction from 3-SAT problem to 0-1-IP.

- For each i add inequalities $0 \leq x_i \leq 1$
- Express each clause as an inequality such as $x_1 + (1 - x_2) + x_3 \geq 1$ for $x_1 \vee \overline{x_2} \vee x_3$

$\square$

The Traveling Salesman Problem

Definition

Given n cities and integer distances d_{ij} between a city pair i and j if connected, the traveling salesman problem asks for the total distance of the shortest tour of all the cities where the tour finishes where it began. In short terms, the shortest hamiltonian cycle.
The decision version asks if there is a hamiltonian cycle with a total distance at most B.

Theorem

TSP is NP-complete

Proof.

1) TSP is NP.
Checking if a given cycle is indeed a Hamiltonian cycle and checking if cycle's length is less than a specific value oth can be done in P-time. □

Theorem

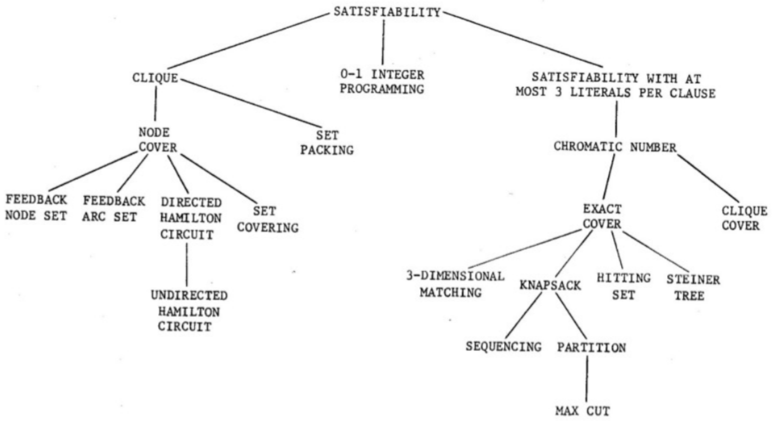
TSP is NP-complete

Proof.

2) There is a polynomial reduction from HAMPATH problem to TSP.

- Construct G' which includes one city c_v for for each vertex v in G, plus one more home city c_H for home.
- Count the vertices in G and call it n. Set $b = n + 1$.
- Define $d(c_u, c_v)$ as follows:
  - $d(c_u, c_v) = 1$, if u and v are vertices in G and have an edge between them.
  - $d(c_u, c_v) = b + 1$, if u and v are vertices in G and don't have an edge between them.
  - $d(c_u, c_v) = 1$ if one city is c_H and the other is c_s or c_t where s and t is "start" and "target" nodes in HAMPATH.
  - $d(c_u, c_v) = b + 1$, if one city is c_H and other is anything other than c_s or c_t

□



Richard Karp. "Reducibility Among Combinatorial Problems." 1972

Roadmap for NP-Completeness Proofs

To show that a decision problem A is NP-Complete, we may take the following steps:

- there is a non-deterministic polynomial-time algorithm that solves A. $A \in NP$
- any NP-Complete problem B can be reduced to A
- the reduction of B to A works in polynomial time
- A's decisions exactly correspond to B's decisions.

Can you discuss if solving sudoku is NP-complete?

Space Complexity

Space Complexity

Assuming that the following machines do halt

Definition

The space complexity of a Turing machine M is the function $f : N \rightarrow N$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n.

Definition

The space complexity of a nondeterministic Turing machine M, $f(n)$ is defined as the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

We typically estimate the space complexity of Turing machines by using asymptotic notation.

SPACE - NSPACE

Definition

SPACE is the class of languages that are decidable using linear space on a deterministic single-tape Turing machine.

Definition

NSPACE is the class of languages that are decidable using linear space on a nondeterministic single-tape Turing machine.

Definition

PSPACE is the class of languages that are decidable using polynomial space on a deterministic single-tape Turing machine.

Definition

NPSPACE is the class of languages that are decidable using polynomial space on a nondeterministic single-tape Turing machine.

SAT revisitied

SAT can be solved with a linear space algorithm. Keep in mind that the space can be reused, whereas time cannot. On input ϕ

```
1 For each truth assignment to the variables $x_1, x_2, \ldots x_m$ of $\phi$ do
    a Evaluate $\phi$ on that truth assignment.
2 If $\phi$ ever evaluated to TRUE accept, otherwise reject.
```

$$ALL_{NFA} = \{\langle A \rangle \mid \text{A is an NFA and } L(A) = \Sigma^*\}$$

A nondeterministic linear space algorithm that decides the complement of this language $\overline{ALL_{NFA}}$

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M
   a. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.
3. Accept if stage 2 reveal some string that M rejects; that is, if at some point none of the markers lie on accept states of M. Otherwise, reject.

Savitch's Theorem

Theorem

For any function $f : \mathbb{N} \rightarrow \mathbb{R}^+$, where $f(n) \geq n$

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

- Any nondeterministic TM that uses $f(n)$ space can be converted to a deterministic TM that uses only $f^2(n)$ space.
- We need to simulate an $f(n)$ space NTM deterministically.
- Trying all the branches of the NTM's computation, one by one requires $2^{f(n)}$ space.

Savitch's Theorem

Yieldability Problem

Given two configurations of the NTM, c_1 and c_2, together with a number t, is it possilbe to test whether the NTM can get from c_1 to c_2 within t steps using only $f(n)$ space?

The yieldability problem can be solved by a deterministic, recursive algorithm.

1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether c_1 yields c_2 in one step according to the rules of N.
2. If $t > 1$, then for each configuration c_m of N using space $f(n)$
   a. Run $CANYIELD(c_1, c_m, t/2)$
   b. Run $CANYIELD(c_m, c_2, t/2)$
   c. If steps 3 and 4 both accept, then accept.
3. If haven't yet accepted, reject .

Savitch's Theorem

Proof.

- Whenever *CANYIELD* invokes itself recursively, it stores the current stage number and the values of c_1, c_2, t on a stack.
- Each level of the recursion thus uses $O(f(n))$ additional space.
- Each level of the recursion divides the size of t in half
- Initially t starts out equal to $2^{df(n)}$, so the depth of the recursion is $O(log_2 df(n))$ or $O(n)$.
- Constant d ensures that N has no more than $2^{df(n)}$ configurations using $f(n)$ tape.

□

PSPACE Class

Definition

PSPACE is the class of languages that are decidable in polynomial space on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k SPACE(n^k)$$

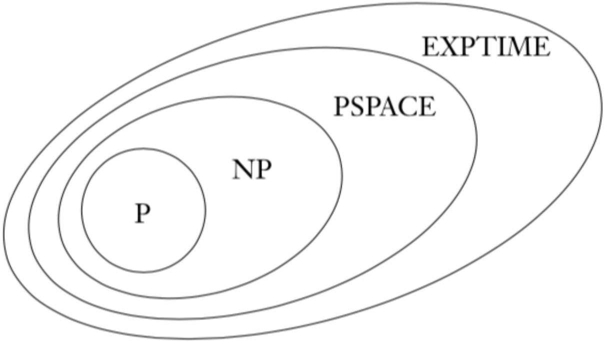
PSPACE = NPSPACE by virtue of Savitch's theorem because the square of any polynomial is still a polynomial.

