Homomorphisms

Let Σ and Δ be two alphabets. A **homomorphisms** over Σ is a function $h : \Sigma \rightarrow \Delta^*$

Informally, a homomorphism is a function which replaces each symbol with a string

Example : Let $\Sigma = \{0, 1\}$ and define $h(0) = ab$, $h(1) = \epsilon$; h is a homomorphism over Σ

Homomorphisms

We extend h to Σ^* : if $w = a_1 a_2 \cdots a_n$ then

$$h(w) = h(a_1)h(a_2)\cdots h(a_n)$$

Equivalently, we can use a **recursive** definition :

$$h(w) = \begin{cases} \epsilon, & \text{if } w = \epsilon; \\ h(x)h(a) & \text{if } w = xa,\ x \in \Sigma^*,\ a \in \Sigma. \end{cases}$$

Example : Using h from previous example on string 01001 results in *ababab*

Homomorphisms

For a language $L \subseteq \Sigma^*$

$$h(L) = \{h(w) \mid w \in L\}$$

Example : Let L be the language associated with the regular expression $\mathbf{10^*1}$. Then $h(L)$ is the language associated with the regular expression $(\boldsymbol{ab})^*$

Closure under homomorphism

Theorem Let $L \subseteq \Sigma^*$ be a regular language and let h be a homomorphisms over Σ. Then $h(L)$ is a regular language

Proof Let E be a regular expression generating L. We define $h(E)$ as the regular expression obtained by substituting in E each symbol $\boldsymbol{a}$ with $\boldsymbol{a_1 a_2 \cdots a_k}$, under the assumption that

And leaving $\varepsilon, \emptyset, +, \cdot, *$ as they are

- $a \in \Sigma$
- $h(a) = a_1 a_2 \cdots a_k$, $k \geqslant 0$

We now prove the statement

$$L(h(E)) = h(L(E)),$$

using structural induction on E

Closure under homomorphism

Base $E = \epsilon$ or else $E = \emptyset$. Then $h(E) = E$, and $L(h(E)) = L(E) = h(L(E))$ → Nothing gets replaced

$E = \boldsymbol{a}$ with $a \in \Sigma$. Let $h(a) = a_1 a_2 \cdots a_k$, $k \geqslant 0$. Then $L(\boldsymbol{a}) = \{a\}$ and thus $h(L(\boldsymbol{a})) = \{a_1 a_2 \cdots a_k\}$

The regular expression $h(\boldsymbol{a})$ is $\boldsymbol{a_1 a_2 \cdots a_k}$. Then $L(h(\boldsymbol{a})) = \{a_1 a_2 \cdots a_k\} = h(L(\boldsymbol{a}))$

Closure under homomorphism

Induction Let $E = F + G$. We can write

$$
\begin{array}{rcll}
L(h(E)) & = & L(h(F+G)) & \\
 & = & L(h(F)+h(G)) & h \text{ defined over regex} \\
 & = & L(h(F)) \cup L(h(G)) & + \text{ definition} \\
 & \overset{\text{induction}}{=} & h(L(F)) \cup h(L(G)) & \text{inductive hypothesis for } F, G \\
 & = & h(L(F) \cup L(G)) & h \text{ defined over languages} \\
 & = & h(L(F+G)) & + \text{ definition} \\
 & = & h(L(E)) &
\end{array}
$$

Closure under homomorphism

Let $E = F.G$. We can write

induction

$$
\begin{aligned}
L(h(E)) &= L(h(F.G)) && \\
&= L(h(F).h(G)) && h \text{ defined over regex} \\
&= L(h(F)).L(h(G)) && . \text{ definition} \\
&= h(L(F)).h(L(G)) && \text{inductive hypothesis for } F, G \\
&= h(L(F).L(G)) && h \text{ defined over languages} \\
&= h(L(F.G)) && . \text{ definition} \\
&= h(L(E))
\end{aligned}
$$

Closure under homomorphism

Let $E = F^*$. We can write

$$
\begin{array}{rcll}
L(h(E)) & = & L(h(F^*)) & \\
& = & L([h(F)]^*) & h \text{ defined over regex} \\
& = & \bigcup_{k \geqslant 0} [L(h(F))]^k & * \text{ definition} \\
& \overset{\text{induction}}{=} & \bigcup_{k \geqslant 0} [h(L(F))]^k & \text{inductive hypothesis for } F \\
& = & \bigcup_{k \geqslant 0} \; h([L(F)]^k) & h \text{ definition over languages} \\
& = & h(\bigcup_{k \geqslant 0} [L(F)]^k) & h \text{ definition over languages} \\
& = & h(L(F^*)) & * \text{ definition} \\
& = & h(L(E)) &
\end{array}
$$

$\square$

Conversion complexity

We can convert among DFA, NFA, ϵ-NFA, and regular expressions

What is the **computational complexity** of these conversions?

