TCS - Appunti Theoretical Computer Science

Theoretical computer science (Politecnico di Milano)

THEORETICAL COMPUTER SCIENCE

1. MODELLI IN INGEGNERIA INFORMATICA

In ingegneria informatica facciamo ampio uso di **modelli** per descrivere molti aspetti. I modelli possono essere di due tipi **Operazionali** e **Descrittivi**, per esempio, il modello operazionale di un'operazione di ordinamento di un array è "trova il minimo degli elementi e mettilo in prima posizione, trova il minimo degli elementi rimasti e mettilo in seconda ecc…", mentre il modello descrittivo è la formula matematica: $\forall i, a[i] \leq a[i+1]$

Una prima nozione fondamentale di un modello è il **linguaggio**, definito da **stringhe,** cioè sequenze ordinate degli elementi di un alfabeto A.
La lunghezza di una stringa è descritta come: $|a| = 1, |ab| = 2$
La stringa nulla è invece indicata con il simbolo **ε** e, ovviamente, la sua lunghezza sarà 0, $|\boldsymbol{\varepsilon}| = \mathbf{0}$.
Indichiamo con A*, l'insieme di tutte le stringhe di A, compresa la stringa nulla. Se per esempio l'alfabeto è definito come A={0,1}, l'insieme delle stringhe sarà A*={ε, 0, 1, 00, 01, 10, …}.
Operazione con le stringhe:

- **Concatenazione** (prodotto), associativa ma non commutativa: x=abb, y=baba, x · y = abbbaba. A* è chiamato **monoide libero** rispetto a questa operazione e ε è l'**operatore unitario**, cioè, εx=xε=x

Il **linguaggio è un sottoinsieme di A***, quindi un insieme di stringhe di un alfabeto ($L \subseteq A^*$).
Operazioni con i linguaggi:

- Unione ∪, intersezione ∩, differenza L1 - L2, negazione logica ¬L = A* - L .
- Concatenazione L1 · L2 = { x · y | x ∈ L 1, y ∈ L2 }
- Elevazione a potenza (language power) $L^0 = \{\varepsilon\}, \ \forall i > 0, L^i = L^{i-1} \cdot L \ L^* = \bigcup_{n=0}^{\infty} L^n$
  - NB: $\{\varepsilon\} \neq \emptyset, \{\varepsilon\} \cdot L = L, \ \emptyset \cdot L = \emptyset$
  - $L^+ = \bigcup_{n=1}^{\infty} L^n = L^* - \{\varepsilon\} \ con \ \varepsilon \notin L$

Ogni stringa può essere trasformata in un numero naturale tramite l'operazione di traslazione:

- Translazione $y = \boldsymbol{\tau}(\boldsymbol{x})$**,** funzione che trasla una stringa da L1 a L2
  Esempio: **τ**1: duplica "1" (1->11), 0010110 -> 0011011110

Possiamo quindi concludere che qualsiasi tipo di sistema può essere descritto dall'uso di un linguaggio e di operazioni su di esso. È importante ricordare che in un computer ogni informazione è una stringa di bits.

2. MODELLI OPERAZIONALI

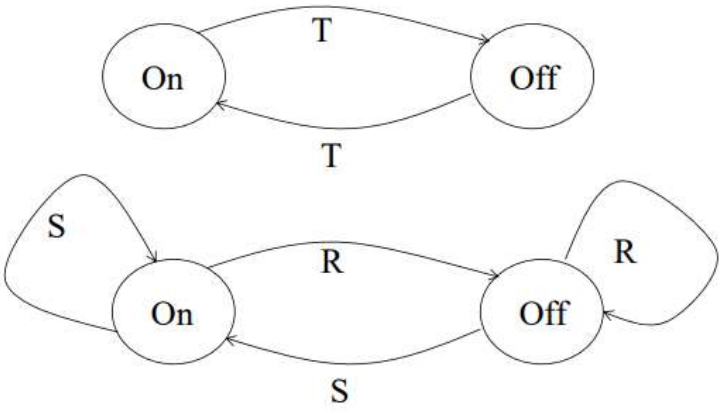
Una **macchina a stati** finiti (Finite State Machines or automata) FSA, or FA rappresenta un insieme di stati finiti e il passaggio tra di essi tramite archi orientati (frecce) e bolle (circonferenze).

Finite State Machines (or *automata*) (*FSA*, or *FA*):

– A finite state set:
{ on, off }, ….
{1, 2, 3, 4, …k}, {TV channels}, …

Graphic representation:

Two very simple flip-flops:



Turning a light on and off, ...

In particolare, è formata da:

- Un insieme di stati finiti: Q
- Un alfabeto finito in input: I
- Una funzione di transizione: $\delta: Q \times I \rightarrow Q$

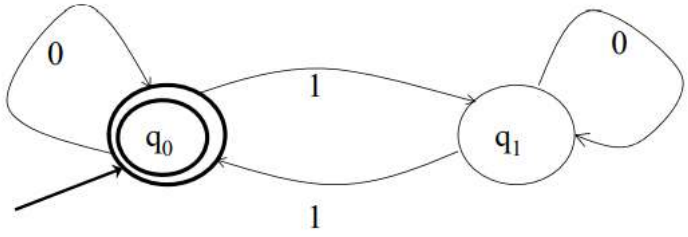
Un automata può essere utilizzato come riconoscitore di stati (o accettore), infatti, una sequenza di stati parte da quello iniziale (q0) e arriva a quello finale ($F \subseteq Q$, F insieme degli stati finali, Q, insieme degli stati) solo se il linguaggio è accettato (la stringa x è accettata e quindi appartenente al linguaggio se è verificata la seguente condizione:

$x \in L \leftrightarrow \delta * (q0, x) \in F$.

Esempio: L = {stringhe che sono formate da un qualsiasi numero di "1" e di "0"}

- A *move sequence* starts from an ***initial state*** and it is *accepting* if it reaches a ***final*** (or *accepting*) *state*.

L = {strings with an even number of "1" any number of "0"}



Formalization of the notion of acceptance of language ***L***

Move sequence for an input ***string*** (not just one symbol):

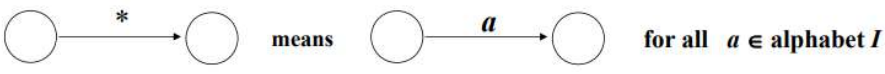
- $\delta^*: Q \times I^* \rightarrow Q$
  δ^* defined inductively from δ, by induction on the string length
  $\delta^* (q, \varepsilon) = q$
  $\delta^* (q, y \cdot i) = \delta(\delta^* (q,y), i)$

Initial state: $q_0 \in Q$

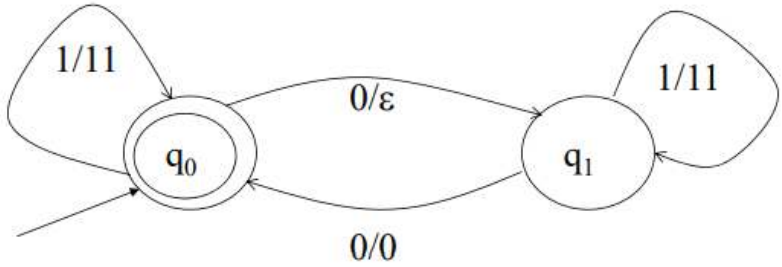
Set of final, or accepting states: $F \subseteq Q$

Acceptance: $x \in L \leftrightarrow \delta^* (q_0, x) \in F$

a useful notation/shorthand that is occasionally adopted



Inoltre, l'automata può essere utilizzato per rappresentare un translatore di linguaggi $y = \boldsymbol{\tau}(\boldsymbol{x})$
Esempio: τ: raddoppia il numero di "1" e accetta soltanto input che hanno un numero dispari di "0"



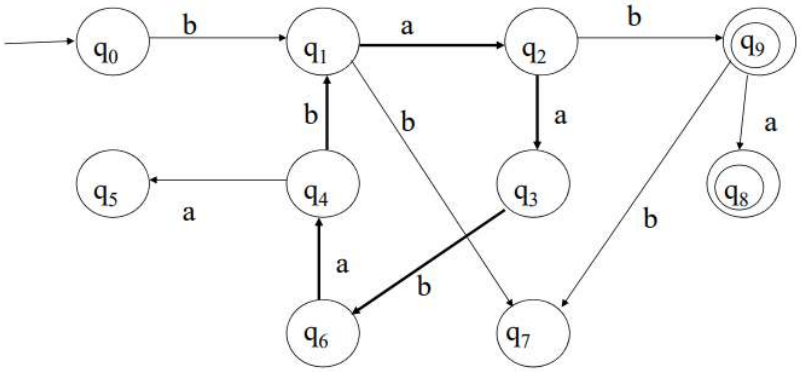
(NB: il valore dopo il / indica l'output che si avrà al ricevimento del corrispettivo input)

Formalization of translating automata (transducers)

$T = <Q, I, \delta, q_0, F, O, \eta>$
- $<Q, I, \delta, q_0, F>$: just like for acceptors
- O: output alphabet
- $\eta : Q \times I \rightarrow O^*$ (NB: the output is a string)

$\eta^* : Q \times I^* \rightarrow O^*$, η^* defined inductively as usual
$\eta^*(q, \varepsilon) = \varepsilon$
$\eta^*(q, y \cdot i) = \eta^*(q, y) \cdot \eta(\delta^*(q,y), i)$
definition of the translation [provided that…]
$\tau(x)\ [x \in L] = \eta^*(q_0, x)\ [\delta^*(q_0, x) \in F]$

Una prima fondamentale proprietà di una FA è il comportamento ciclico (cyclic behavior). Se leggendo una stringa l'automata partendo da uno stato si ritrova di nuovo su di esso abbiamo un ciclo.



There is a cycle q_1 ----***aabab***---> q_1

If, when reading a string, one goes through the cycle once, then one can also go through it 2, 3, …, n, … times

Formalmente se x ∈ L and |x| > |Q| e esistono q ∈ Q e w ∈ I^+ tali che x=ywz e δ∗ (q, w) = q allora yw^nz ∈ L, ∀ n ≥ 0. **Lemma di Pumping**
Conseguenze del Pumping Lemma:

- |L| = ∞. Un linguaggio ha lunghezza infinita se e solo se esiste un ciclo nel grafico con un nodo nel percorso tra lo stato iniziale e quello finale
- L = ∅. Un linguaggio è vuoto se esiste ∃ x ∈ L ↔ ∃ y ∈ L, |y| < |Q|. Cioè se non ci sono percorsi dallo stato iniziale a quello finale
- Non è possibile contare usando un FA, perché la memoria di un computer deve essere infinita tale da contenere tutte le stringhe che manipola e mette in memoria. Per questo motivo occorrono modelli più performanti rispetto a un FA.

3. Proprietà di chiusura di una FA (Closure properties)

La nozione matematica di chiusura per un linguaggio ci dice che presa una famiglia di linguaggi L = {Li} diremo che L è chiusa rispetto a un'operazione OP se e solo se L1 OP L2 $\in$ L.
L'insieme della famiglia dei linguaggi regolari R, quindi l'insieme dei linguaggi accettati da una FA, è chiusa rispetto alle operazioni teoretiche, quali concatenazione, intersezione, complemento e unione.

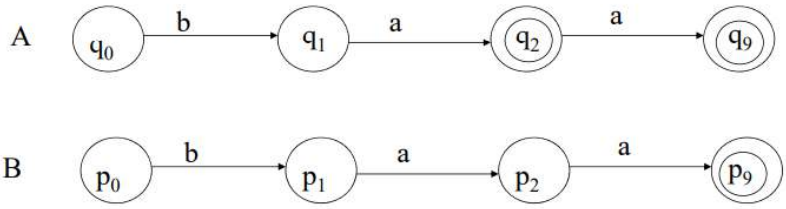
$\mathcal{L} = \{L_i\}$: a *family* of languages

$\mathcal{L}$ is closed under OP if and only if (iff)
for every $L_1, L_2 \in \mathcal{L}$, L_1 OP $L_2 \in \mathcal{L}$

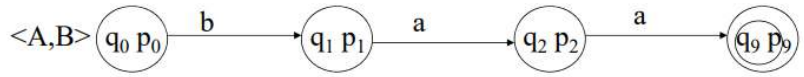
$\mathcal{R}$: *regular languages*, those accepted by an FA

$\mathcal{R}$ is closed under set theoretic operations, concatenation, "*", … and virtually "all others"

Intersection



One can simulate the "parallel run" of A and B by simply "coupling them":



Formally:

Given $A^1 = <Q^1, I, \delta^1, q_0{}^1, F^1>$ and
$A^2 = <Q^2, I, \delta^2, q_0{}^2, F^2>$

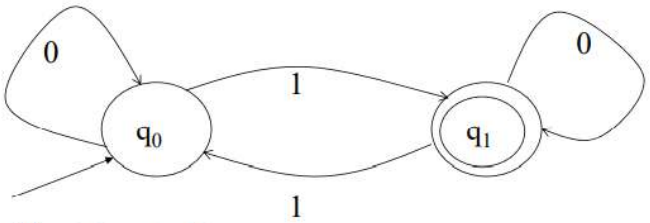
The automaton $< A^1, A^2 >$ is defined as:
$<Q^1 \times Q^2, I, \delta, <q_0{}^1, q_0{}^2>, F^1 \times F^2>$
$\delta(<q^1, q^2>, i) = <\delta^1(q^1, i), \delta^2(q^2, i)>$

One can show by a simple induction that
$L(< A^1, A^2 >) = L(A^1) \cap L(A^2)$

Union

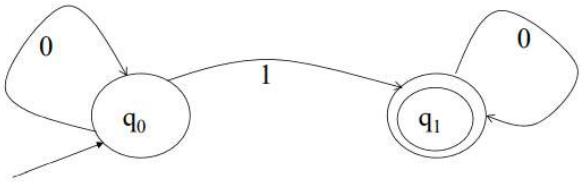
- A similar construction …
  otherwise ... exploit identity $A \cup B = \neg(\neg A \cap \neg B)$
  ⇒ need a FA for the complement language

Complement: example automaton accepting
binary strings x with odd number of 1's



An idea: F^ = Q - F:

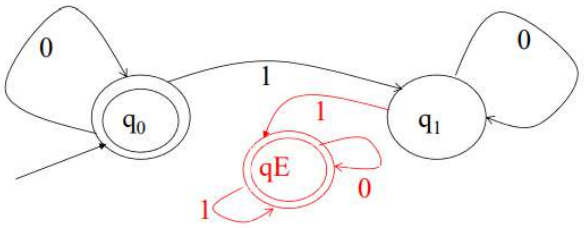
Yes, it works for the automaton above, but ….