SAT is in linear *SPACE* and ALL_{NFA} is in *coNSPACE* and hence, by Savitch's theorem, in $SPACE(n^2)$ which means in *PSPACE*.

PSPACE Class

- For $t(n) \geq n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space because a machine can explore at most one new cell at each step of its computation
- A TM that uses n space can have at most $n2^{O(n)}$ different configurations.
- A TM computation that halts may not repeat a configuration, therefore this TM must run in time $n2^{O(n)}$.
- $NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$

PSPACE Class



PSPACE Completeness

Definition

A language B is PSPACE-complete if it satisfies two conditions.

1. B is in PSPACE
2. Every A in PSPACE is polynomial *time* reducible to B

If [1] does not apply but [2] applies we say that it is PSPACE-hard.

The TQBF Problem

Prenex Normal Form

A logical statment containing quantifiers is in prenex normal form if all quantifiers appear at the beginning of the statement and that each quantifier's scope is everything following it.

Example

Following is a Quantifies Boolean Formula.

$$\phi = \forall x \exists y[(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

ϕ is true, it would have been false if the quantifiers were reversed in order.

Definition

When each variable of a formula appears within the scope of some quantifier, the formula is said to be fully quantified. The TQBF problem is to determine whether a fully quantified Boolean formula is true or false.

The TQBF Problem

Theorem

TQBF problem is PSPACE-complete.

- To show that TQBF is in PSPACE, we give a straightforward algorithm that assigns values to the variables and recursively evaluates the truth of the formula for those values.
- To show that every language A in PSPACE reduces to TQBF in polynomial time, we begin with a polynomial space-bounded Turing machine for A.
- Then we give a polynomial time reduction that maps a string to a quantified Boolean formula ϕ that encodes a simulation of the machine on that input.
- The formula is true iff the machine accepts.

The TQBF Problem

Proof

First, we give a polynomial space algorithm deciding TQBF.

1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and accept if it is true; otherwise, reject
2. If ϕ equals $\exists x \psi$, recursively run machine on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then accept; otherwise, reject.
3. If ϕ equals $\forall x \psi$, recursively run machine on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then accept ; otherwise, reject

The TQBF Problem

Proof.

Next, we give a polynomial time reduction from language A to TQBF. Language A is an arbitrary language decided by a TM M in PSPACE.

1. The reduction maps a string ω to a quantified Boolean formula ϕ that is true iff M accepts ω
2. The formula encodes the contents of configuration cells as in the proof of the Cook-Levin theorem.
3. Each cell has several variables associated with it, one for each tape symbol and state, corresponding to the possible settings of that cell.
4. Each configuration has n^k cells and so is encoded by $O(n^k)$ variables.
5. Proof continues as in Savitch's theorem. We let ϕ be the formula $\phi_{c_{start}, c_{accept}, h}$.
6. We perform the mapping recursively by solving the yielding problem.

□

The Formula Game

Definition

Let $\phi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_k [\psi]$. Two players select variables bound to $\forall$ and $\exists$ quantifiers in turn. Players compete to make ψ TRUE and FALSE respectively.

Example I

$$\exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})]$$

Second player has, so called, a winning strategy: Always choose $x_2 = 0$

Example II

$$\exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})]$$

First player has a winning strategy: Choose $x_1 = 1$ and than select $x_3 = \neg x_2$

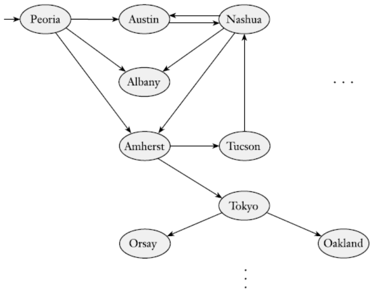
The Formula Game

Theorem

The Formula Game is PSPACE-complete.

FORMULA-GAME is PSPACE-complete because it is the same as TQBF.

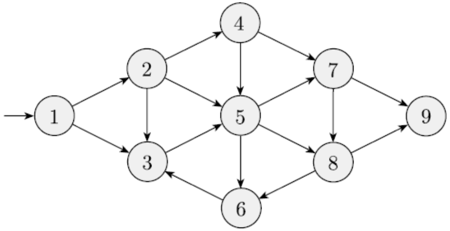
The Geography Game



Definition

In Geography, players take turns naming cities from anywhere in the world. Each city chosen must begin with the same letter that ended the previous city's name. Repetition isn't permitted. The game starts with some designated starting city and ends when some player loses because he or she is unable to continue.

The Geography Game



Definition

In generalized geography, instead of using city-first/last letter convention we take an arbitrary directed graph with a designated start node. Players took turns to select next node to make the other player get stuck.

The Geography Game

Theorem

The problem of determining which player has a winning strategy in a generalized geography(GG) game is PSPACE-complete.

First we show that GG is PSPACE.

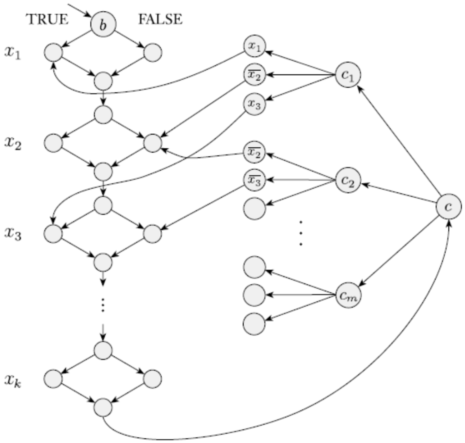
Proof.

1. If initial node (a) has outdegree 0, reject because Player I loses immediately
2. Remove node a and all connected arrows to get a new graph G'
3. For each of the nodes $a_1, a_2, \ldots, a_k$ that b originally pointed at recursively call M on G'
4. If all of these accept, Player II has a winning strategy in the original game, so reject. Otherwise, Player II doesn't have a winning strategy, so Player I must; therefore, accept.

□

The Geography Game

Next, we show that GG is PSPACE-hard by showing that formula game is polynomial time reducible to GG.



L and NL Complexity Classes

Definition

L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine.

NL is the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine.

We may assume that we are working with a two tape Turing Machine in order to solidify the examples in this context.

L and NL Complexity Classes

Example I

The language $A = \{0^k 1^k | k \geq 0\}$ is in L.

Example II

Recall the PATH problem for a graph G. We don't know whether PATH can be solved in logarithmic space deterministically, but we do know a nondeterministic log space algorithm for PATH.

The machine nondeterministically guesses the nodes of a path from s to t. It records only the position of the current node at each step on the work tape, not the entire path.

L and NL Complexity Classes

- Our earlier claim that any $f(n)$ space bounded Turing machine also runs in time $2^{O(f(n))}$ is no longer true for very small space bounds
- A configuration of M on ω is a setting of the *state*, *the work tape*, and *the positions of the two tape heads*
- If M runs in $f(n)$ space and ω is an input of length n, the number of configurations of M on ω is $n2^{O(f(n))}$.
- We can also extend Savitch's theorem to hold for sublinear space bounds down to $f(n) \geq logn$.