We investigate the computational complexity as a function of

- number of states n for an FA
- number of operators n for a regular expressions
- we assume $|\Sigma|$ is a constant

From ϵ-NFA to DFA

Suppose an ϵ-NFA has n states. To compute ECLOSE(p) we visit at most n^2 arcs. We do this for n states, resulting in time $\mathcal{O}(n^3)$

The resulting DFA has 2^n states. For each state S and each $a \in \Sigma$ we compute $\delta(S, a)$ in time $\mathcal{O}(n^3)$. In total, the computation takes $\mathcal{O}(n^3 \cdot 2^n)$ steps, that is, **exponential time**

If we compute δ just for the **reachable** states

- we need to compute $\delta(S, a)$ s times only, with s the number of reachable states
- in total the computation takes $\mathcal{O}(n^3 \cdot s)$ steps

↳ still exponential

Other conversions

From NFA to DFA : computation takes **exponential time**

From DFA to NFA :

- put set brackets around the states
- computation takes time $\mathcal{O}(n)$, that is, **linear time**

From FA to regular expression via state elimination construction: computation takes **exponential time**

Other conversions

From regular expression to ϵ-NFA :

- construct a tree representing the structure of the regular expression in time $\mathcal{O}(n)$
- at each node in the tree, we build new nodes and arcs in time $\mathcal{O}(1)$ and use **pointers** to previously built structure, avoiding copying
- grand total time is $\mathcal{O}(n)$, that is, **linear time**

Decision problems

No proofs needed

In the problem instances below, languages L and M are expressed in any of the four representations introduced for regular languages

- $L = \emptyset$?
- $w \in L$?
- $L = M$?

Empty language

$L(A) \neq \emptyset$ for FA A if and only if at least one final state is **reachable** from the initial state of A

Algorithm for computing reachable states :

Base The initial state is reachable

Induction If q is reachable and there exists a transition from q to p, then p is reachable

Computation takes time proportional to the number of arcs in A, thus $\mathcal{O}(n^2)$ → Polynomial time

We already saw this idea in the lazy evaluation for translating NFA into DFA

Empty language

Given a regular expression E, we can decide $L(E) \stackrel{?}{=} \emptyset$ by structural induction

Base

- $E = \epsilon$ or else $E = \boldsymbol{a}$. Then $L(E)$ is non-empty
- $E = \emptyset$. Then $L(E)$ is empty

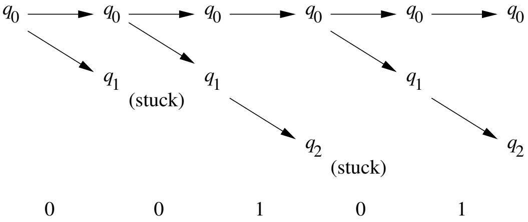
Induction

- $E = F + G$. Then $L(E)$ is empty if and only if both $L(F)$ and $L(G)$ are empty
- $E = F.G$. Then $L(E)$ is empty if and only if either $L(F)$ or $L(G)$ are empty
- $E = F^*$. Then $L(E)$ is not empty, since $\epsilon \in L(E)$

Language membership

We can test $w \in L(A)$ for DFA A by simulating A on w. If $|w| = n$ this takes $\mathcal{O}(n)$ steps

If A is an NFA with s states, simulating A on w requires $\mathcal{O}(n \cdot s^2)$ steps



Language membership

If A is an ϵ-NFA with s states, simulating A on w requires $\mathcal{O}(n \cdot s^3)$ steps

Alternatively, we can pre-process A by calculating ECLOSE(p) for s states, in time $\mathcal{O}(s^3)$. Afterwards, the simulation of each symbol a from w is carried out as follows