L: strings with exactly one '1'

The "complement F construction", i.e., turning F into Q – F, doesn't work

Why? because δ is partial



¬L: strings with no '1' or with more than one '1'

Some general remarks on the complement

If the analysis of the string terminates for all possible strings then it suffices the "turn a yes into a no" (F into Q-F)

If, for some string, the analysis of the string does not terminate (it gets blocked or continues forever) then turning F into Q-F does not work

With FA the problem is easily solved …

In general if we are unable to provide a positive answer to a problem (e.g., $x \in L$), this does not necessarily mean we can provide a positive answer for the complement problem (e.g., $x \in \neg L$)

4. Teorema di Myhill-Nerode

Un linguaggio L è accettato da una **DFA** (*deterministic finite automaton*, automa a stati finiti dove per ogni coppia di stato e simbolo in ingresso c'è una e una sola transizione allo stato successivo) se e solo se le classi di equivalenza sono finite.

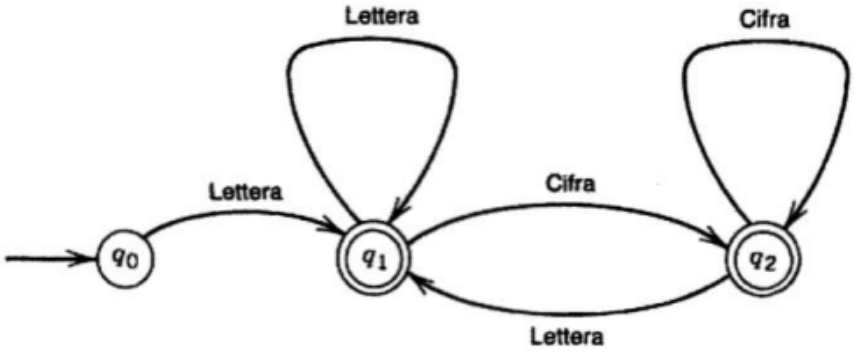
Definiamo x e y indistinguibili ($x \approx_L y$) se e solo se $\forall z \in I^*$, $x \cdot z \in L \Leftrightarrow y \cdot z \in L$

– note that $\approx_L$ is an equivalence relation ('$\Leftrightarrow$' is a sort of equality)
– hence I^* is *partitioned* by $\approx_L$ into equivalence classes

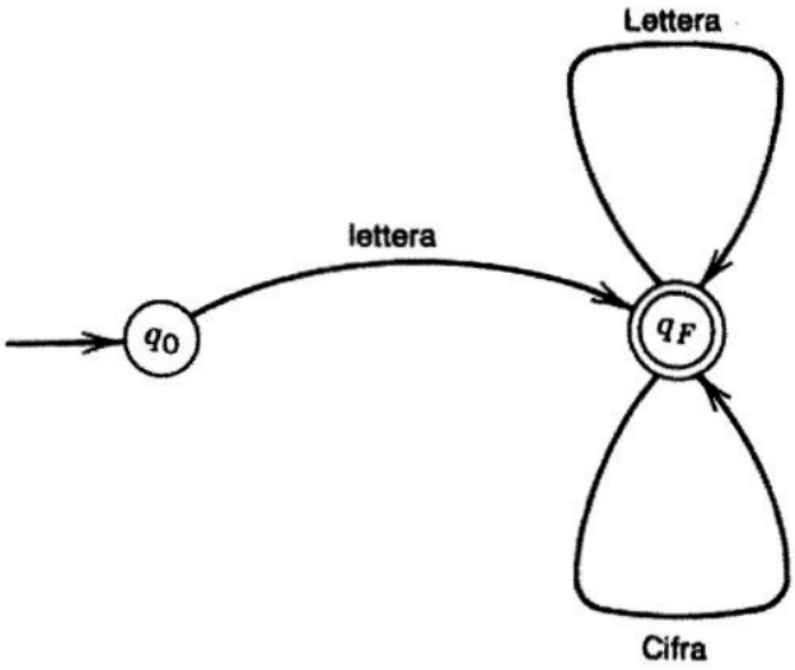
the cardinality of the equiv. classes of $\approx_L$ may be

– **finite**. Example: $L = \{ x \in \{a, b\}^* \mid$ no two adjacent symbols of x are equal $\}$
  • {ε}, {strings ending with a}, {strings ending with b}, {strings with two equal adjacent symbols}
– **infinite**. Example: $L = \{ a^n b^n \mid n \geq 0 \}$
  • {ε}, {a}, {aa}, {aaa}, {$aaaa$}, … {a^n}, … and others for strings containing b's

Il teorema ci dice, in parole più semplici, che, sia A un automa a stati finiti, è possibile minimizzarlo, cioè avere una cardinalità minore o uguale a quella di A a meno di un isoformismo, cioè un riassegnamento dei nomi agli stati e questa minimizzazione è unica e prende il nome di riconoscitore canonico del linguaggio. Per applicare il teorema a un automa, basta eliminare gli stati inutili, cioè gli stati che sono equivalenti ad altri e rinominare gli stati ottenuti.



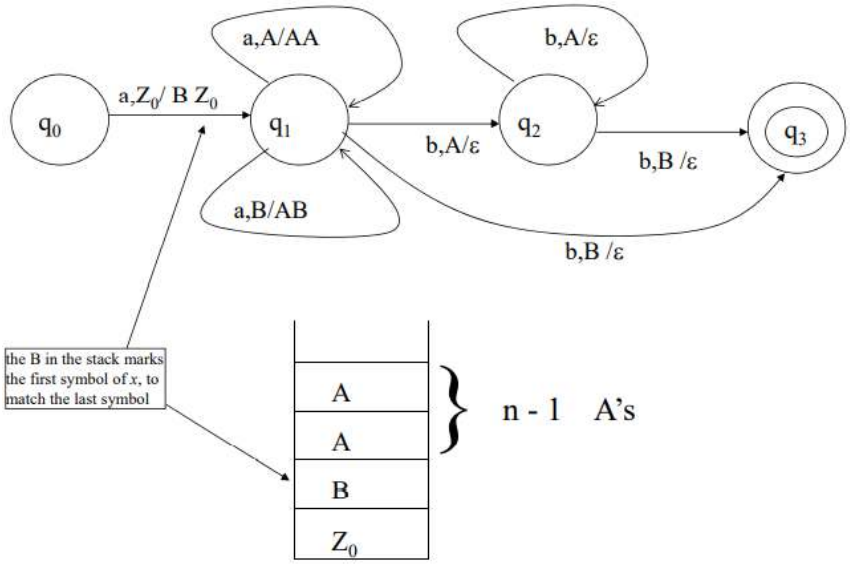
DIVENTA:



5. Allocazione in memoria di una FA e le Pushdown Automata (PA), definizioni generali

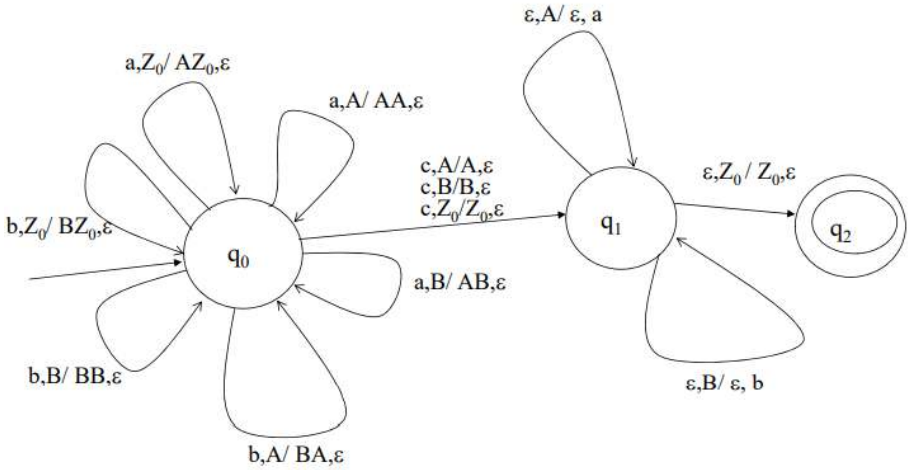
L'allocazione in memoria di una FA avviene tramite due tape (liste/nastri), una di input e una di output, un dispositivo di controllo e una memoria stack dove sono memorizzati gli stati.
La PA è il cuore di un compilatore e funziona a svuotamento della stringa in input e in base allo stato di partenza dello stack scrive lo stato di destinazione. Lo stato finale viene raggiunto spesso tramite la ε-move, mossa spontanea.

A first example: accepting $\{a^n b^n \mid n > 0\}$



Un altro esempio:

It reverses a string: $\tau(wc)=w^R$, $\forall w \in \{a,b\}^+$

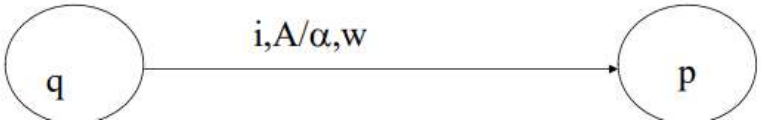


Definizione di una PA <Q, I, Γ, δ, q_0, Z_0, F [, O, η]> dove, Q, I, q_0 e F[0] assumono lo stesso significato delle FA. Γ è lo stack alphabet, Z_0 lo stato iniziale dello stack.

$\delta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ δ is partial just as in FSA

$\eta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow O^*$ (η defined where δ is)

Graphical notation: $<p,\alpha> = \delta(q,i,A)$, $w = \eta(q,i,A)$



<p, α> = δ(q, i, A)

<stato finale, output> = δ (stato iniziale, input, stato iniziale stack)

w = η(q, i, A)

stato finale stack = η(stato iniziale, input, stato iniziale stack)

Configurazione:

c = <q, x, γ, [z]> dove q è lo stato di controllo del dispositivo, x la porzione della stringa in input che ancora si deve leggere, γ la stringa dei simboli nello stack, z la stringa scritta nell'output finale.

6. Relazione di transizione di una PA

La relazione di una transizione di una PA, che ci indica come avviene il passaggio da uno stato all'altro, è indicata con "|--".

c = <q, x, γ, [z]> |-- c' =<q', x', γ, [z']>

La relazione è formata soltanto da due casi, il primo, caso ordinario, si verifica quando viene "consumata" (svuotata) la stringa di input e il secondo in cui si raggiunge lo stato finale (ε-move, mossa spontanea). **Si noti che il caso 2 è soltanto un'alternativa a ogni caso 1.**

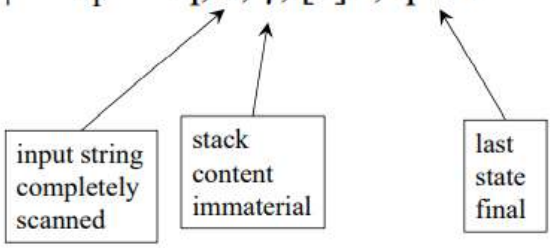
CASO 1: x = i.y e δ(q, i, A) =<q', α> [η(q, i, A) = w] dove x' = y, γ'= $\alpha\beta$, [z' = z.w]

CASO 2: δ(q, ε, A) = <q', α> [η(q, ε, A) = w] dove x' = x, γ'= $\alpha\beta$, [z' = z.w]

La **relazione di chiusura** della relazione di transizione è indicata con "|-*-" ed è riflessiva e transitiva. Indica il numero positivo di "steps" che compie la transizione di relazione per passare dallo stato iniziale c0 allo stato finale cF.

$$x \in L\ [z = \tau(x)] \leftrightarrow$$

$$c_0 = <q_0, x, Z_0, [\varepsilon]> \text{ |-*- } c_F = <q, \varepsilon, \gamma, [z]>,\ q \in F$$



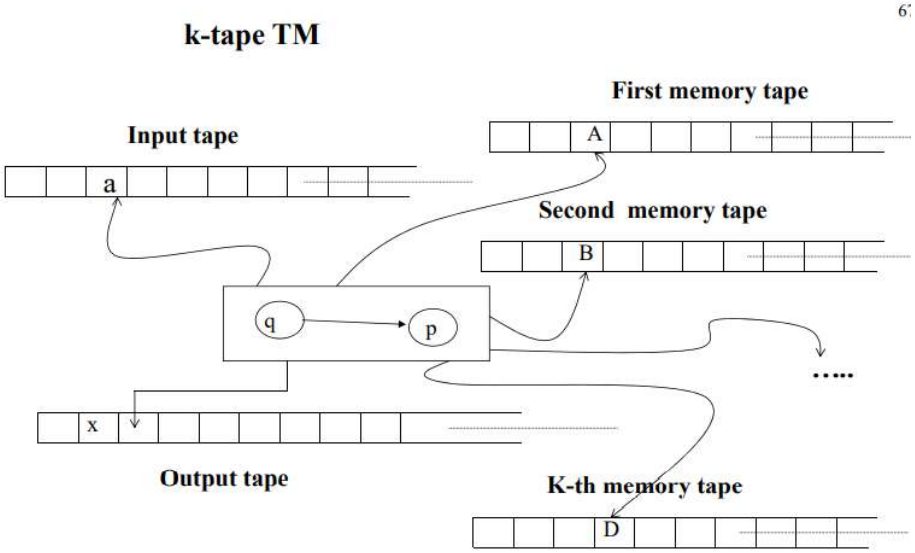
Notiamo che ε indica che la stringa in input è stata completamente consumata e che q è lo stato finale

7. Proprietà di una PA

Indichiamo con LP l'insieme delle classi di linguaggi accettati da una PA. LP non è chiuso sotto le relazioni di unione e intersezione. Considerando però il principio di relazione complementare, già usata nelle FA, sappiamo che questa cambia gli stati accettati in quelli non accettati, troviamo però delle nuove problematiche. La mossa spontanea potrebbe causare cicli e quindi non si avrebbe mai lo svuotamento completo della stringa in input e quindi la stringa non verrebbe accettata. Esiste però una costruzione che associa a ogni automata una equivalente che è loop-free, cioè non presenta cicli. Tale costruzione si ottiene forzando l'automata ad accettare stringhe soltanto alla fine di una sequenza di mosse spontanee.
Infine, il problema più grande di un automata è quello di poter accettare un linguaggio, ma non il suo complementare. Le PA sono più potenti delle FA, sia come accettatori PDA che come translatori PDT, poiché le FA sono un caso speciale di PDA con la differenza che quest'ultime non hanno limite di contabilità.