NL Completeness

- Analogous to the question of whether P = NP, we have the question of whether L = NL
- We define an NL-complete language to be one that is in NL and to which any other language in NL is reducible.
- However, we don't use polynomial time reducibility here because, as you will see, all problems in NL are solvable in polynomial time except $\emptyset$ and Σ^*.
- P-time reducibility is too strong to differentiate problems in NL from one another. Instead we use a new type of reducibility called log space reducibility.

NL Completeness

Definition

- A log space transducer is a two-tape Turing machine. Write head cannot only write and move rightward.
- A log space transducer M computes a function where $f(\omega)$ is the string with length $O(logn)$ remaining on the output tape after M halts when it is started with ω on its input tape.
- f is called a log space computable function.
- Language A is log space reducible to language B, if A is reducible to B by means of a log space computable function f.

Definition

A language B is NL-complete if

- $B \in NL$ and
- Every A in NL is log space reducible to B.

NL Completeness

Theorem

PATH problem is NL-complete

Proof.

- We must show that every language A in NL is log space reducible to PATH.
- The reduction maps a string ω to a graph whose nodes correspond to the configurations of the NTM on input ω.
- Nodes are connected if the corresponding first configuration can yield the second configuration in a single step.
- Hence the machine accepts ω whenever there is a path between initial configuration node and accepting configuration node.

$\square$

NL Completeness

Corollary

$NL \subseteq P$

Proof.

- Recall that a Turing machine that uses space $f(n)$ runs in time $n2^{O(f(n))}$.
- A reducer that runs in log space also runs in polynomial time.
- Any language in NL is polynomial time reducible to PATH, which in turn is in P.
- Every language that is polynomial time reducible to a language in P is also in P

□

coNL Complexity

Theorem

NL=coNL

- Let m be the number of nodes in G and c be the number of nodes in G that are reachable from s.
- We assume that c is provided as an input to M and show how to use c to solve $\overline{PATH}$
- M nondeterministically selects exactly c nodes reachable from s, not including t, and proves that each is reachable from s by guessing the path.
- For each i from 0 to m, we define A_i to be the collection of nodes that are at a distance of i or less from s.
- So $A_0 = \{s\}$, each $A_i \subseteq A_{i+1}$, and A_m contains all nodes that are reachable from s.
- The algorithm goes through all the nodes of G, determines whether each is a member of A_{i+1}, and counts the members.

An algorithm for $\overline{PATH}$. On input $\langle G, s, t\rangle$ and m as the number of nodes

```
Let $c_0 = 1$
for $i \leftarrow 0$ to $m-1$ do
    Let $c_{i+1} = 1$
    for each node $v \neq s$ in $G$ do
        Let $d = 0$
        for each node $u$ in $G$ nondeterministically do
            Follow a path of length at most $i$ from $s$ and reject if it doesn't end at $u$
            Increment $d$
            If $(u, v)$ is an edge of $G$, increment $c_{i+1}$ and continue to loop with the
              next v.
        If $d \neq c_i$, then reject.
Let $d = 0$
for each node $u$ in $G$ nondeterministically do
    Follow a path of length at most $m$ from $s$ and reject if it doesn't end at $u$
    If $u = t$, then reject.
    Increment d
If $d \neq c_m$ then reject. Otherwise accept.
```

Intractability

Time and Space Complexity Hierarchy

$$L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE$$

Intractability

Definition

Certain computational problems are solvable in principle, but the solutions require so much time or space that they can't be used in practice. Such problems are called intractable.

Big-O notation

When $f \in O(g)$, for *at least one* choice of a constant $k > 0$, the inequality $0 \leq f(x) \leq kg(x)$ holds after some x.

Little-o notation

When $f \in o(g)$, for *every* choice of a constant $k > 0$, the inequality $0 \leq f(x) < kg(x)$ holds after some x.

Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language L exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

Space Constructible Function

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $O(logn)$, is space constructible if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.

Consider $f(n) = n^2$

Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language L exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

- Should find a language L, which is decidable in O(f(n)) space but not decidable in o(f(n)) space.
- L should run in $O(f(n))$ space but it is different from any language that is decidable in $o(f(n))$ space.
- We design an algorithm A to implement the diagonalization; guaranteeing that language L is different from any machine M that decides a language in $o(f(n))$ space.

Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language L exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

- A takes its input to be the description of an arbitrary TM M that decides a language in $o(f(n))$ space.
- A runs M with input $\langle M \rangle$ within the space bound $f(n)$. If M halts within that much space, A accepts iff M rejects. If not, A rejects.
- L is decidable in space $O(f(n))$ because A does so.
- Assume that L is decided in space $g(n) \in o(f(n))$. A can simulate M, using space $dg(n)$ for some constant d.
- Because $g(n)$ is $o(f(n))$, some constant n_0 exists, where $dg(n) < f(n)$ for all $n \geq n_0$
- When A is run on input $\langle M \rangle 10^{n_0}$, M will not decide A which contradicts our assumption.

Corollary

For any two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$, where $f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible, $SPACE(f_1(n)) \subset SPACE(f_2(n))$

Corollary

For any two real numbers $0 \leq \epsilon_1 < \epsilon_2$, $SPACE(n^{\epsilon_1}) \subset SPACE(n^{\epsilon_2})$

Corollary

$NL \subset PSPACE$

Corollary

$PSPACE \subset EXPSPACE$

Time Hierarchy Theorem

For any time constructible function $t : \mathbb{N} \rightarrow \mathbb{N}$, a language L exists that is decidable in $O(t(n))$ time but not in $o(t(n)/logt(n))$ time.

Time Constructible Function

A function $t : \mathbb{N} \rightarrow \mathbb{N}$, where $t(n)$ is at least $O(nlogn)$, is time constructible if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.

Consider $f(n) = n\sqrt{n}$

Corollary

For any two functions $t_1, t_2 : \mathbb{N} \rightarrow \mathbb{N}$, *where* $t_1(n)$ *is* $o(t_2(n)/logt_2(n))$ *and* t_2 *is time constructible,* $TIME(t_1(n)) \subset TIME(t_2(n))$

Corollary

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$, $TIME(n^{\epsilon_1}) \subset TIME(n^{\epsilon_2})$

Corollary

$P \subset EXPTIME$

EXPSPACE Completeness

- We know that we can test the equivalence of two regular expressions in polynomial space.
- Let $\uparrow$ be the exponentiation operation.
- $EQ_{REX\uparrow}$ is intractable, and it is complete for the class EXPSPACE.

Definition

A language L is EXPSPACE-complete if it is in EXPSPACE and every A in EXPSPACE is P-time reducible to L.

EXPSPACE Completeness

Theorem

$EQ_{REX\uparrow}$ is EXPSPACE-complete.

To test whether two expressions with exponentiation are equivalent, we first use repetition to eliminate exponentiation, then convert the resulting expressions to NFAs. Finally, we use an NFA equivalence testing procedure.

Theorem

$EQ_{REX\uparrow}$ is EXPSPACE-complete.

First, we present a nondeterministic algorithm for testing whether two NFAs are inequivalent. On input $\langle N_1, N_2 \rangle$, where N_1 and N_2 are NFAs.

1. Place a marker on each of the start states of N_1 and N_2.
2. Repeat $2^{q_1+q_2}$ times, where q_1 and q_2 are the numbers of states in N_1 and N_2.
   (a) Nondeterministically select an input symbol and change the positions of the markers on the states of N_1 and N_2 to simulate reading that symbol.
3. If at any point a marker was placed on an accept state of one of the finite automata and not on any accept state of the other finite automaton, accept . Otherwise, reject

This algorithm runs in nondeterministic linear space. Thus, Savitch's theorem provides a deterministic $O(n^2)$ space algorithm for this problem.