- from the current states, find the successor states under a in time $\mathcal{O}(s^2)$
- compute the ϵ-closure for the successor states in time $\mathcal{O}(s^2)$

This takes time $\mathcal{O}(n \cdot s^2)$

Language membership

If $L = L(E)$, for some regular expression E of length s, we first convert E into an ϵ-NFA with $2s$ states. Then we simulate w on this automaton, in $\mathcal{O}(n \cdot s^3)$ steps

Language membership

We can convert an NFA or an ϵ-NFA into a DFA, and then simulate the input string in time $\mathcal{O}(n)$

The time required by the conversion could be **exponential** in the size of the input FA

This method is used

- when the FA has small size
- when one needs to process several strings for membership with the same FA

Equivalent states

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and let $p, q \in Q$. We define

$$p \equiv q \;\Leftrightarrow\; \forall w \in \Sigma^* \;:\; \hat{\delta}(p, w) \in F \text{ if and only if } \hat{\delta}(q, w) \in F$$

In words, we require p, q to have equal response to input strings, with respect to acceptance

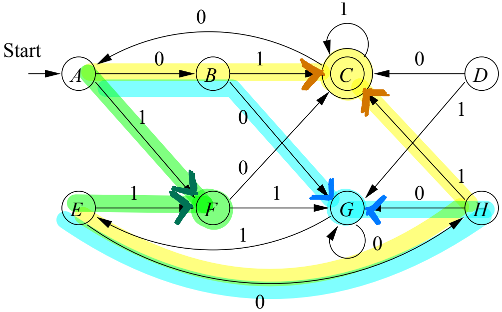
If $p \equiv q$ we say that p and q are **equivalent** states

If $p \not\equiv q$ we say that p and q are **distinguishable** states

Equivalently : p and q are distinguishable if and only if

$$\exists w \;:\; \hat{\delta}(p, w) \in F \text{ and } \hat{\delta}(q, w) \notin F, \text{ or the other way around}$$

Example



1 ... =
00.. =
01.. =

since after 1 or 2 iterations they always end up in the same state, all further actions will have the same outcome
$A \equiv E$

$\hat{\delta}(C,\epsilon) \in \mathcal{F},\ \hat{\delta}(G,\epsilon) \notin \mathcal{F} \ \Rightarrow\ C \not\equiv G$ ($\mathcal{F}$ finale states)

$\hat{\delta}(A,01) = C \in \mathcal{F},\ \hat{\delta}(G,01) = E \notin \mathcal{F} \ \Rightarrow\ A \not\equiv G$

Example

We prove $A \equiv E$

$\hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)$. Thus $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)$. Thus $\hat{\delta}(A, 00x) = \hat{\delta}(E, 00x) = \hat{\delta}(G, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)$. Thus $\hat{\delta}(A, 01x) = \hat{\delta}(E, 01x) = \hat{\delta}(C, x)$, $\forall x \in \{0, 1\}^*$

State equivalence algorithm

We can compute distinguishable state pairs using the following recursive relation

Base If $p \in F$ and $q \notin F$, then $p \not\equiv q$

Induction If $\exists a \in \Sigma \,:\, \delta(p, a) \not\equiv \delta(q, a)$, then $p \not\equiv q$

We compute distinguishable states by backward propagation

State equivalence algorithm

Apply the recursive relation using an **adjacency table** and the following dynamic programming algorithm

- initialize table with pairs that are distinguishable by string ϵ
- for all not yet visited pairs, try to distinguish them using one symbol string: if you reach a pair of **already** distinguishable states, then update table
- iterate until no new pair can be distinguished

Example

$$\exists a \in \Sigma \; : \; \delta(p,a) \not\equiv \delta(q,a)$$
$$\Rightarrow p \not\equiv q$$

	A	B	C	D	E	F	G
B	x_1						
C	x_0	x_0					
D	x_1	x_1	x_0				
E		x_1	x_0	x_1			
F	x_1	x_1	x_0		x_1		
G	$(F,E)_1$ x_1	$(E,C)_1$ x_1	x_0	x_1	x_1	x_1	
H	(F,C) x_1		x_0	$(G,C)_0$ x_1	$(F,C)_1$ x_1	x_1	$(E,C)_1$ x_1