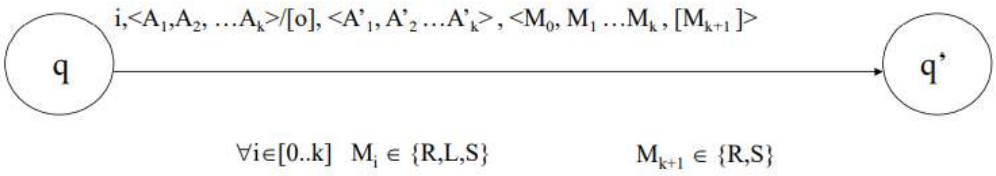
8. La macchina di Touring TM

La macchina di Touring è il modello storico di un computer, semplice e concettualmente fondamentale. In un primo momento consideriamo però le K-tape TM, una versione delle TM un po' più complessa.



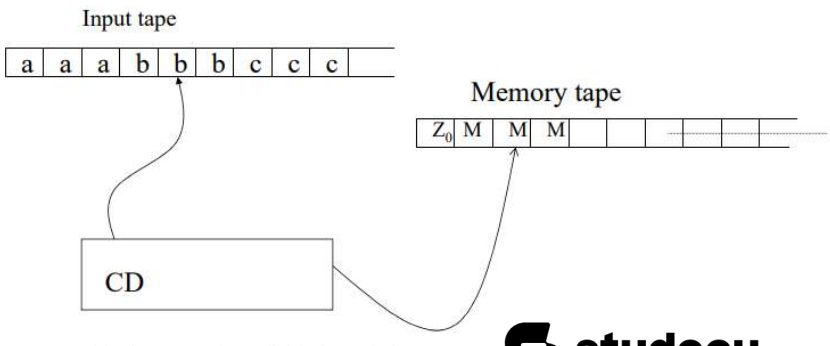
Lo stato e l'alfabeto di questo modello sono uguali a quelli già visti (input, output, dispositivo di controllo, memoria e alfabeto). Per motivi storici e per alcune tecniche matematiche, i nastri di questa automata sono infiniti e sequenziali [0,1,2, …] piuttosto che stringhe di lunghezza finita. La cella può però assumere un valore speciale vuoto (blank, " " o "barred b" o "_") e assumiamo a priori che ogni pacco contiene soltanto un numero finito di celle non-vuote.

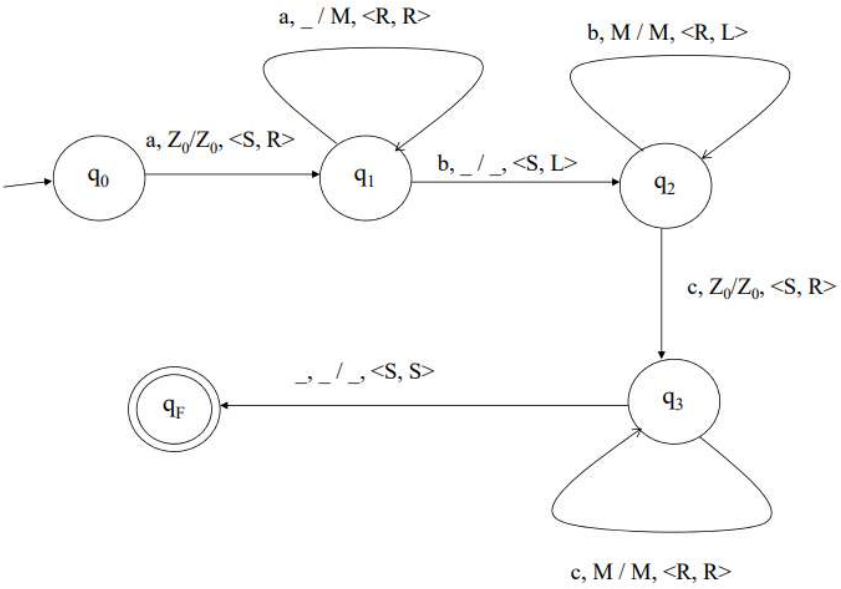
La macchina di Touring compie ***due "movimenti o spostamenti".*** Il primo è in **lettura,** viene letto un simbolo alla volta dal nastro di input, k simboli sui k nastri di memoria (uno per ogni nastro) e lo stato del dispositivo di controllo. Il secondo movimento è l'**azione**. Lo stato cambia da q--->q', scrive un simbolo al posto di quello letto su ciascuno dei k nastri di memoria Ai--->Ai' con 1<= i <= k, scrive un simbolo sul nastro di output. In totale sposta k+2 volte la testina di lettura del nastro. La testina che legge la memoria e scanerizza l'input può spostarsi di una posizione a destra o a sinistra o stare ferma, sull'output invece può spostarsi solo a destra o stare ferma.
Notazione grafica:



La **configurazione iniziale** prevede Z0 seguito da tante celle vuote sui nastri di memoria. Il nastro di output è tutto vuoto. La testina è posizionata nella posizione 0 su tutti i nastri. Lo stato iniziale del dispositivo di controllo è q0. La stringa di input inizia dalla cella zero sul nastro in input e tutte le altre celle sono vuote.
La **configurazione finale prevede** che, premesso che gli stati finali siano un sottoinsieme minore o uguale degli stati presenti nella TM, che l'automata si stoppi quando <δ,[η]> (q, …) = ⊥, ∀ q ∈ F (⊥ è lo stato "indefinito") e che la stringa sia accettata se e solo se dopo un numero finito di spostamenti lo stato q in cui si ferma è uno stato finale. Come conseguenza la stringa non viene accettata se l'automa si stoppa in uno stato che non è finale o se non si stoppa mai (non è loop-free).

- A TM accepting $\{a^n b^n c^n \mid n > 0\}$



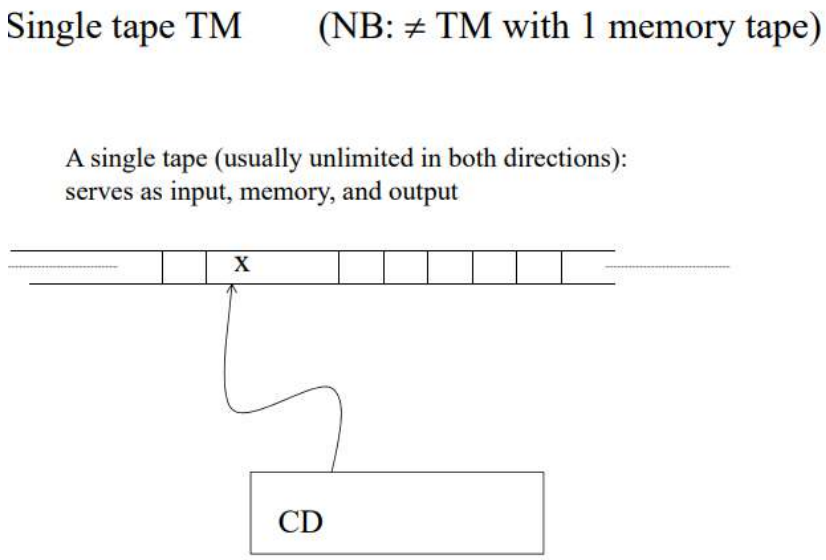


9. Proprietà della macchina di Touring TM

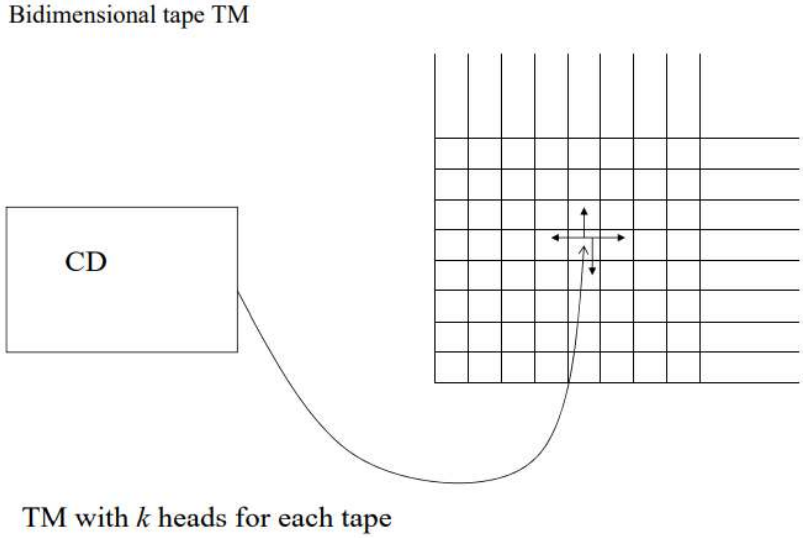
Una TM è chiusa rispetto alle relazioni di intersezione, unione, concatenazione, *, …, ma non per il complemento.

La TM ha delle versioni equivalenti, cioè accettano o traslano tutte nello stesso modo

Single tape TM



Bidimensional tape TM



Una **TM può simulare il funzionamento di una macchina di Von Neumann**, l'unica grande differenza è la modalità di accesso della memoria che è sequenziale piuttosto che random (diretta).

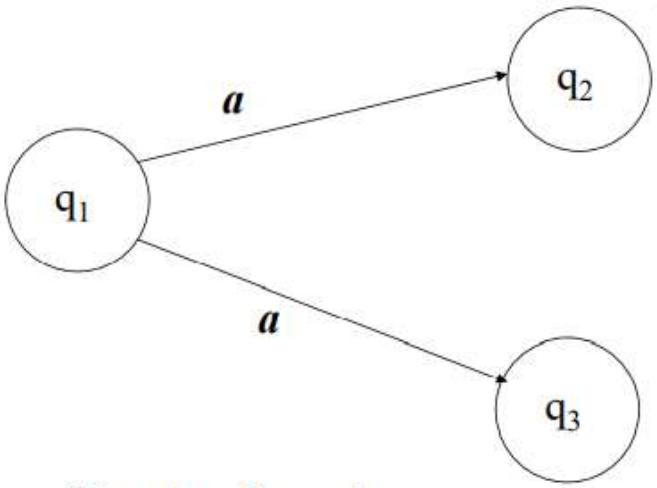
10. Modelli operazionali non-deterministici (ND)

I modelli operazionali non deterministici sono quelli che permettono di realizzare operazioni computazionali in **maniera parallela**. Pensiamo per esempio a un algoritmo di ricerca, quali per esempio la ricerca binaria. Come sappiamo la prima cosa che fa è verificare se l'elemento cercato è la radice, altrimenti verifica se la radice è maggiore o minore della radice, scegliendo quindi se spostarsi a destra o a sinistra. Quello che un modello non-deterministico si pone di fare è invece, analizzarli in contemporanea.

11. Modelli operazionali non-deterministici dei precedenti modelli

ND FA, tra tutti i vari possibili runs(vengono analizzati tutti i possibili path in maniera parallela), con lo stesso input, è sufficiente che uno di essi (**esiste** almeno uno che) accetta la stringa in input.

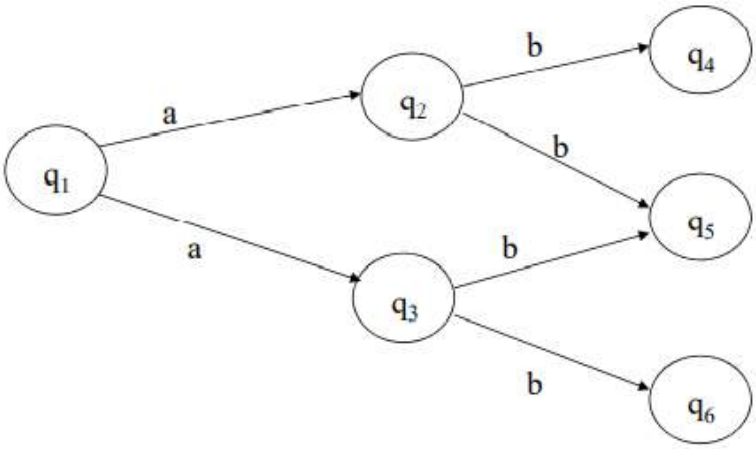
ND FA (we will soon see how handy it is)



Formally: $\delta(q_1,a) = \{q_2, q_3\}$

$\delta : Q \times I \rightarrow \mathcal{P}(Q)$

δ^* : formalization of a move sequence



$\delta(q_1,a) = \{q_2, q_3\}$, $\delta(q_2,b) = \{q_4, q_5\}$, $\delta(q_3,b) = \{q_6, q_5\}$

$\delta^*(q_1,ab) = \{q_4, q_5, q_6\}$

$$\delta^*(q,\varepsilon) = \{q\}$$

$$\delta^*(q,y.i) = \bigcup_{q' \in \delta^*(q,y)} \delta(q',i)$$

How does a ND FA accept?

$$x \in L \quad \leftrightarrow \quad \delta^*(q_0, x) \cap F \neq \varnothing$$

Among the various possible runs (with the same input) of the ND FA ***it suffices that one of them*** (that is, ***there exists one that***) succeeds and accepts the input string

Another, alternative interpretation of nondeterminism:

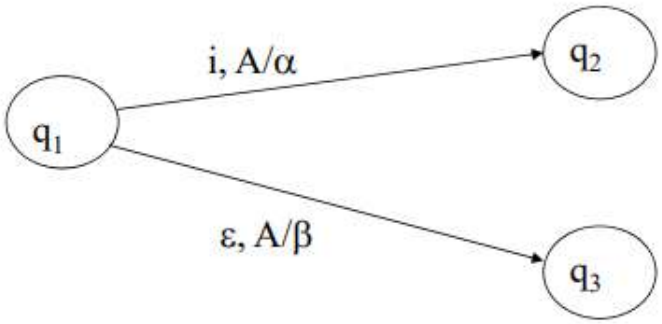
universal nondeterminism (the previous one is ***existential***):

all runs of the automaton accept

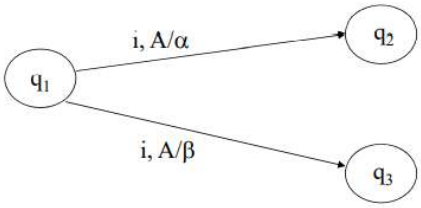
$$(\delta^*(q_0, x) \subseteq F)$$

Non-deterministic PDA (NPDA)