Theorem

$EQ_{REX\uparrow}$ is EXPSPACE-complete.

Following algorithm decides $EQ_{REX\uparrow}$. On input $\langle R_1, R_2 \rangle$, where R_1 and R_2 are regexps with exponentiation.

1. Convert R_1 and R_2 to equivalent regular expressions ${R_1}'$ and ${R_2}'$ that use repetition instead of exponentiation.
2. Convert ${R_1}'$ and ${R_2}'$ to equivalent NFAs N_1 and N_2
3. Use the deterministic version of the previous algorithm to determine whether N_1 and N_2 are equivalent

$EQ_{REX\uparrow}$ is EXPSPACE-complete

$EQ_{REX\uparrow}$ is decidable in EXPSPACE.

- Using repetition to replace exponentiation may increase the length of an expression by a factor of 2^{ℓ}, where ℓ is the sum of the lengths of the exponents.
- $R_1{}'$ and $R_2{}'$ have a length of at most $n2^n$ where n is the input length
- NFAs N_1 and N_2 have at most $O(n2^n)$ states.
- Deterministic version of NFA equivalence algorithm uses space $O(n^2 2^{2n})$ space.

$EQ_{REX\uparrow}$ is EXPSPACE-complete

$EQ_{REX\uparrow}$ is EXPSPACE-hard.

- A language L in EXPSPACE is polynomial time reducible to $EQ_{REX\uparrow}$
- Given a TM M for L, a P-time reduction maps an input ω to a pair of expressions, R_1 and R_2, that are equivalent exactly when M accepts ω.
- The expressions R_1 and R_2 simulate the computation of M on ω.
  - R_1 generates all strings over the alphabet consisting of symbols that may appear in computation histories.
  - R_2 generates all strings that are not rejecting computation histories.
- If the TM accepts its input, no rejecting computation histories exist, and expressions R_1 and R_2 generate the same language.

Time and Space Complexity Hierarchy

- $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE$
- $P \subset EXPTIME$
- $NP \subset NEXPTIME$
- $NL \subset PSPACE \subset EXPSPACE$

Relativization and Oracle Machines

- In the relativization method, we modify our model of computation by giving the Turing machine certain information essentially for "free".
- For example, suppose that we grant the TM the ability to solve the satisfiability problem in a single step, for any size Boolean formula by attaching a so called "oracle" to the machine.
- This TM could use the oracle to solve any NP problem in polynomial time, regardless of whether P equals NP.
- Every NP problem is polynomial time reducible to the satisfiability problem.
- Such a TM is said to be computing relative to the satisfiability problem; hence the term relativization.

Oracles

Definition

- An oracle for a language A is a device that is capable of reporting whether any string w is a member of A
- An oracle Turing machine M^A is a modified Turing machine that has the additional capability of querying an oracle for A.
- Whenever M^A writes a string on a special oracle tape, it is informed whether that string is a member of A in a single computation step.
- Let P^A be the class of languages decidable with a polynomial time oracle Turing machine that uses oracle A. Define the class NP^A similarly.

Example

$NP \subset P^{SAT}$ and $coNP \subset P^{SAT}$. P^{SAT} is closed under complement.

Oracles

Definition

A Boolean formula is minimal if no shorter Boolean formula is equivalent to it. Let MIN-FORMULA be the collection of minimal Boolean formulas.

Example

We don't know if $\overline{MIN - FORMULA}$ is in NP or not. However $\overline{MIN - FORMULA}$ is in NP^{SAT}.

- The inequivalence problem for two Boolean formulas is solvable in NP
- Hence the equivalence problem is in coNP because a nondeterministic machine can guess the assignment on which the two formulas have different values
- The nondeterministic oracle machine for $\overline{MIN - FORMULA}$ nondeterministically guesses the smaller equivalent formula, tests whether it actually is equivalent, using the SAT oracle, and accepts if it is.

Limits of Diagonalization

Theorem

- *An oracle A exists where* $P^A \neq NP^A$
- *An oracle B exists where* $P^B = NP^B$

- At its core, the diagonalization method is a simulation of one Turing machine by another.
- Any theorem proved about Turing machines by using only the diagonalization method would still hold if both machines were given the same oracle.
- If we could prove that P and NP were different by diagonalizing, we could conclude that they are different relative to any oracle as well. But P^B and NP^B are equal.
- No proof that relies on a simple simulation could show that the two classes are the same because that would show that they are the same relative to any oracle; but in fact, P^A and NP^A are different.

Theorem

An oracle B exists where $P^B = NP^B$

Considering TQBF problem

$$NP^{TQBF} \subseteq NPSPACE \subseteq PSPACE \subseteq P^{TQBF}$$

- $NP^{TQBF} \subseteq NPSPACE$ holds because we can compute TQBF answers instead of using an oracle.
- $NPSPACE \subseteq PSPACE$ holds due to Savitch's theorem.
- $PSPACE \subseteq P^{TQBF}$ holds because TQBF is PSPACE-complete.

Theorem

An oracle A exists where $P^A \neq NP^A$

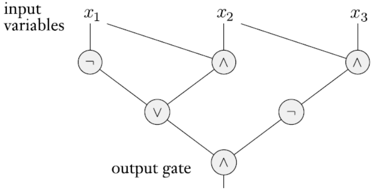
Let oracle A's language be defined as

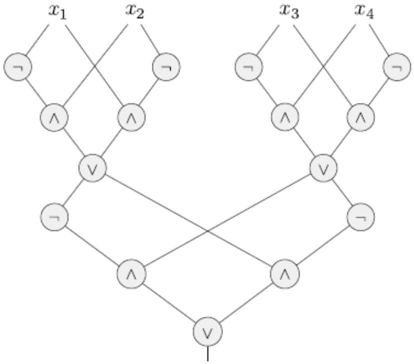
$$L_A = \{\omega | \exists x \in A[|x| = |\omega|]\}$$

- We construct A in stages where stage i constructs a part of A.
- Each part ensures a polynomial time oracle TM ${M_i}^A$ doesn't decide L_A. Each M_i runs in n^i
- We construct A by declaring that certain strings are in A and others aren't in A.
- Each stage determines the status of only a finite number of strings.

Relativization method tells us that to solve the P versus NP question, we must analyze computations, not just simulate them.

Circuit Complexity





Definition

A Boolean circuit is a collection of gates and inputs connected by wires. Cycles aren't permitted. Gates take three forms: AND gates, OR gates, and NOT gates.

Circuit Complexity

- We plan to use circuits to test membership in languages
- Any particular circuit can handle only inputs of some fixed length, whereas a language may contain strings of different lengths.
- Instead of using a single circuit to test language membership, we use an entire family of circuits, one for each input length, to perform this task.

Circuit Complexity

Definition

- Size of a circuit: Number of gates that it contains
- Size minimal circuit: No smaller circuit is equivalent to it.[a]
- Size complexity of a circuit family: Minimal size of any circuit in a circuit family.
- Depth of a circuit: The length of the longest path from an input variable to the output gate
- Depth minimality, depth complexity.