In fact PDA are "natural born" ND :



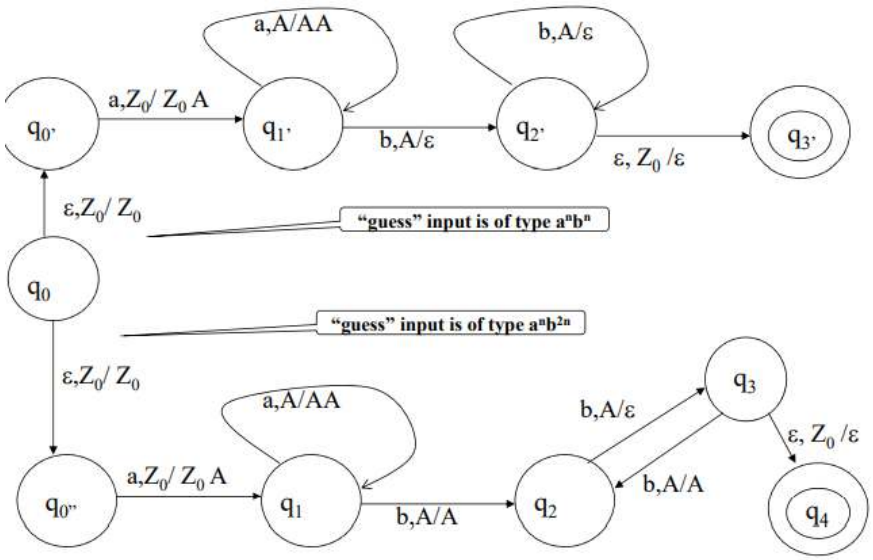
- We might as well remove the deterministic constraint and generalize:



$$\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^*)$$

Why index F? (finite subsets, we do not want infinite ones)
As usual, the NPDA accepts ***x*** if *there exists a sequence*
c_0 |-*- <q, ε, γ>, q ∈ F
in case of nondeterminism, the relation '|--' is not unique (i.e., ***functional***) any more

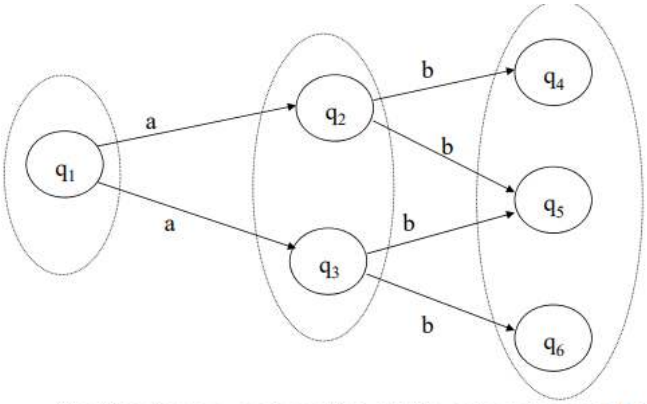
A "trivial" example: accepting $\{a^n b^n \mid n>0\} \cup \{a^n b^{2n} \mid n>0\}$



NPDA può accettare linguaggi che non sono accettati da una PDA, quindi sono più potenti, inoltre esse sono chiuse rispetto l'unione (le PDA non lo sono), resta però non chiusa rispetto l'intersezione. Una NPDA non è però chiusa rispetto al complemento.

Non-deterministic Finite-state Automata (NFA)

Le NFA non sono più potenti delle loro equivalenti deterministiche, ma per ogni NFA è possibile costruire la sua equivalente deterministica. Il suo utilizzo è proprio questo, spesso è più facile disegnare una NFA e ricavarne la FA, ma bisogna considerare che se per esempio una NFA ha 5 stati, l'equivalente potrebbe averne, nel caso peggiore, 2^5.



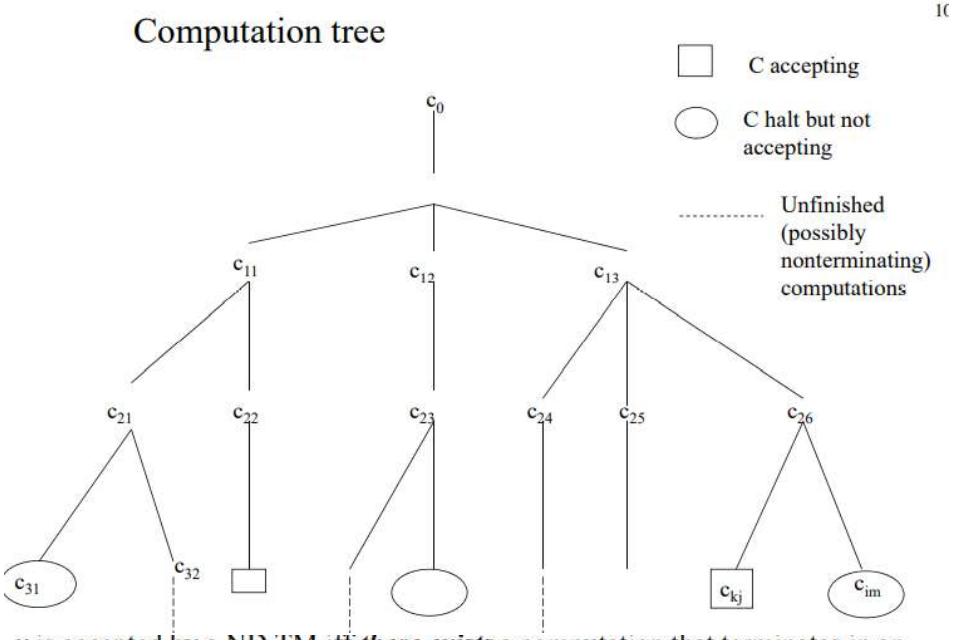
Starting from q_1 and reading ***ab*** the automaton reaches a state that belongs to the set $\{q_4,q_5,q_6\}$

Let us call again "state" the set of possible states in which the NFA can be during a run.

Non-deterministic TM

Neanche in questo caso non è più potente di una normale TM

$$\delta([\{S,R\}\times Q\times]^{k+1}\{S,L,R\}\times\Gamma^k\times Q)\leftarrow\Gamma^k\times\Gamma\times Q:<[\Gamma],\delta>$$



x is accepted by a ND TM iff ***there exists*** a computation that terminates in an accepting state

Can a deterministic TM establish whether a "sister" ND TM accepts ***x***, that is, accept ***x*** if and only if the ND TM accepts?

This amounts to "visit" the computation tree of the NDTM to establish whether it contains a path that finishes in an accepting state

This is a (***almost***) trivial, well known problem of tree visit, for which there are classical algorithms

The problem is therefore reduced to implementing an algorithm for visiting trees through TM's: a boring, but certainly feasible exercise … but beware the above "almost" ...

Everything is easy if the computation tree is finite

But it could be that some paths of the tree are infinite (they describe nonterminating –hence non-accepting– computations)

In this case a depth-first visit algorithm (for instance leftmost preorder) might "get stuck in an infinite path" without ever discovering that another branch is finite and leads to acceptance.

The problem can however be easily overcome by adopting a breadth-first visit algorithm (it uses a queue data structure rather than a stack to manage the nodes still to be visited).

Hence nondeterminism does ***not*** increase the power of the TM

Per concludere possiamo dire che le non-deterministic sono astrazioni che descrivono problemi di ricerca e algoritmi o situazioni dove non ci sono elementi di scelta e quindi computazione parallela. Generalmente non migliorano le deterministiche, ma possono dare una descrizione più compatta e descrivere fenomeni non deterministici

12. Grammatica

Come abbiamo visto, gli automata sono modelli che riconoscono/accettano, traslano e operano su linguaggi, ricevono una stringa in input e procedono in vari modi.

Un ***generative model*** è un modello grammatico o sintattico che generare stringhe di un linguaggio. La **grammatica o sintassi** (alfabeto, vocabolario, grammatica e sintassi sono sinonimi) definisce le regole per creare frasi di un linguaggio (stringhe), tramite un processo chiamato ***"riscrittura"*** (rewriting). Per esempio "una frase è formata da un soggetto seguito da un predicato", "un soggetto può essere un nome o un pronome, …", "un predicato può essere un verbo seguito da un complemento …".

Definizione formale di grammatica:
G = <V_N, V_T, P, S>

Dove V_N è il ***nonterminal*** alfabeto o vocabolario, V_T è il ***terminal*** alfabeto o vocabolario, V = $V_N \cup V_T$, S $\in V_T$ ed è un elemento particolare di V_N chiamato assioma o simbolo iniziale, P $\subseteq V_N^+ \; x \; V^*$ insieme delle regole di rewriting anche chiamato produzione. P = { < α, β > | $\alpha \in V_N^+ \wedge \beta \in V^*$} (in generale < α, β > va scritto come α --> β per enfatizzare l'operazione di rewriting).

ESEMPIO e **RELAZIONE DI DERIVAZIONE** " $\Rightarrow$ "

Example

$V_N = \{S, A, B, C, D\}$

$V_T = \{a,b,c\}$

S

$P = \{S \rightarrow AB, BA \rightarrow cCD, CBS \rightarrow ab, A \rightarrow \varepsilon\}$

Relation of Immediate Derivation "$\Rightarrow$"

$\alpha \Rightarrow \beta, \alpha \in V^+, \; \beta \in V^*$

if and only if

$\alpha = \alpha_1\alpha_2\alpha_3 \quad , \quad \beta = \alpha_1\beta_2\alpha_3 \quad \wedge \quad \alpha_2 \rightarrow \beta_2 \in P$

α_2 is rewritten as β_2 in the context of α_1 and α_3

With reference to the previous grammar: applying rule **BA → cCD**

aaBAS $\Rightarrow$ aacCDS

As usual, define the reflexive and transitive closure of $\Rightarrow$

$\overset{*}{\Rightarrow}$

it means: "zero or more rewriting steps"

Il **linguaggio** generato da una grammatica consiste in tutte le stringhe, che contengono solo simboli terminali (terminal), che possono essere derivati (in un numero qualsiasi di *steps*) da S

$$L(G) = \{x \mid x \in V_T^* \wedge S \overset{*}{\Rightarrow} x\}$$

ESEMPIO

A first example

$G_1 = < \{S, A, B\}, \{a, b, 0\}, P, S >$

$P = \{S \rightarrow aA, A \rightarrow aS, S \rightarrow bB, B \rightarrow bS, S \rightarrow 0\}$

Some derivations

$S \Rightarrow 0$

$S \Rightarrow aA \Rightarrow aaS \Rightarrow aa0$

$S \Rightarrow bB \Rightarrow bbS \Rightarrow bb0$

$S \Rightarrow aA \Rightarrow aaS \Rightarrow aabB \Rightarrow aabbS \Rightarrow aabb0$

Through an easy generalization :

$L(G_1) = \{aa, bb\}^*.0$

Second example

$G_2 =< \{S\}, \{a,b\}, P, S >$

$P = \{S \to aSb \mid ab\}$ (abbreviation for $S \to aSb$, $S \to ab$)

Some derivations

$S \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aabb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

Through an easy generalization :

$L(G_2) = \{a^n b^n \mid n \geq 1\}$

By substituting production $S \to ab$ with $S \to \varepsilon$ we obtain

$L(G_2) = \{a^n b^n \mid n \geq 0\}$

Third example: G_3

$\{S \to aACD, A \to aAC, A \to \varepsilon, B \to b, CD \to BDc, CB \to BC, D \to \varepsilon\}$

$S \Rightarrow aACD \Rightarrow a\mathbf{CD} \Rightarrow a\mathbf{B}\mathbf{D}c \overset{*}{\Rightarrow} abc$

$S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaCC\mathbf{D} \Rightarrow aaCC\downarrow$

$S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaC\mathbf{CD} \Rightarrow aa\mathbf{CB}Dc \Rightarrow$

$aa\mathbf{B}CDc \Rightarrow aab\mathbf{CD}c \Rightarrow aab\mathbf{B}Dcc \Rightarrow aabb\mathbf{D}cc \Rightarrow aabbcc$

$S \overset{*}{\Rightarrow} aaaACCCD \Rightarrow aaaCC\mathbf{CD} \Rightarrow aaaCC\mathbf{B}Dc \Rightarrow aaaCCb\mathbf{D}c \Rightarrow aaaCCbc\downarrow$

...

$S \to aACD$, $A \to aAC$ and $A \to \varepsilon$ generate as many a's as C's, and a final D

any $x \in L$ includes only terminal symbols, hence nonterminal symbols must disappear

C disappears only when it "hits" the D and then it generates a 'B' and a 'c'

C's and B's must switch to permit all the C's to reach the D

Hence $C^n D \Rightarrow^* b^n c^n$

Hence $L=\{a^n b^n c^n \mid n > 0\}$

L'uso pratico di una grammatica è quindi quello di definire la sintassi di un linguaggio di programmazione, in particolare genera linguaggio che possono essere riconosciute da un automata. Un chiaro esempio è il linguaggio di compilazione, in questo caso, la grammatica definisce il linguaggio, l'automata lo accetta e lo transla (traduce).

13. Classi di Grammatica

Context-free grammars:

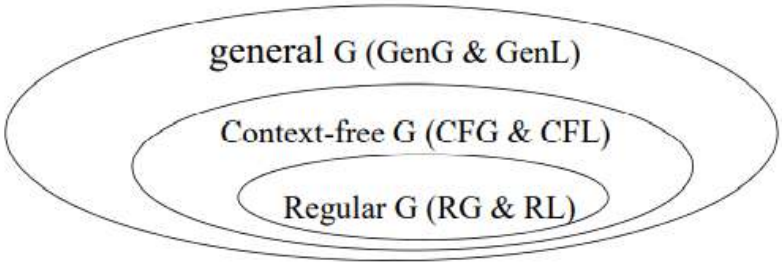
- $\forall$ regola (α --> β) $\in$ P, $|\alpha| = 1$, cioè α è un elemento A di V_N
- Context-free poiché la riscrittura (rewriting) di α (cioè A $\in V_N$) non dipende dal contesto, le parti di stringa che lo "circondano" non appaiono nella parte sinistra della regola
- Nell'esempio precedente G1 e G2 sono context-free, G3 non lo è

Regular Grammars:

- $\forall$ regola (α --> β) $\in$ P, $|\alpha| = 1$, $\beta \in ((V_T \,.\, V_N) \cup V_T \cup \{\varepsilon\})$
- Regular G are also Context-free ma non il contrario
- Nell'esempio precedente G1 è regular, G2 non lo è

Relazione di inclusione tra le varie classi

Inclusion relations among grammars and corresponding languages



Relazione tra la grammatica e l'automata

Definiamo l'equivalenza tra RG e FA (cioè le FA che accettano il linguaggio generato dalle RG)

Da FA a RG

Data una FA, A = (Q, I, q0, F), definiamo V_N = Q, V_T = I, S = <q0> e per ogni δ(q, i) = q', <q> --> i<q'> e se q' $\in$ F, aggiungiamo che <q> --> i.

È facile intuire che $\delta^*(q, x)$ = q' se e solo se <q> $\Rightarrow^*$ x <q'> e quindi, se q' F, <q> x*

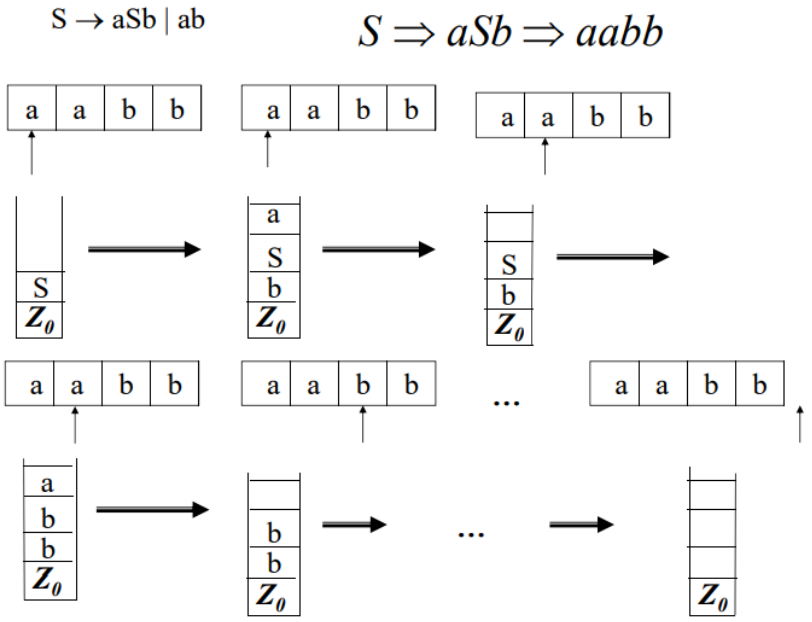
Viceversa da RG a FA:

Data una RG, definiamo **Q =** $V_N \cup$ {qF}, I = V_T, <q0> = S, qF $\in$F e

- Per ogni A --> bC definiamo δ(A,b) = C
- Per ogni A --> b definiamo δ(A,b) = qF
- Per ogni A --> ε, A $\in$ F

La FA ottenuta è non-deterministic

Una CFG equivale a una PDA (non deterministic)



genG equivalente a una TM

Data una grammatica G costruiamo una ND TM, M, che accetta L(G):

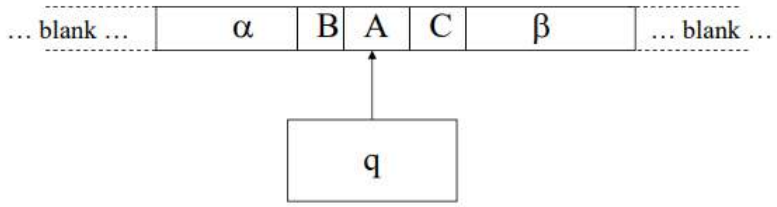
- Mettiamo la stringa x sul nastro in input
- M ha solo un nastro di memoria, e prova, in tutti i modi possibili, a derivare x su di esso e accetterà x se e solo se troverà una derivazione
- Il nastro di memoria è inizializzato con S (o meglio con Z_0S)
- Il nastro di memoria (che, generalmente conterrà la stringa α, $\alpha \in V^*$), viene scansionato cercando la parte sinistra della produzione P
- Quando uno viene trovato, viene sostituito con la corrispondente parte destra da M che ragiona in maniera non deterministica in questo modo:

$\alpha \Rightarrow \beta \leftrightarrow$ c = <w, qs, $Z_0\alpha$> |-*- <x, qs, $Z_0\beta$>
Se il nastro contiene una stringa y $\in V_T^*$, questa viene comparata con x e se coincidono, x viene accettata.

Vice versa Dato M (nastro singolo) definiamo una grammatica G che genera L(M):

- G genera tutte le stringhe del tipo x$X, x $\in$ VT^* e X diventa una "copia di x" composta da simboli nonterminal
- G simula le successive configurazioni di M usando la stringa sulla parte destra di $
- G è definita in modo che la sua derivazione x$X $\Rightarrow^*$ x, con x$\in$$VT^*$, se e solo se x è accettata da M
- L'idea è quella di simulare ogni movimento (spostamento della testina di lettura) su M immediatamente per ogni derivazione di G

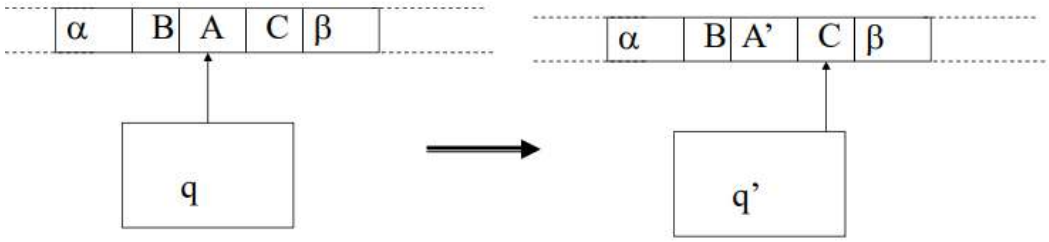
– We represent the configuration



through the string (special cases are left as an exercise): $αBqACβ
to start, G has therefore a set of derivations of the kind x$X⇒x$$q_o$X (where q_oX encodes the initial configuration of M)
for each value of the transition function δ of M, a rule of G is defined :

– δ(q,A) = <q', A', R> G includes the production qA → A'q'
– δ(q,A) = <q', A', S> G includes the production qA → q' A'
– δ(q,A) = <q', A', L> G includes the productions BqA → q' BA'
∀ B in the alphabet of M (recall that M is single tape, hence it has a unique alphabet for input, memory, and output)

– This way, for instance:



– If and only if: x$αBqACβ ⇒ x$αBA'q'Cβ,
– etc. …
– We finally add productions allowing G to derive from x$αB$q_F$ACβ a unique x if –and only if– M reaches an accepting configuration (αBq_FACβ), by deleting whatever is to the right of $, and also $

14. Uso della logica matematica come formalismo descrittivo

La logica è un formalismo universale e può essere applicata a una grandissima varietà di contesti, soprattutto con la theoretical computer science. Possiamo utilizzare la logica per definire Linguaggi e proprietà di alcuni sistemi. Le caratteristiche fondamentali della logica dipendono da:

- Dominio (Universo U): insieme di valori possibili (constanti, variabili, espressioni)
- Alfabeto scelto per predicati e simboli delle funzioni adottate
- Possibilità per variabili (quantificate) di denotare solo oggetti del primo ordine (elementi dell'universo) o oggetti del secondo ordine (insiemi di tali elementi e relazioni tra di loro)

15. Logica matematica per la definizione di un linguaggio

Dato un linguaggio

$$\mathrm{L} = \{a^n b^n \mid n \geq 1\}$$

specificato dalla formula di primo ordine

$$\forall \mathrm{x}(\mathrm{x} \in L \leftrightarrow \exists\, \mathrm{n}(\mathrm{n} \geq 1 \wedge \mathrm{x} = a^n \cdot b^n))$$

Dove ricordiamo che:

- La variabile x denota la stringa
- La variabile n un numero naturale
- La costante L un linguaggio (insieme di stringhe, con ε stringa vuota)
- Predicato '$\in$' appartenenza e '=' predicato di uguaglianza
- Le operazioni '$\cdot$' (concatenazione tra stringhe) e x^n (potenza su stringhe) definita dalla formula:

$$\forall n\ ((n = 0 \rightarrow x^n = \mathcal{E}) \wedge (n > 0 \rightarrow x^n = x^{n-1} \cdot x))$$

Possiamo definire con $L_1 = a^* b^*$ senza usare l'operazione potenza x^n (cioè che,

$x \in L_1 \leftrightarrow \exists m \exists n\ (x = a^m b^n \wedge n \geq 0 \wedge m \geq 0\ ))$

$$x \in L_1 \leftrightarrow ((x = \mathcal{E}) \vee \exists y\ (\ y \in L_1 \wedge (x = ay \vee x = yb)))$$

Allo stesso modo possiamo definire $L_2 = b^* c^*$

$L_3 = a^* b^* c^* (= L_1 \cdot L_2)$ che quindi può essere definita senza l'uso di x^n

$$x\ \in L_3 \leftrightarrow (x \in L_1) \vee (x \in L_2) \vee \exists \mathrm{y}(y \in L_3 \wedge (x = ay \vee x = yc))$$

16. Definizioni generali MFO

- Monadic First Order Logic (mondaic = predicati con un solo argomento)
- Binary order predicate '<'
- Boolean connectives ($\neg, \vee, \wedge, \rightarrow, \leftrightarrow$, etc ….) e quantificatori ($\exists, \forall$)
- L'alfabeto I considera tutte le stringhe con $w\ I^*,$ con $|w| = n, cio\grave{e}, w = w_0 w_1 \ldots w_{n-1}$
- L'universo $U = \{0, \ldots, n-1\}$ di posizione di stringhe (se $x = \mathcal{E}$ allora $U = \emptyset$)
- Per ogni simbolo $a \in$ I monodic predicate a(x), vero per la stringa w se e solo se $w_x = a$, sioè, se e solo se, il simbolo A si trova nella stringa w alla posizione x
- Il binary predicate '<' utilizzato tra posizioni di stringa ha il significato usuale.

Definizioni derivanti

- $x \geq y$ defined as $\neg(x < y)$
- $x \leq y$ as $y \geq x$
- $x = y$ as $x \leq y \wedge y \leq x$
- $x \neq y$ as $\neg(x = y)$
- $x > y$ as $y < x$
- immediate successor: $succ(x, y)$ ($S(x, y)$ for short) as $x < y \wedge \neg\exists z\ (x < z \wedge z < y)$

- constant $\mathbf{0}$: $x = 0$ as $\forall y \neg (y < x)$
- all natural integer constants $\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots$: defined as successors of $\mathbf{0}, \mathbf{1}, \mathbf{2}$, etc.
- $y = x + 1$ defined as $succ(x, y)$
- $y = x+k$, for every $k > 1$, as $\exists z_1 \ldots \exists z_{k-1}\ (z_1 = x+1 \wedge \ldots \wedge y = z_{k-1}+1)$
- $y = x - 1$ as $succ(y, x)$
- $y = x - k$, for every $k > 1$, as $x = y + k$
- first position in the string : $first(x)$ as $\neg\exists y\ (y < x)$ (equiv. to $x = 0$)
- last position in the string : $last(x)$ as $\neg\exists y\ (y > x)$

- **NB**: terms like $x + y$, with x and y both variables are ***not admitted***

17. Interpretare una formula logica MFO scritta come stringa

Ogni stringa $w \in I^*$ corrisponde a una struttura di interpretazione con un alfabeto I e un universo $U = \{0, \ldots, |w| - 1\}$.

ex: for string $w = acbaa$, with $n = |w| = 5$, we have

- alphabet $I = \{a, b, c\}$
- f.o. variables denote positions in w; they are interpreted over the Universe $U = [0 \ldots n\text{-}1] = [0 \ldots 4]$
- for every alphabet element $i \in I$, predicate $i(x)$ denotes the set of positions x at which $w_x = i$, (NB: $\forall i \neq j\ \forall x\ \neg(\ i(x) \wedge j(x)\ )$)
  - for $w = acbaa$, $a(0)$, $c(1)$, $b(2)$, $a(3)$, $a(4)$ are ***true***;
    $b(0)$, $c(0)$, $a(1)$, $b(1)$, $a(2)$, $c(2)$, $b(3)$, $c(3)$, $b(4)$, $c(4)$ are ***false***
  - NB: in the atomic formula $a(x)$, a is a predicate ***constant***: it is not quantified
- less-than relation for $w = acbaa$ is a set of pairs of positions:
  '<' = {(0,1), (0,2), ... (1,2), (1,3), … (3,4)}

Una frase ***(sentence)*** è una formula con tutte le variabili quantificate

Ogni sentence φ definisce il linguaggio $L(\varphi)$ che include esattamente tutte e soltanto le stringhe che soddisfano φ:

$$L(\varphi) = \{\, w \mid w \models \varphi \}$$

Examples of MFO sentences defining strings and languages [13]:

φ: $\exists x(\ x = 0 \wedge a(x)\ )$ $L(\varphi) = aI^*$ strings starting with an a

strings where every a is followed by a b :

φ: $\forall x\ (\ a(x) \rightarrow \exists y\ (\ S(x, y) \wedge b(y)\ )\ )$

$aaba \not\models \varphi$, $babbab \models \varphi$, $bba \not\models \varphi$

(non-empty) strings ending with an 'a' : $\exists x(\ last(x) \wedge a(x)\ )$

strings (of length ≥ 3) where second symbol before the last is a

$\exists x(\ \exists y\ (\ y = x+2 \wedge last(y)\ )\ \wedge\ a(x)\ )$

Every ***singletone*** (one-element) language easily defined;
example $L_{abc} = \{\ abc\ \}$

$\exists x \exists y \exists z\ (x=0 \wedge S(x,y) \wedge S(y,z) \wedge last(z) \wedge a(x) \wedge b(y) \wedge c(z)\ )$

18. Le stringhe vuote Ɛ richiedono alcune accortezze

Per le stringhe Ɛ, il set di posizioni è vuoto. Convenzionalmente, quando $U = \emptyset$, $\exists x\varphi$ è falso e $\forall x\varphi$ vero per ogni φ. Una sentence per {Ɛ} deve essere vera per Ɛ ma falsa per ogni stringa w diversa da Ɛ.

$$\neg\exists x\,(\,a(x) \vee \neg a(x)\,)\ \text{(i.e., } \neg\exists x\,(true)\text{, or } \neg\exists x \text{ or } U{=}\emptyset)$$

$$\text{or, equivalently, } \forall x\,(\,a(x) \wedge \neg a(x)\,)\quad (\text{i.e., } \forall x\,(false))$$

19. Proprietà di una MFO

Le MFO sono chiuse rispetto alle operazioni set-theoretic (unione, intersezione, complemento, ecc..).
(Banalmente si ricava dalle operazioni di $\vee$, $\wedge$, $\neg$)
Ogni linguaggio finito o "co-finito" (il suo complementare è finito) è esprimibile tramite una MFO. Inoltre, ogni linguaggio esprimibile tramite MFO è finito o co-finito.
Per esempio, il linguaggio $L_e = (aa)^*$ non è esprimibile tramite MFO poiché non è né finito ne co-finito
Una MFO è rigorosamente (stricly) meno potente di una FA. Un linguaggio espresso tramite MFO è facilmente tradotto in FA. L'esempio precedente, $L_e = (aa)^*$ è riconosciuto da una semplice FA, ma come abbiamo detto non è esprimibile tramite MFO.