[a] Even the problem of testing whether a particular circuit is minimal does not appear to be solvable in NP.

Circuit Complexity

Definition

The circuit complexity of a language is the size complexity of a minimal circuit family for that language. The circuit depth complexity of a language is defined similarly, using depth instead of size.

Example

Let L be the language of strings that contain an odd number of 1s. Then L has circuit complexity $O(n)$

Theorem

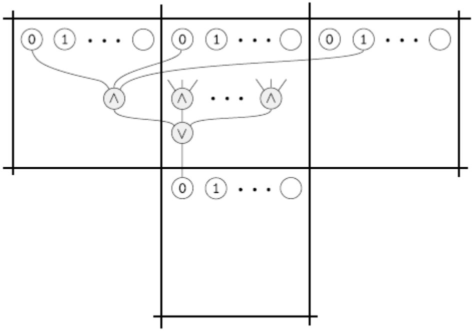
Let $t : \mathbb{N} \to \mathbb{N}$ be a function, where $t(n) \geq n$. If $L \in TIME(t(n))$, then L has a circuit complexity $O(t^2(n))$.

Proof

- Let M be a machine to decide L in time $t(n)$ and let ω be an input of length n.
- We keep a tableau of $t(n) \times t(n)$, each row corresponding to a configuration.
- We represent both the state and the tape symbol under the tape head by a single composite character.
- When M is about to accept, it moves its head onto the leftmost tape cell and writes the blank symbol.
- If we know the values at $cell[i-1, j-1]$,$cell[i-1, j]$, and $cell[i-1, j+1]$, we can obtain the value at $cell[i, j]$ with M's transition function.

Theorem

Let $t : \mathbb{N} \rightarrow \mathbb{N}$ be a function, where $t(n) \geq n$. If $L \in TIME(t(n))$, then L has a circuit complexity $O(t^2(n))$.



Proof.

A wiring as shown above constructs a circuit that simulates running of TM. □

Theorem

Circuit-satisfiability problem tests whether some setting of the inputs causes the circuit to output true. CIRCUIT-SAT is NP-complete.

Proof.

To show that any language L in NP is reducible to CIRCUIT-SAT we must give a polynomial time reduction f that maps strings to circuits as $f(\omega) = \langle C \rangle$ where $\omega \in L \Leftrightarrow C$ is satisfiable.

The construction of the circuit can be done in time that is polynomial in n. Verification of circuit runs in n^k so the size of the circuit construction is $O(n^{2k})$ $\square$

Theorem

3SAT is NP-Complete. CIRCUIT-SAT can be reduced to 3SAT in P-time.

Proof.

- To show that all languages in NP reduce to 3SAT in P-time we can reduce CIRCUIT-SAT to 3SAT in P-time.
- The reduction converts a circuit C to a formula ϕ, whereby C is satisfiable iff ϕ is satisfiable.
- The formula contains one variable for each variable and each gate in the circuit.
- Conceptually, the formula simulates the circuit.

□

Fighting Intractability (Parameterized Algorithms)

Definition

The main idea of parameterized complexity is to develop a framework that addresses certain parameters of the problem at hand that might significantly affect the complexity.

VERTEX COVER problem

- A vertex cover of $G = (V, E)$ is a set of vertices $V' \subseteq V$ that covers all edges: that is $V' = \{v_1, \ldots, v_k\}$ is a vertex cover for G iff, for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$.
- Does a given graph G have a vertex cover of size at most k?

Theorem

VERTEX COVER is NP-complete. Best known algorithm for a k-vertex cover decision runs in time $O(1.286^k + kn)$ where n is the number of vertices and k is the solution set size.

DOMINATING SET problem

- A set of vertices V' is a dominating set for G iff, for every vertex $v \in V$, either $v \in V'$ or there is some $u \in V$ such that $(u, v) \in E$.
- Dominating set might not be the vertex cover.

Theorem

DOMINATING SET is NP-complete. The brute force algorithm of trying all k-subsets runs in time $O(n^{k+1})$

Definition

Kernelization is based on eliminating those parts of the input data that are relatively easy to cope with, shrinking the given instance to some hard kernel that must be dealt with using a computationally expensive algorithm. The elimination process should be performed in polynomial time.

TRAIN COVER problem

- A bipartite graph $G = (V_S \cup V_T, E)$, where the set of vertices V_S represents railway stations and the set of vertices V_T represents trains.
- E contains an edge (s, t), $s \in V_S, t \in V_T$, iff the train t stops at the station s.
- Find a minimum set $V \subseteq V_S$ such that V covers V_T, that is, for every vertex $t \in V_T$, there is some $s \in V$ such that $(s, t) \in E$.

Kernelization Steps

1. Let $N(t)$ denote the neighbors of t in V_S. If $N(t) \subseteq N(t')$ then remove t' and all adjacent edges of t' from G. If there is a station that covers t, then this station also covers t'.
2. Let $N(s)$ denote the neighbors of s in V_T. If $N(s) \subseteq N(s')$ then remove s and all adjacent edges of s from G. If there is a train that is covered by s, then this train is also covered by s'.

K-VERTEX COVER Kernelization Steps

1. Remove all isolated vertices
2. For any degree one vertex v, add its single neighbor u to the solution set and remove u and all of its incident edges from the graph.
3. If there is a vertex v of degree at least $k+1$, add v to the solution set and remove v and all of its incident edges from the graph.

Definition

A crown in a graph $G = (V, E)$ consists of an independent set $I \subseteq V$ (no two vertices in I are connected by an edge) and a set H containing all vertices in V adjacent to I.
A crown in G is formed by $I \cup H$ iff there exists a size $|H|$ maximum matching in the bipartite graph induced by the edges between I and H, that is, every vertex of H is matched.

K-VERTEX COVER Kernelization by Crown Elimination

For any crown $I \cup H$ in G, add the set of vertices H to the solution set and remove $I \cup H$ and all of the incident edges of $I \cup H$ from G.

Depth Bounded Search Trees

Definition

- Many parameterized problems can be solved by the construction of a search tree whose depth depends only upon the parameter.
- The total size of the tree will necessarily be an exponential function of the parameter.
- To keep the size of the tree manageable the trick is to find efficient branching rules to successively apply to each node in the search tree.

K-VERTEX COVER Search Tree

- We begin by labeling the root of the tree with the empty set and the graph $G = (V, E)$. Now we pick any edge $(u, v) \in E$
- The first child of the root is labeled with $\{u\}$ and $G - u$, the second with $\{v\}$ and $G - v$.
- For each node with subgraph H, we arbitrarily choose an edge and create the two child nodes by removing incident vertex.
- If we create a node at height at most k with an empty subgraph, then a vertex cover of size at most k has been found.

This algorithm runs in time $O(2^k \cdot n)$.

K-VERTEX COVER Search Tree Simplification

- It has been observed that, if G has no vertex of degree three or more, then G consists of a collection of cycles.
- If such a G is sufficiently large, then this graph cannot have a size k vertex cover.
- In constructing the search three instead of picking arbitrary vertex we may select vertex v of degree three or greater.
- In any vertex cover of G we must have either v or all of its neighbors, so we create children of the root node corresponding to these two possibilities.

This algorithm runs in time $O(5^{\frac{k}{4}} \cdot n)$.

K-VERTEX COVER Search Tree Further Simplification

- If there is a degree one vertex v in G, with single neighbor u, then there is a minimum size cover that contains u. Thus, we create a single child node.
- If there is a degree two vertex v in G, with neighbors w_1 and w_2, then either both w_1 and w_2 are in a minimum size cover, or v together with all other neighbors of w_1 and w_2 are in a minimum size cover.
- If there is a degree three vertex v in G, then either v or all of its neighbors are in a minimum size cover.

This search tree has a size bounded by $O(1.47^k)$.

CLOSEST STRING problem

- Given k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a non-negative integer d.
- Is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?

CLOSEST STRING Search Tree

- We begin by labeling root of the three with an arbitrary string s' from the input set.
- If any other string $s_i \in S$ differs from s' in more than $2d$ positions, then there is no solution for the problem.
- At each step we look for an input string s_i that differs from s' in more than d positions but less than $2d$ positions.
- Choosing $d + 1$ of these positions, we branch into $d + 1$ subcases, in each subcase modifying one position in s' to match s_i.

Definition

- We begin with a solution that is large enough to be computed in polynomial time.
- Inductively, we apply a compression routine that either calculates a smaller solution or proves that the given solution is of minimum size.

K-VERTEX COVER Problem

Compression routine:

- Find a non-optimal cover by randomly including edges to a solution set C.
- Go over all partitions (C', C'') of C
- If C'' is an independent C' is a smaller vertex cover.

Iterative compression

- Iteratively build a larger graph by adding each node and corresponding vertices one by one.
- Apply the compression routine above at each step.
- If there is a solution larger than k at any step, answer NO.

FEEDBACK VERTEX SET Problem

In a multigraph G, is there a subset of vertices S of size at most k in the graph such that when $G - S$ the graph becomes acyclic.

Color Coding

- Randomly assign colors to the input structure
- If there is a solution and we are lucky with the coloring, every element of the solution has received a different color
- This technique is useful for problems that involve finding small subgraphs in a graph, such as paths and cycles.
- Since this is a randomized technique it gives a probabilistic guarantee in certain time bounds when applied.

K-PATH Problem

Given a graph G, an integer k, two vertices s and t, find a simple $s - t$ path with exactly k internal vertices

K-PATH Color Coding

- Assign colors from $[k]$ to vertices $V(G) - \{s, t\}$ uniformly and independently at random
- Check if there is a colorful $s - t$ path: a path where each color appears exactly once on the internal vertices; output "YES" or "NO".
- Repeating the whole algorithm a constant number of times can make the error probability an arbitrary small constant

How to check if there is a colorful $s - t$ path?

- We may check all permutations of k colorings (factorial complexity).
- We may apply dynamic programming (Exponential complexity).

Approximation, Alternation and Probabilistic Algorithms

Approximation

- In certain problems called optimization problems, we seek the best solution among a collection of possible solutions
- In practice, a solution that is nearly optimal may be good enough and may be much easier to find. An approximation algorithm is designed to find such approximately optimal solutions.

Example

In MIN-VERTEX-COVER, we aim to produce one of the smallest vertex covers among all possible vertex covers in the input graph.

For an undirected graph G:

1. Repeat the following until all edges in G touch a marked edge
   - **(a)** Find an edge in G untouched by any marked edge.
   - **(b)** Mark that edge.
2. Output all nodes that are endpoints of marked edges.

Approximation

Theorem

This algorithm is a polynomial time algorithm that produces a vertex cover of G that is no more than twice as large as a smallest vertex cover.

- MIN-VERTEX-COVER is an example of a minimization problem. In a maximization problem, we seek a largest solution.
- An approximation algorithm for a minimization problem is k-optimal if it always finds a solution that is not more than k times optimal.
- For a maximization problem, a k-optimal approximation algorithm always finds a solution that is at least $\frac{1}{k}$ times the size of the optimal.

Approximation

Definition

A cut in an undirected graph is a separation of the vertices V into two disjoint subsets S and T. The size of a cut is the number of edges that have one endpoint in S and the other in T.

Example

The MAX-CUT problem asks for a largest cut in a graph G. MAX-CUT is NP-complete. The following algorithm approximates MAX-CUT within a factor of 2.

For an undirected graph G, with nodes V:

1. Let $S = \emptyset$ and $T = V$
2. If moving a single node, either from S to T or from T to S, increases the size of the cut, make that move and repeat this stage.
3. If no such node exists, output the current cut and halt.

Approximation

Theorem

Previous algorithm is a polynomial time, 2-optimal approximation algorithm for MAX-CUT.

Proof.

B's cut edges are at least half of all edges in G.

- At every node of G, the number of cut edges is at least as large as the number of uncut edges, or B would have shifted that node to the other side
- We add up the numbers of cut edges at every node. That sum is twice the total number of cut edges because every cut edge is counted once for each of its two endpoints.
- That sum must be at least the corresponding sum of the numbers of uncut edges at every node

□

Probabilistic Algorithms

- Certain types of problems seem to be more easily solvable by probabilistic algorithms than by deterministic algorithms
- Sometimes, calculating the best choice may require excessive time, and estimating it may introduce a bias that invalidates the result.

Definition

A probabilistic Turing machine M is a type of nondeterministic Turing machine in which each computation branch is assigned a probability b on input ω as $Pr[b] = 2^{-k}$.

The probability that M accepts ω is the sum of probabilities of accepting branches.

M decides language L with error probability ϵ if $w \in L$ implies $Pr[M \;\; acc. \;\; \omega] \geq 1 - \epsilon$

BPP Complexity Class

Definition

BPP[a] is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$

[a]Bounded-error Probabilistic Polynomial Time

- Any constant error probability would yield an equivalent definition as long as it is strictly between 0 and $\frac{1}{2}$

Primality Testing

- A polynomial time algorithm of testing whether an integer is prime or composite is now known[16]
- Much simpler probabilistic polynomial time algorithms for primality testing also exists.

Theorem

Fermat's little theorem: If p is prime and $a \in {\mathbb{Z}_p}^+$, then $a^{p-1} \equiv 1 \pmod{p}$

- A number pseudoprime if it passes Fermat tests for all smaller a's relatively prime to it.
- With the exception of the infrequent Carmichael numbers, which are composite yet pass all Fermat tests, the pseudoprime numbers are identical to the prime numbers.

[16] M. Agrawal et al.

Primality Testing

Select $k \geq 1$ as a parameter, the following algorithm determines the maximum error probability to be 2^k.

1. If p is even, accept if p = 2; otherwise, reject
2. Select $a_1, \ldots, a_k$ randomly in $\mathbb{Z}_p{}^+$
3. For each i from 1 to k
   - (a) Compute $a_i{}^{p-1}$ mod p and reject if different from 1
   - (b) Let $p - 1 = s \cdot 2^\ell$ where s is odd
   - (c) Compute the sequence $a_i{}^{s \cdot 2^0}, a_i{}^{s \cdot 2^1}, a_i{}^{s \cdot 2^2}, \ldots, a_i{}^{s \cdot 2^\ell}$, modulo p
   - (d) If some element of this sequence is not 1, find the last element that is not 1 and reject if that element is not -1.
4. All tests have passed at this point, so accept.