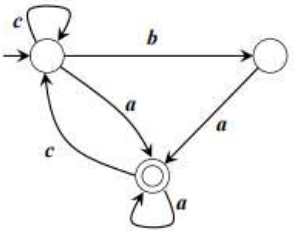
I linguaggi definiti da una MFO non sono chiusi rispetto l'operazione Kleen star '*'.
La formula MFO

$$\exists x \exists y\,(\,x{=}0 \wedge y{=}1 \wedge a(x) \wedge a(y) \wedge last(y))$$

definisce il linguaggio $L_{e2} = \{aa\}$, e quindi abbiamo $L_e = L_{e2}^*$

Diciamo che una MFO definisce la famiglia dei star-free linguaggi, che possono essere ottenuti partendo da un qualsiasi linguaggio finito che significa, da un finito numero di unioni, intersezioni, complementi e operazioni di concatenazione (ma no Kleen star '*').
Esempio della relazione tra FA e MFO:

$$\varphi = \left(\begin{array}{c} \textit{no } 'a' \textit{ is followed by a } 'b' \\ \wedge \\ \textit{every } 'b' \textit{ is followed by an } 'a' \\ \wedge \\ \textit{the string ends with an } 'a' \end{array}\right)$$



$$\varphi = \left(\begin{array}{c} \neg\exists x \exists y (S(x,y) \wedge a(x) \wedge b(y)) \\ \wedge \\ \forall x\,\big(b(x) \rightarrow \exists y (S(x,y) \wedge a(y))\big) \\ \wedge \\ \exists x\,(last(x) \wedge a(x)) \end{array}\right)$$

- Are the two models equivalent?
- Is it always possible to
  - find an FA equiv. to a MFO?

20. Monadic di secondo ordine (MSO) Logic

Il potere espressivo di una MFO può essere migliorato introducendo delle variabili chiamate "monodic predicates", cioè, un insieme (set) di numeri.
Se un predicato X rappresenta un insieme, allora X(3) è lo stesso di $3 \in X$.
Esempio:
Il linguaggio $L_e = (aa)^*$ è definito dalla seguente formula MSO, dove il predicato E identifica le posizioni pari (indici dispari).

$$\begin{aligned} \exists E\,(\forall x\,(\quad & a(x) \wedge \\ & (\,x = 0 \rightarrow \neg E(x)\,) \wedge \\ & \forall y\,(\,y = x{+}1 \rightarrow\ (\neg E(x) \leftrightarrow E(y))\,) \wedge \\ & \forall y\,(\,last(y) \rightarrow E(y)\,) \qquad\qquad) \end{aligned}$$

Secondo esempio:
Parole con alfabeto I = {a,b}, dove ogni due occorrenze di b, non ci sono altre b tra di esse e sono separate da un numero dispari di a.

$$\varphi = \forall x \forall y \left(\begin{array}{c} b(x) \wedge x < y \wedge b(y) \wedge \forall z\big(x < z \wedge z < y \rightarrow \neg b(z)\big) \\ \rightarrow \\ \exists X \Big(X(x) \wedge X(y) \wedge \forall u \forall v \ \big(S(u,v) \rightarrow \big(X(u) \leftrightarrow \neg X(v)\big)\big)\Big) \end{array} \right)$$

second order quantification : X is a predicate that alternates true and false values

$a\ b\ a\ a\ a\ b\ a\ b \models \varphi$

$\neg X\ X\ \neg X\ X\ \neg X\ X\ \neg X\ X$

OK OK

$a\ b\ a\ a\ b\ a\ b \not\models \varphi$

$\neg X\ X\ ????\ X\ \neg X\ X$

KO OK

Abbiamo ottenuto un importante risultato, le famiglie di linguaggio definite da sentences MSO sono uguali a RL (regular Languages)

21. Teorema di Buchi's

Un linguaggio (di parole con lunghezza finita) è riconoscibile da un automata finita se e solo se è definibile da una sentence MSO. Vale il vice-versa.

Dimostrazione da sinistra a destra:

automaton $A = (Q, I, q_0, \delta, F)$, $Q = \{q_0, \dots q_k\}$, reads a string w

use $|Q|$ distinct predicates X_i = {positions of w where A reaches state q_i}

= { x | A is in state q_i when it reads w_x }

formula φ equivalent to A :

$$\varphi = \exists X_0 \dots \exists X_k \left(\begin{array}{c} \wedge_{i \neq j} \forall x \neg\big(X_i(x) \wedge X_j(x)\big) \\ \wedge \forall x \ (first(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left(S(x,y) \rightarrow \bigvee_{q_j \in \delta(q_i,a)} \big(X_i(x) \wedge a(x) \wedge X_j(y)\big) \right) \\ \wedge \forall x \left(last(x) \rightarrow \bigvee_{\exists q_j \in F:\, q_j \in \delta(q_i,a)} (X_i(x) \wedge a(x)) \right) \end{array} \right)$$

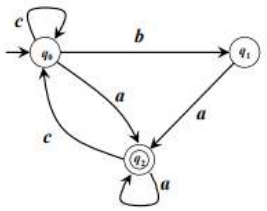
state positions pairwise disjointed

A in the **initial state** right before reading the first symbol

final state: reached after reading the last symbol of the string

transitions: reading a at position x, A goes from state q_i to state q_j

Esempio con k=2



$$\exists X_0 \exists X_1 \exists X_2 \left(\begin{array}{c} \forall x\big(\neg(X_0(x) \wedge X_1(x)) \wedge \neg(X_0(x) \wedge X_2(x)) \wedge \neg(X_1(x) \wedge X_2(x))\big) \\ \wedge \forall x \ (first(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left(S(x,y) \rightarrow \left(\begin{array}{l} X_0(x) \wedge c(x) \wedge X_0(y) \ \vee \ X_0(x) \wedge b(x) \wedge X_1(y) \\ \vee\ X_0(x) \wedge a(x) \wedge X_2(y) \ \vee \ X_1(x) \wedge a(x) \wedge X_2(y) \\ \vee\ X_2(x) \wedge a(x) \wedge X_2(y) \ \vee \ X_2(x) \wedge c(x) \wedge X_0(y) \end{array} \right) \right) \\ \wedge \forall x\big(last(x) \rightarrow (X_0(x) \wedge a(x) \vee X_1(x) \wedge a(x) \vee X_2(x) \wedge a(x))\big) \end{array} \right)$$

Dimostrazione da destra a sinistra:

an illustrative example: (regular) language L of strings $w \in \{a, b\}^*$ that include 2 consecutive a's (i.e., $L = \{a, b\}^*\ aa\ \{a, b\}^*$)

- costruction of FA recognizing L in [Thomas96] and [Wehr07]: a (technically complex) proof by induction on the structure of φ

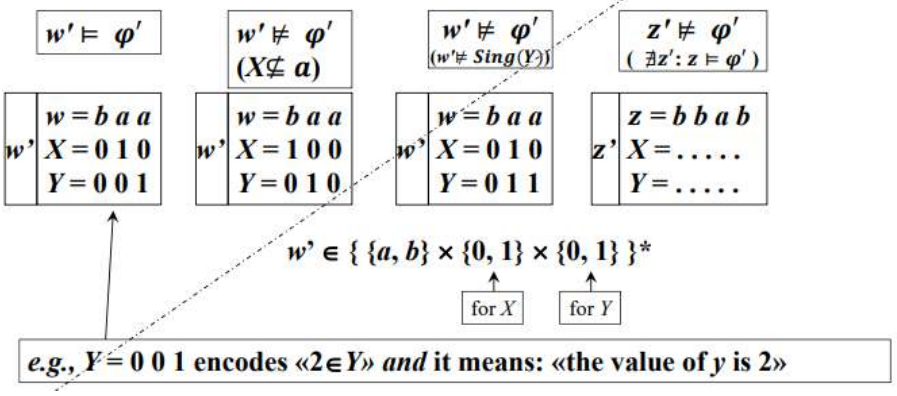
MSO formula: $\varphi = \exists x\, \exists y\, S(x, y) \wedge a(x) \wedge a(y)$ **procedure in 4 steps**

f.o. variables x, y turned into second order: $X, Y \subseteq U = [\ 0\ ..\ |w|\text{-}1\ ]$

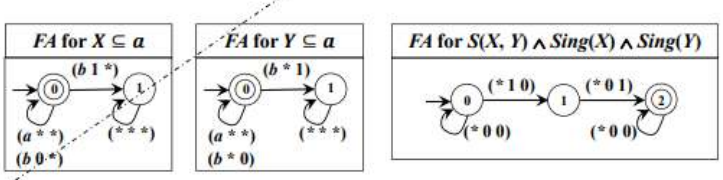
- X, Y represent variables: we introduce in the s.o. logic the ***Sing*** predicate (singleton, a derived predicate) to state that they are true for only one value
- s.o. logic includes (as a derived predicate) '$\subseteq$' with the usual meaning
- formula φ becomes (*NB*: in the formula below a denotes a predicate!)

$$\varphi' = \exists X\, \exists Y\, S(X, Y) \wedge Sing(X) \wedge Sing(Y) \wedge X \subseteq a \wedge Y \subseteq a$$

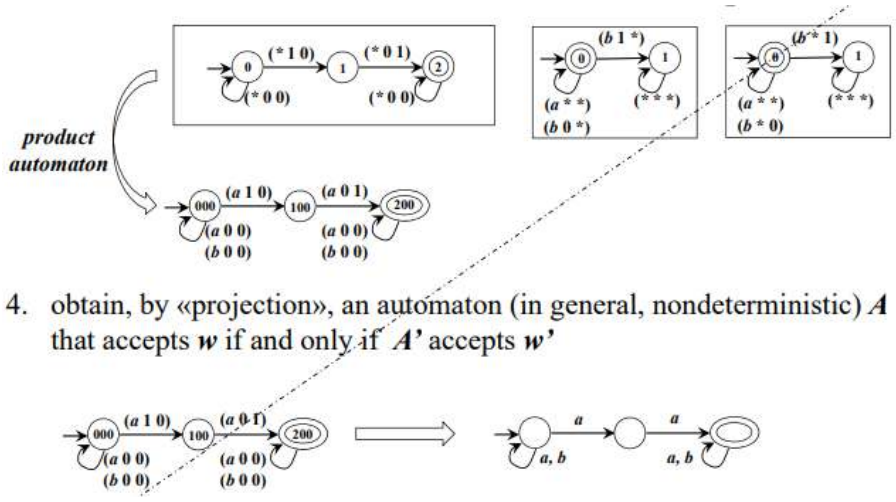
2. string $w \in \{a, b\}^*$ enriched with 2 components for X and Y, obtain an «enriched string» w' that includes elements for evaluating X and Y, hence it can be used to evaluate φ'



3. build automaton A'
   - with input alphabet $I' = \{a, b\} \times \{0, 1\} \times \{0, 1\}$ (for X, for Y)
   - A' accepts w' iff $w' \models \varphi'$
   - A' built from automata for subformulas
     $X \subseteq a$, $Y \subseteq a$, $S(X, Y) \wedge Sing(X) \wedge Sing(Y)$
     exploiting closure properties of FA under set-theoretic operations



NB: in the above FA's, '*' means «any value», e.g. «(b 1 *)» stands for «(b 1 0) or (b 1 1)»



4. obtain, by «projection», an automaton (in general, nondeterministic) ***A*** that accepts ***w*** if and only if ***A'*** accepts ***w'***

22. Uso della logica per definire le proprietà di un programma

L'alfabeto include sets, relazioni e operatori che sono alla base dei linguaggi di programmazione.
Esempio di un algoritmo di ricerca:
La variabile logica "found" è vera se e solo se esiste un elemento nell'array, di n elementi, uguale all'elemento x cercato:

$$found \leftrightarrow \exists i(1 \le i \le n \wedge a[i] = x)$$

Esempio di un algoritmo che inverte un array:
L'output b contiene gli stessi elementi in input a, ma in ordine inverso

$$\forall i\left(1 \le i \le n \ \rightarrow \ b[i] = a[n-i+1]\right)$$

In termini generali, è possibile anche definire un insieme di logiche

{Pre-condition}
Program – o parte di esso – P
{Post-condition}

Per esempio, per descrivere un algoritmo di ricerca di un elemento in un array ordinato:

$$\{\forall i(1 \le i < n \rightarrow a[i] \le a[i+1])\}$$
$$P$$
$$\{found \ \leftrightarrow \exists i(1 \le i \le n \wedge a[i] = x)\}$$

In poche parole, prima di cercare nell'array, dobbiamo ordinarlo.