Primality Testing

Theorem

PRIMES $\in$ *BPP*

- For the probabilistic primality algorithm an incorrect answer can only occur when the input is a composite number.
- The one-sided error feature is common to many probabilistic algorithms, so the special complexity class RP is designated for it.

Definition

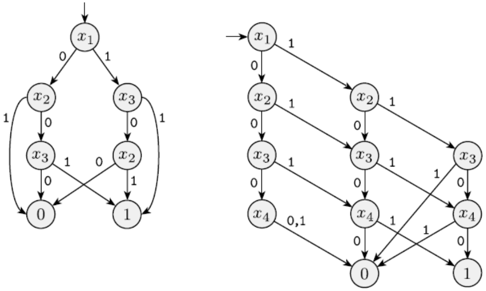
RP[a] class languages are decided by BPP TMs by accepting with a probability of at least $\frac{1}{2}$, and rejecting with a probability of 1.

[a]Randomized Polynomial Time

Theorem

COMPOSITES $\in$ *RP*

Branching Programs



- A branching program is a directed acyclic graph where all nodes are labeled by variables, except for two output nodes labeled 0 or 1.
- A read-once branching program is one that can query each variable at most one time on every directed path from the start node to an output node.
- EQ_{ROBP} is the equivalency decision on read-once branching programs.

Theorem

$EQ_{ROBP} \in BPP$

- We can assign random values to the variables x_1 through x_m and evaluate these branching programs.
- What if two inequivalent read-once branching programs disagree only on a single assignment out of 2^m
- We assign polynomials over the variables in branching program as follows
  - The constant function 1 is assigned to the start node
  - If a node labeled x has been assigned polynomial p, assign the polynomial xp to its outgoing 1-edge
  - assign the polynomial $(1-x)p$ to its outgoing 0-edge.
  - If the edges incoming to some node have been assigned polynomials, assign the sum of those polynomials to that node.

Proof.

Let $\mathbb{F}$ be a finite field with at least $3m$ elements.

1. Select elements a_1 through a_m at random from $\mathbb{F}$
2. Evaluate the assigned polynomials p_1 and p_2 at a_1 through a_m
3. If $p_1(a_1, \ldots, a_m) = p_2(a_1, \ldots, a_m)$ accept, otherwise reject.

□

Notes on polynomials over fields

A polynomial over a field $\mathbb{F}$ is a sequence $(a_0, a_1, \ldots, a_n, \ldots)$ where $a_i \in$ F $\forall i$

Three special polynomials can be defined as

$0 = (0, 0, 0, 0, \ldots) \qquad 1 = (1, 0, 0, 0, \ldots) \qquad x = (0, 1, 0, 0, \ldots)$

Let $\mathbb{F}$ be a finite field with f elements and let each variable has degree at most d. For randomly selected elements in $\mathbb{F}$,
$Pr[p(a_1, \ldots, a_m) = 0] \leq md/f$

Alternation

Definition

- An alternating Turing machine is a nondeterministic Turing machine with additional labels to states as universal states($\wedge$) and existential states($\vee$).
- A universal state accepts when all of its children are accepting in the computation tree.
- An existential state accepts when any of its children are accepting in the computation tree.
- The time and space complexity of these machines are defined in the same way that for nondeterministic Turing machines: the maximum time or space used by any computation branch.
- ATIME, ASPACE, AP, APSPACE and AL are corresponding complexity classes for Alternating TMs.

Example

TAUT: Tautology decision for a boolean formula ϕ is in AP

1. Universally select all assignments to the variables of ϕ
2. For a particular assignment, evaluate ϕ
3. If ϕ evaluates to 1, accept ; otherwise, reject.

TAUT is a member of coNP. In fact, any problem in coNP can easily be shown to be in AP by using a similar algorithm.

Example

A Boolean formula is minimal if no shorter Boolean formula is equivalent to it. Let MIN-FORMULA be the collection of minimal Boolean formulas. We don't know if MIN-FORMULA is in NP or coNP.

The following algorithm shows that MIN-FORMULA is in AP

1. Universally select all formulas ψ that are shorter than ϕ
2. Existentially select an assignment to the variables of ϕ
3. Evaluate both ϕ and ψ on this assignment.
4. Accept if the formulas evaluate to different values. Reject if they evaluate to the same value.

Alternating Complexity Classes

Theorem

For $f(n) \geq n$, $ATIME(f(n)) \subseteq SPACE(f(n)) \subseteq ATIME(f^2(n))$

For $f(n) \geq logn$, $ASPACE(f(n)) = TIME(2^{O(f(n))})$

Consequently $AL = P$ *and* $AP = PSPACE$ *and* $APSPACE = EXPTIME$

Lemma

For $f(n) \geq n$, $ATIME(f(n)) \subseteq SPACE(f(n))$

Proof.

We convert an alternating time $O(f(n))$ machine M to a deterministic space $O(f(n))$ machine S that simulates M as follows

- On input ω, the simulator S performs a depth-first search of M's computation tree to determine which nodes in the tree are accepting
- Then S accepts if it determines that the root of the tree is accepting.
- Machine S requires $O(f^2(n))$ space for storing the recursion stack that is used in the depth-first search.
- Instead of storing the entire configuration at each level of the recursion, S may record only the nondeterministic choice that M made to reach that configuration from its parent.
- Making this change reduces the space usage to $O(f(n))$

$\square$

Lemma

For $f(n) \geq n$, $SPACE(f(n)) \subseteq ATIME(f^2(n))$

Proof.

We start with a deterministic space $O(f(n))$ machine M and construct an alternating machine S that uses time $O(f^2(n))$ to simulate it

- Machine S uses a recursive alternating procedure similar to Savitch's theorem to test whether the start configuration can reach an accepting configuration within $2^{df(n)}$ steps.
- It takes $O(f(n))$ to write a configuration at each level of the recursion.
- The depth of the recursion is $log 2^{df(n)} = O(f(n))$
- The maximum time used on any branch of this alternating procedure is hence $O(f^2(n))$ in ATIME.

□

Lemma

For $f(n) \geq logn$, $ASPACE(f(n)) \subseteq TIME(2^{O(f(n))})$

Proof.

We construct a deterministic time $2^{O(f(n))}$ machine S to simulate an alternating space $O(f(n))$ machine M

- On input ω, the simulator S constructs the graph of the computation of M on ω.
- The nodes are the configurations of M. Edges go from a configuration to those configurations it can yield in a single move of M.
- After constructing the graph, S repeatedly scans it and marks certain configurations as accepting.
- S accepts if the start configuration of M is marked.
- The size of the configuration graph is $2^{O(f(n))}$
- The total number of scans is at most the number of nodes in the graph.

Lemma

For $f(n) \geq logn$, $ASPACE(f(n)) \supseteq TIME(2^{O(f(n))})$

Proof.