Esempio di un algoritmo che ordina un array di n elementi senza ripetizioni:

$$\{\neg\exists i, j(1 \le i \le n \wedge 1 \le j \le n \wedge i \ne j \ \wedge a[i] = a[j])\}$$
$$SORT$$
$$\{\forall i(1 \le i < n \rightarrow a[i] \le a[i+1])\}$$

Questo ultimo algoritmo non è però specificato in modo adeguato in quanto potrebbe eliminare elementi dall'array, è quindi corretto implementarlo nel seguente modo:

$$\{\neg\exists i, j(1 \le i \le n \ \wedge \ 1 \le j \le n \ \wedge \ i \ne j \ \wedge \ a[i] = a[j]) \quad \wedge \quad \%\ no\ duplicates\ in\ a$$
$$\forall i(1 \le i \le n \rightarrow a[i] = b[i])\}$$
$$SORT$$
$$\{\forall i(1 \le i < n \ \rightarrow \ a[i] \le a[i+1]) \quad \wedge$$
$$\forall i(1 \le i \le n \ \rightarrow \ \exists j(1 \le j \le n \ \wedge \ a[i] = b[j])) \quad \wedge \ \%\ all\ a[i]\ are\ in\ some\ b[j]$$
$$\forall j(1 \le j \le n \ \rightarrow \ \exists i(1 \le i \le n \ \wedge \ b[j] = a[i]))\} \quad \%\ all\ b[j]\ are\ in\ some\ a[i]$$

23. Uso della logica matematica per definire le proprietà di sistemi (timing)

Valori distinti della variabile t, denotano punti diversi di tempo. I predicati con una variabile temporale come argomento denotano «fatti» che possono valere o meno in determinati istanti di tempo.
Esempio:

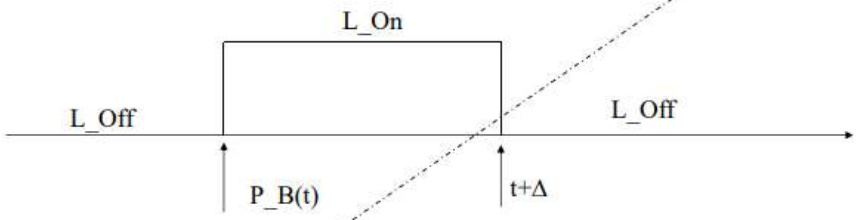
"If I push the button the light goes on within **Δ** time units (t.u.)":

- ***P_B(t)***: a predicate denoting Push Button at time ***t***
- ***L_On(t)***: a predicate denoting that the Light is On at time ***t***

$$\forall t(P\_B(t) \rightarrow \exists t_1((t \leq t_1 \leq t+\Delta) \wedge L\_On(t_1)))$$

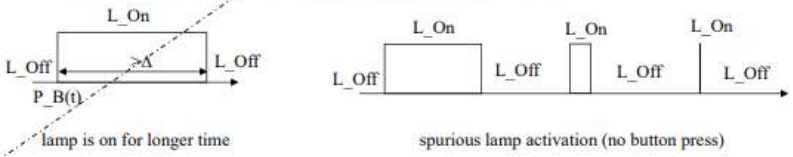
L'algoritmo sembrerebbe corretto ma in realtà le proprietà del sistema non sono realistiche, poiché non è deterministico e le lampade temporizzate di solito restano accese per un dato intervallo di tempo.

- If the button is pushed the light is on *for* Δ t.u. (similar requirements for an alarm or an electronic safe): here is a typical ***intended interpretation*** (assuming ∀t(L_On(t)↔¬L_Off(t)))



$$\forall t(P\_B(t) \rightarrow \forall t_1((t \leq t_1 \leq t+\Delta) \rightarrow L\_On(t_1)))$$

- But here are some ***unintended*** interpretations



Possiamo notare che la formula che abbiamo definito precedentemente da una condizione sufficiente affinché la lampada sia accesa, ma non necessaria. Dobbiamo trovare quindi una condizione necessaria e sufficiente come quella di seguito:

$$\forall t(L\_On(t) \rightarrow \exists d((0 < d < \Delta) \wedge P\_B(t-d)))$$

24. Theory of computation

Come abbiamo visto, tramite un linguaggio possiamo formalizzare ogni "computer science problem"

$x \in L?$ (Language recognition problem)
$y = \tau(x)?$ (Function computation problem)

In parole povere, si dice che:

- Se riesco a trovare una macchia che risolve un problema del tipo $y = \tau(x)$ e vorrei risolvere il problema $x \in L$ è sufficiente definire il predicato $\tau(x) = 1$ se $x \in L, \tau(x) = 0$ se $x \notin L$.
- Viceversa, se voglio calcolare una funzione $y = \tau(x)$ devo definire il linguaggio

$$L_\tau = \{x \$ y \mid y = \tau(x)\}$$

-

Assuming that I can recognize L_τ using some machine, then, for a fixed x, I could enumerate all possible strings y over the output alphabet and for each of them ask the machine if $x\$y \in L_\tau$: soon or late, ***if $\tau(x)$ is defined***, I will find the string for which the machine answers positively: this is a way to compute $y = \tau(x)$.
The procedure is "a bit long" but at the moment we are not concerned about the length of computations

Concerning the machine … in fact there exist many, besides the ones we know; and many more can be invented, so we might be able to get results of the kind

$\{a^n b^n | n > 0\}$ is accepted by a PDA and a TM but not by a FA.

However we noticed that it is not so easy to overcome the computing power the TM: adding tapes, heads, nondeterminism, … does not increase the power, (i.e., the class of accepted languages);
It is not so difficult to have the TM do what a normal computer does: It suffices to simulate the memory of one of the two by the other one

⟶ ***HENCE***

there is a ultimate generalization:

CHURCH THESIS (1930)

Secondo Church, non esiste un dispositivo computazionale più potente di una TM.
La domanda da porci adesso è la seguente:
Esistono problemi che non possono essere risolti da una TM?
Primo fatto di importanza:
Possiamo numerare algoritmicamente una TM e indicheremo con E questa numerazione.
Esempio:

Algorithmic Enumeration of $\{a, b\}^*$:

$\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots.\}$
– Usually called the "lexicographical ordering"

Esempio:
TM single tape (nastro singolo)
Alfabeto $|A| = 2$, $a = \{0, 1\}$

	0	1
q_0	⊥	⊥
q_1	⊥	⊥

TM_0

	0	1
q_0	⊥	⊥
q_1	⊥	$<q_0, 0, S>$

TM_1 ……

Quanti stati abbiamo?

$$\delta: Q \times A \to Q \times A \times \{R,L,S\} \cup \{\perp\}$$

Quante funzioni di tipo f:D-->R ci sono?

$|R|^{|D|}$ (for each $x \in D$ we have $|R|$ choices}
With $|Q| = 2$, $|A| = 2$, $(2 \cdot 2 \cdot 3+1)^{(2 \cdot 2)} = 13^4$ two-state TM's

Possiamo poi ordinare queste TM: $\{M_0, M_1, M_2, \ldots., M_{13^4-1}\}$
E ottenere un'enumerazione

$$E: \{TM\} \longleftrightarrow \mathcal{N}$$

E è algoritmico, cioè, possiamo scrivere un programma in C (cioè una TM) che dato n, produce le n-th TM e viceversa, date un insieme di n-th TM (tabelle) ci restituisce la posizione E(M) di M nell'enumerazione
E(M) è chiamata **Goedel number** di M, E **goedelization.**

Definiamo delle convenzioni

- Risolvere un problema = calcolare la funzione $f: N \to N$

- f_y = funzione calcolata dalla y-th TM
  $f_y(x) = \perp$ se e solo se M_y non si ferma quando viene dato x in input
- Funzione computable = esiste una TM che la calcola
- Funzione uncomputable = non esiste una TM che la possa calcolare
- Funzione total computable = definita per ogni valore del suo dominio
- Funzione partial computable = indefinita per alcuni valori del suo dominio

Esempio:

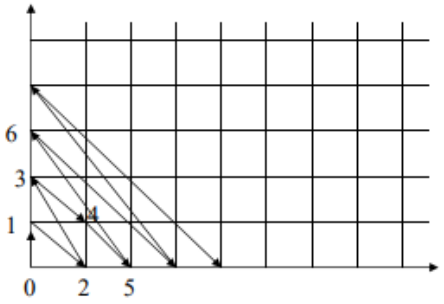
– $\forall x \in \mathbb{N}, f_x$ is computable by definition

computable	3	4

1) function f(x)=x+1 is total and computable
2) the everywhere undefined function $f(x)=\perp$ for all x is partial and computable
   function $f(x)$ = if (x is even) then x else $\perp$ is partial and computable
3) the (predicate) function $f(x)$=if $f_x(y)=2\cdot y$ then 1 else 0 is total, not computable
4) $f(x)$= if (x is even and $f_{x/2}$ is total) then 1 elsif (x is even and $f_{x/2}$ is partial) then 0 else $\perp$
   is partial and not computable (this will be an easy exercise …)

– 3) and 4) to be explained later… as well as many other nontrivial, interesting cases

Secondo fatto importante:
Esiste una Universal Turing Machine (UTM), cioè la TM che calcola le funzioni $g(y,x) = f_y(x)$. La UTM non fa però parte della famiglia $\{M_y\}$ poiché la funzione f_y è definita da una sola variabile mentre g da due.
È possibile però enumerare coppie di interi, cioè $N \times N \to N$



$$d(x,y) = \frac{(x+y)(x+y+1)}{2} + x$$

Quindi possiamo immaginare che $g^n = g\left(d^{-1}(n)\right) = g(y,x)$,
$dove\ g(y,x)$ è codificata come $g^n, con\ n = d(y,x),\ cioè < y,x > = d^{-1}(n)$.

Sketch of the operation of the UTM that computes g^ (NB from now on for simplicity we will simply write g instead of g^):

– Given n, the UTM computes $d^{-1}(n) = <y,x>$
– Then it builds the transition function of M_y (by computing $E^{-1}(y)$) and stores it on some portion of a tape:

	$	q	a	q'	a'	S	$	..	..							

- In another tape portion is stores an encoding of the configuration of M_y

	#	0	1		0			..	1	q	1		1		0	#

NB: The special symbols #, $ and other ones are coded as binary strings

At the end the UTM leaves on the tape only $f_y(x)$ if and only if M_y terminates its computation on x

Possiamo quindi immaginare una TM come un semplice modello di un computer che calcola sempre la stessa funzione tramite un algoritmo. Una UTM come un computer memory-stored, y è il programma, x l'input per il programma.

Per rispondere alla domanda, possiamo dire che la maggior parte delle funzioni (problemi) non può essere risolta algoritmicamente e che esistono molti più problemi che programmi.

$$\{\text{Solvable Problems}\} \subseteq \{\text{Definable Problems}\}$$

La domanda che allora ci poniamo è: quali problemi possono essere risolti?

25. Problemi incalcolabili

In generale un computer (che è un programma) non può segnalarci che il programma che abbiamo scritto non può essere risolto e che quindi non si troverà mai in una posizione terminal (entrerà in loop infinito), poiché questo è un problema non risolvibile algoritmicamente.

Dimostrazione

It employs a typical *diagonal technique* (adopted also in the Cantor theorem to show that $\aleph_0 < 2^{\aleph_0}$)

Let us ***assume*** (by contradiction) that the total function :

$$g(y,x) = 1 \text{ if } f_y(x) \neq \perp, \quad g(y,x) = 0 \text{ if } f_y(x) = \perp$$

is computable

Then also the partial function

$$h(x) = \begin{cases} 1 \text{ if } g(x,x) = 0 \text{ (i.e., if } f_x(x) = \perp), \\ \perp \text{ if } g(x,x) = 1 \text{ (i.e., if } f_x(x) \neq \perp) \end{cases}$$

is computable

NB: we went on the *diagonal* $y=x$, we changed the no answer ($g(x,x) = 0$) into a yes answer ($h(x)=1$), and we turned the yes ($g(x,x) = 1$) into a nontermination ($h(x)=\perp$), which can always be easily done

If h is computable then $h = f_{xh}$ for some xh.

Question: $h(xh) = 1$ or $h(xh) = \perp$?

Let us assume that $h(xh) = f_{xh}(xh) = 1$

Then $g(xh,xh) = 0$, that is, $f_{xh}(xh) = \perp$:

A contradiction

Then let us assume the opposite: $h(xh) = f_{xh}(xh) = \perp$

Then $g(xh,xh) = 1$, that is, $f_{xh}(xh) \neq \perp$:

Another contradiction

then the assumption was wrong

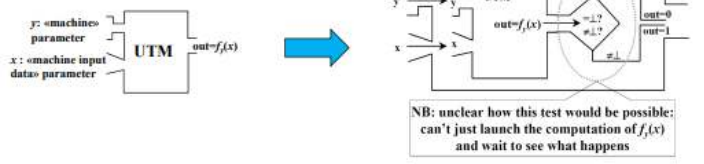
QED

PICTORIAL REPRESENTATION OF THE UNSOLVABILITY OF HALTING

176

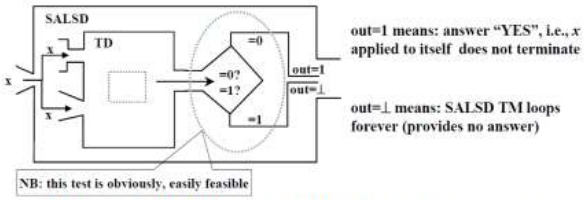
$g(y,x)=1$ if $f_y(x)\neq\perp$, $g(y,x)=0$ if $f_y(x)=\perp$

g is a "Termination Detecting" function (TD)



We introduce function $h(x)=1$ if $g(x,x)=0$, $h(x)=\perp$ if $g(x,x)=1$

h detects if a function, applied to itself, loops forever (the opposite of terminating), but otherwise *h* (i.e., the TM that computes *h*) does not "reply" (it goes into a loop); we say that it "semidetects" loops; therefore we call *h* "Self Application Loop Semi Detecting" function (SALSD)

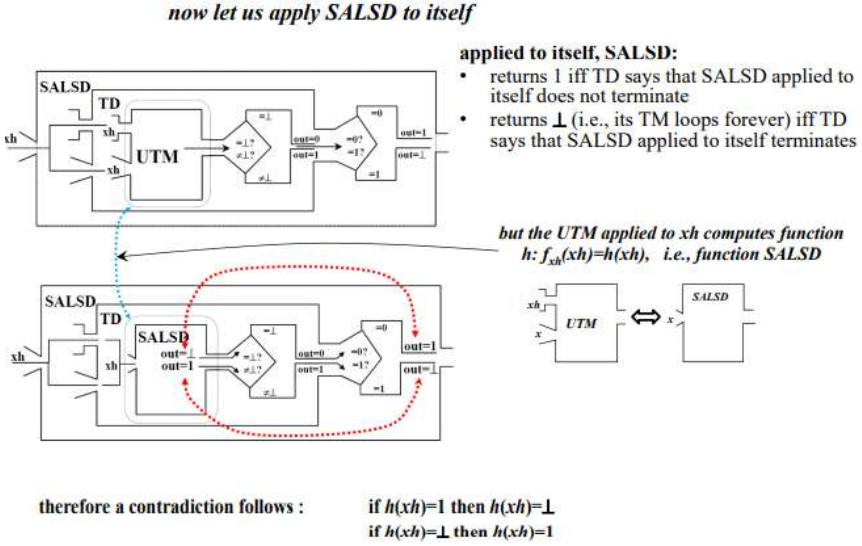


out=1 means: answer "YES", i.e., *x* applied to itself does not terminate

out=⊥ means: SALSD TM loops forever (provides no answer)

h is a variant of TD that is very easy to compute *if TD is computable*

assume SALSD=$h=f_{xh}$, i.e., *xh* is the Goedel # of SALSD



therefore a contradiction follows : if $h(xh)=1$ then $h(xh)=\perp$
if $h(xh)=\perp$ then $h(xh)=1$

Primo corollario dell'irrisolvibilità del problema dell'arresto per una TM:

- Il predicato (quindi tutta la funzione) caratterizzato dall'insieme delle funzioni calcolabili che sono definite quando applicate a loro stesse (come l'esempio di prima)

$$h(x)=1 \text{ if } f_x(x)\neq\perp;\ \ h(x)=0 \text{ if } f_x(x)=\perp$$

  non sono calcolabili.
- In generale, se un problema è irrisolvibile, un suo specifico caso può essere calcolato. Ma vale anche che, se uno specifico caso è calcolabile, un problema generale del caso è necessariamente irrisolvibile.