We show how to simulate a deterministic time $2^{O(f(n))}$ machine M by an alternating Turing machine S that uses space $O(f(n))$

- We start by building a $2^{O(f(n))} \times 2^{O(f(n))}$ tableau to store pointers for M on input ω.
- To verify the contents of a cell d outside the first row
  - S existentially guesses the contents of the parents
  - checks whether their contents would yield d's contents according to M's transition function
  - universally branches to verify these guesses recursively
- S verifies the first row directly because it knows M's starting configuration.
- S never needs to store more than a single pointer to a cell in the tableau, so it uses space $log 2^{O(f(n))} = O(f(n))$

Interactive Proof Systems and Parallel Computation

Interactive Proof Systems

- The languages in NP are those whose members all have short certificates of membership that can be easily checked
- Let's rephrase this formulation by creating two entities: a Prover that finds the proofs of membership, and a Verifier that checks them.
- A prover can convince a Verifier for the SAT problem.
- The complement of SAT is not known to be in NP, so we can't rely on the certificate idea.
  - Prover and Verifier should be permitted to engage in a two-way dialog.
  - Verifier may be a probabilistic polynomial time machine that reaches the correct answer with a high degree of, but not absolute, certainty
  - Such a Prover and Verifier constitute an interactive proof system.

Interactive Proof Systems

We define the Verifier to be a function V that computes its next transmission to the Prover from the message history sent so far. The function V has three inputs

1. Input string
2. Random input.
3. Partial message history as $m_1\#m_2\#\ldots\#m_i$

The Prover is a party with unlimited computational ability. We define it to be a function P with two inputs:

1. Input string
2. Partial message history.

Graph nonisomorphism

- Graph isomorphism is in NP but we don't know if it is NP-complete.
- NONISO is not known to be in NP because we don't know how to provide short certificates that graphs aren't isomorphic.
- Nonetheless, when two graphs aren't isomorphic, a Prover can convince a Verifier of this fact
  - The Verifier randomly selects one of the graphs and then randomly reorders its nodes and send the result to the Prover.
  - If the graphs were indeed nonisomorphic, the Prover could always carry out the protocol
  - If the graphs were isomorphic the Prover would have no better than a 50-50 chance of getting the correct answer.
- If the Prover is able to answer correctly consistently, the Verifier has convincing evidence that the graphs are actually nonisomorphic.

Graph nonisomorphism

Definition

A language L is in IP complexity if some P-time computable function V exists such that for an arbitrary function P and for every function $\widehat{P}$ and for every string ω

1. $\omega \in L$ implies $Pr[V \leftrightarrow P \text{ accepts } \omega] \geq \frac{2}{3}$ and
2. $\omega \notin L$ implies $Pr[V \leftrightarrow \widehat{P} \text{ accepts } \omega] \leq \frac{1}{3}$

In other words

- If $\omega \in L$ an "honest" Prover causes the Verifier to accept with high probability
- If $\omega \notin L$ even a "crooked" Prover cannot cause the Verifier to accept with high probability.

IP contains both NP and BPP and the language NONISO.

IP Class

Theorem

$IP = PSPACE$

Definition

$\#SAT = \{\langle \phi, k \rangle | \phi$ is a cnf formula with exactly k satisfying assignments$\}$

Theorem

$\#SAT \in IP$

Uniform Boolean Circuits

- Boolean circuit model of a parallel computer, we take each gate to be an individual processor, so we define the processor complexity of a Boolean circuit to be its size
- We consider each processor to compute its function in a single time step, so we define the parallel time complexity of a Boolean circuit to be its depth.
- We use circuit families for deciding languages. We need to impose a technical requirement on circuit families so that a single machine is capable of handling all input lengths.

Definition

A family of circuits $(C_0, C_1, C_2, \ldots)$ is uniform if some log space transducer T outputs $\langle C_n \rangle$ when T's input is 1^n.

Size-Depth Complexity

- A language has simultaneous size–depth circuit complexity at most $(f(n), g(n))$ if a uniform circuit family exists for that language with size complexity $f(n)$ and depth complexity $g(n)$.
- We consider the simultaneous size and depth of a single circuit family in order to identify how many processors we need in order to achieve a particular parallel time complexity or vice versa.

Example

Let A be the language over 0,1 consisting of all strings with an odd number of 1s. We can test membership in A by computing the parity function.

The size-depth complexity of A is $(O(n), O(logn))$

Size-Depth Complexity

Example

For boolean matrix multiplication, we may use an input of $2m^2 = n$ variables for two $m \times m$ matrices a and b. The output is m^2 values representing the $m \times m$ resulting matrix c.

The circuit for this function has gates g_{ijk} that compute $a_{ij} \wedge b_{jk}$. Additionally, for each i and k, the circuit contains a binary tree of $\vee$ gates to compute $\bigvee_j g_{ijk}$.

Each such tree contains $m - 1$ OR gates and has $logm$ depth. Consequently, these circuits for Boolean matrix multiplication have size $O(m^3) = O(n^{3/2})$ and depth $O(logn)$

Size-Depth Complexity

Example

If $A = \{a_{ij}\}$ is an $m \times m$ matrix, the transtivie closure of A is the matrix $A \vee A^2 \vee \ldots \vee A^m$

The transitive closure operation is closely related to the PATH problem and hence to the class NL

We can represent the computation of A^i with a binary tree of size i and depth $logi$ wherein a node computes the product of the two matrices below it. The circuit computing A^m has size $O(n^2)$ and depth $O(log^2 n)$

A separate circuit exists for each A^i which adds another factor of m to the size and an additional layer of $O(logn)$ depth. Hence the size–depth complexity of transitive closure is $(O(n^{5/2}), O(log^2 n))$

NC Complexity Class

Definition

- For $i \geq 1$, NC^i is the class of languages that can be decided by a uniform family of circuits with polynomial size and $O(log^i n)$ depth.
- NC is the class of languages that are in NC^i for some i.
- Functions that are computed by such circuit families are called NC^i computable or NC computable.

- NC class problems may be considered to be highly parallelizable with a moderate number of processors.
- Steven Cook coined the name NC for "Nick's class" because Nick Pippenger was the first person to recognize its importance.

Theorem

$NC \subseteq L$

Proof.

- We sketch a log space algorithm to decide a language A in NC^1
- On input ω of length n, the algorithm can construct the description as needed of the nth circuit in the uniform circuit family for A.
- Then the algorithm can evaluate the circuit by using a depth-first search from the output gate
- Memory for this search is needed only to record the path to the currently explored gate, and to record any partial results that have been obtained along that path.
- The circuit has logarithmic depth; hence only logarithmic space is required by the simulation.

$\square$

Theorem

$NL \subseteq NC^2$

Proof.

- Compute the transitive closure of the graph of configurations of an NL-machine.
- Output the position corresponding to the presence of a path from the start configuration to the accept configuration.

□

Theorem

NC subseteq P

Proof.

A polynomial time algorithm can run the log space transducer to generate circuit C_n and simulate it on an input of length n. $\square$

P-Completeness

The phenomenon of P-completeness can give theoretical evidence that some problems in P are inherently sequential.

Definition

A language B is P-complete if $B \in P$ and every A in P is log space reducible to B.

Theorem

If A is log space reducible to B and B is in NC, then A is in NC.

P-Completeness

Definition

For a circuit C and input setting x, we write $C(x)$ to be the value of C on x. CIRCUIT-VALUE problem desides if $C(x) = 1$

Theorem

CIRCUIT-VALUE is P-complete

Proof.

- Any language A in P can be reduced to CIRCUIT-VALUE
- On input ω, the reduction can be performed by constructing a circuit that simulates the polynomial time Turing machine for A
- The input to the circuit is ω itself.
- The reduction can be carried out in log space.

$\square$