Un altro importante problema non risolvibile

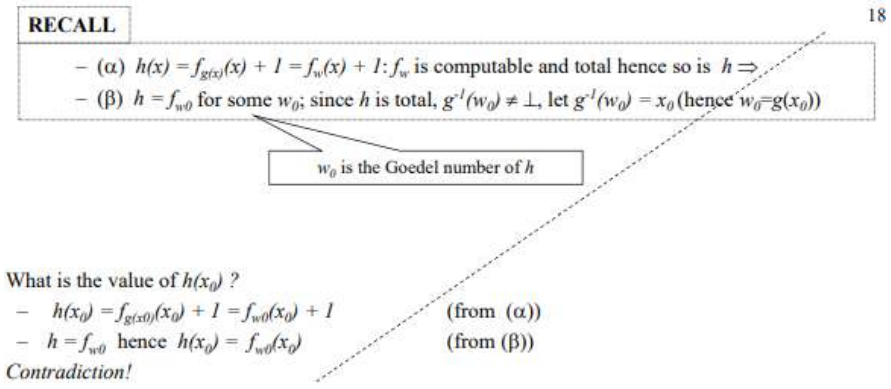
La funzione $k(y) = 1$ *se* f_y è ***totale***, *cioè se* $f_y(x) \neq \perp \forall\ x \in \mathrm{N},\ k(y) = 0$ altrimenti
non è calcolabile. In particolare, questo problema è simile ma di poco differente al precedente. Qui abbiamo una quantificazione di tutti gli input possibili. In questo caso, il problema è il non poter stabilire se appunto la funzione è totale, poiché per alcuni input lo è ma non per altri.
Da un punto di vista pratico, questo problema è forse anche più rilevante del problema di arresto: dato un programma, si vuole sapere se terminerà l'esecuzione **per ogni** input dato o se, per alcuni, entrerà in un loop infinito. Nel problema dell'arresto, volevamo invece sapere se il programma terminava con **alcuni** input.
Dimostrazione

Standard technique: diagonal + contradiction, with some more technical detail.
Hypothesis: $k(y) = 1$ if f_y *is total*, i.e., if $f_y(x) \neq \perp\ \forall\ x \in \mathcal{N}$; otherwise $k(y) = 0$ is computable and obviously, by definition, total
Then define $g(x) = w$ = index (Goedel number) of the x-th TM (in $\mathcal{E}$) that computes a total function.
If k is computable and total, then so is g:

- compute $k(0)$, $k(1)$, ..., let w_0 the first value such that $k(w_0) = 1$, then let $g(0) = w_0$;
- then let $g(1) = w_1$, w_1 being the second value such that $k(w_1) = 1$; ...
- the procedure is algorithmic; furthermore, being total functions infinite in number, $g(x)$ is certainly defined for each x, hence it is total.

g is also strictly monotonic: $w_{x+1} > w_x$;
hence g^{-1} is also a function, strictly monotonic too, though not total: $g^{-1}(w)$ is defined only if w is the Goedel number of a total function.
next define

- (α) $h(x) = f_{g(x)}(x) + 1 = f_w(x) + 1$: f_w is computable and total hence so is $h \Rightarrow$
- (β) $h = f_{w0}$ for some w_0; since h is total, $g^{-1}(w_0) \neq \perp$, let $g^{-1}(w_0) = x_0$ (hence $w_0 = g(x_0)$)



Ricordiamo però che sapere se un problema è risolvibile, non implica che ne conosciamo la soluzione!
Come in alcune nozioni matematica, spesso abbiamo delle dimostrazioni non-costructive, cioè prendiamo per vero che un oggetto (una nozione, teorema, ecc.…) esiste, senza trovarlo (verificarlo).
Per un automa possiamo dire che:

- Un problema è risolvibile se esiste una TM che lo risolve
- Per molti problemi, possiamo concludere che esiste una TM che li risolve, ma, per le poche conoscenze che abbiamo, non siamo in grado di trovarli o non sappiamo quale esattamente, in un insieme di TM, include la macchina che è in grado di risolverlo.

Alcuni esempi:

Is it true that the number of atoms in the universe is $10^{10^{10^{10}}}$?
Is it true that the "perfect chess game" will end in parity?
(30 years ago ...) Is it true that $\neg\exists x, y, z, w \in \mathcal{N}(x^y + z^y = w^y \wedge y > 2)$? (proved in 1994)

Possiamo chiederci se esiste una risposta chiusa (closed question, problemi che hanno come risposta yes/no, o comunque, un output non dipendente dall'input. Ammettono solo un numero finito di output) ai seguenti problemi. Per descrivere i seguenti problemi possiamo utilizzare le seguenti funzioni

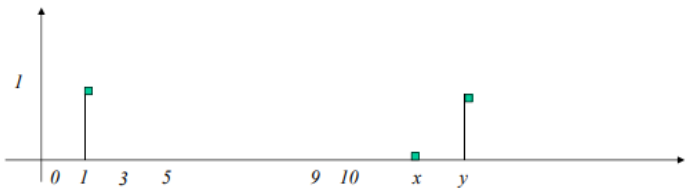
$$f_1(x) = 1 \ \forall x, o \ f_0(x) = 0 \ \forall x$$

Entrambe le funzioni sono calcolabili, poiché costanti, ma non ne conosciamo la risposta. Possiamo quindi concludere che alcuni problemi sono calcolabili, ma non se ne conosce necessariamente una soluzione.

More abstractly, considering for instance function $g(y, x)$ of the halting problem:
$g(10,20) = 1$ if $f_{10}(20) \neq \perp$, $g(10,20) = 0$ if $f_{10}(20) = \perp$
$g(100,200) = 1$ if $f_{100}(200) \neq \perp$, $g(100,200) = 0$ if $f_{100}(200) = \perp$
$g(7,28) = 1$ if $f_7(28) \neq \perp$, $g(7,28) = 0$ if $f_7(28) = \perp$
….
Hence $g(10,20)$ (i.e.: does TM M_{10} stop on input 20?), $g(100,200)$, $g(7,28)$ *etc.* are all solvable problems, though we do not necessarily know the solution (i.e., the value of g for those arguments).

Consideriamo adesso alcuni casi meno banali e più costruttivi:

- $f(x) =$ x-th cifra decimale dell'espansione di π
  $f(x)$ è certamente calcolabile, infatti conosciamo algoritmi (TM) che lo riescono a calcolare
- $g(x) = 1$ se da qualche parte, nell'espansione di π, esistono esattamente x cifre consecutive di 5, 0 altrimenti
- Considerando la sequenza $\pi = 3.14159 \ldots$
  Sappiamo che:
  $\{f(0) = 3, f(1) = 1, f(2) = 4, f(3) = 1, f(4) = 5, f(5) = 9, \ldots\}$
- E quindi concludiamo che $g(1) = 1$, cioè che esistono delle sequenze di esattamente 1 cifra consecutiva di 5:



E che $g(1) = 1$ è **semidecidable**, cioè non esiste un corretto modo di derivare la risposta.

- Se la seguente congettura: "Per ogni input x, producendo un'opportuna lunghezza di sequenza di π, prima o poi troveremo esattamente 5 x cifre consecutive", fosse stata vera, avremmo potuto concludere che la funzione *g* è una costante

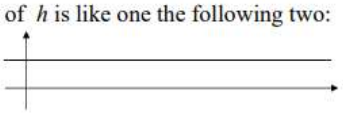
$$g(x) = 1 \ \forall x$$

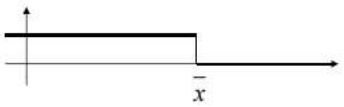
  E quindi avremmo detto che *g* è calcolabile
- Allo stato attuale però, non possiamo dire che *g* è calcolabile

Consideriamo un esempio di poco differente dal precedente:

- $h(x) = 1$ se in π esistono almeno x cifre di 5 consecutive, 0 altrimenti (notiamo che se *g* è 1 allora anche *h* sarà 1)
- Notiamo inoltre che per ogni x:
  - Se $h(x) = 1$, allora $h(y) = 1 \ \forall$ y ≤ x, cioè diciamo che *h* è downward closed for value 1 (chiusa al di sotto del valore 1)
  - Se $h(x) = 0$, allora $h(y) = 0 \ \forall$ y ≥ x, cioè diciamo che *h* è upward closed for value 0 (chiusa al di sopra del valore 0)

Hence the plot of h is like one the following two:

1)  $(h(x) = 1 \ \forall x)$

2)  $h(x) = 1 \ \forall \ x \leq \bar{x}$, $h(x) = 0 \ \forall \ x > \bar{x}$

  - Quindi *h* appartiene sicuramente al seguente insieme di funzioni:
    $\{h_{\bar{x}} | \ h_{\bar{x}}(x) = 1, \forall \text{ x} \leq \ \bar{x} \wedge h_{\bar{x}}(x) = 0, \forall \ x > \ \bar{x}\} \cup \{\bar{h} \mid \bar{h}(x) = 1 \ \forall \text{ x}\}$
    Ogni funzione scritta sopra è calcolabile, in quanto per ogni $\bar{x}$ fissata, è immediato costruire una TM che la calcola, la stessa cosa vale per $\bar{h}$
  - Siccome *h* è sicuramente computabile, allora esiste una TM che la calcola

26. Decidability and Semidecidability

Concentriamoci sui problemi espressi in modo tale che la risposta sia binaria (la risposta, o per meglio dire, l'output, assume solo due valori). Ricordiamo che ogni problema può essere riespresso in tal modo.
La funzione caratteristica o il predicato caratteristico di un insieme S è quello che caratterizza l'insieme:

$$c_s(x) = 1 \ if \ x \in S, c_s(x) = 0 \ if \ x \notin S, \qquad dove \ c_s \text{è} \ totale \ per \ definizione$$

(Stiamo essenzialmente dicendo che la funzione/predicato caratteristica/o assume valore 1 se l'input soddisfa S, valore 0 altrimenti)

Un set (insieme) S è ricorsivo (R) o ***decidable*** se e soltanto se la sua funzione caratteristica è calcolabile. Problemi risolvibili sono quindi ***decidable.***

Un set (insieme) S è ricorsivo enumerabile (RE) o ***semidecidable*** se e solo se:

- S è un insieme vuoto, o altirmenti,
- S è l'immagine di una funzione g_s totale e calcolabile chiamata **generating function** di S:
  - $S = I_{g_s} = \{x \mid x = g_s(y), y \in N\}, \ questo \ implica \ che$:

$$S = \{g_s(0), g_s(1), g_s(2), g_s(3), \dots\}$$

  Che appunto ci dice perché la definiamo enumerabile

- Il termine "semidecidable" è anch'esso spiegato intuitivamente, dato il problema $x \in S$?, se sì, enumerando gli elementi di S, prima o poi troveremo x e potremo rispondere di sì alla domanda, se però x non appartiene ad S avremo un loop infinito senza risposta

TEOREMA:

A) Se S è ricorsivo, allora è anche RE
B) S è ricorsivo, se e solo se, S e il suo complementare $S^{=N} - S$ sono RE (due "semidecidabilities" formano una "decidability", in altre parole, rispondere "NO" divenata difficile tanto quanto rispondere "SI")
C) Corollario: la classe delle decidable (linguaggio, problemi, …) è chiusa rispetto all'operazione complemento

DIMOSTRAZIONI:

A): *S* recursive implies *S* RE

If S is empty it is RE by definition

Let us then assume $S \neq \emptyset$ and call c_s its characteristic function: note that, since $S \neq \emptyset$,

$\exists\, k \in S$, that is $\exists\, k\ c_s(k) = 1$

Let us define the generating function g_s as follows:
$g_s(x) = x$ *if* $c_s(x) = 1$, otherwise $g_s(x) = k$ (the k above)

g_s is total, computable (because so is c_s), and $I_{gs} = S$

$\rightarrow S$ is RE

NB: it is a *non-constructive* proof:
do we know if $S \neq \emptyset$? not necessarily…
We only know that if $S \neq \emptyset$ *there exists* g_s : this is enough for us!

B) *S* is recursive if and only if both *S* and *S*^ = $\mathcal{N}$-*S* are RE

B) equivalent to: (B.1 S recursive $\rightarrow$ both S and $S^\wedge$ RE) **and**
(B.2 both S and $S^\wedge$ RE $\rightarrow$ S recursive)

B.1.1) S recursive $\rightarrow S$ RE (already proved in part A)

B.1.2) S recursive $\rightarrow c_S(x)$ ($= 1$ *if* $x \in S$; $= 0$ *if* $x \notin S$) computable
$\rightarrow c_{S^\wedge}(x)$ ($= 0$ *if* $x \in S$; $= 1$ *if* $x \notin S$) computable
$\rightarrow S^\wedge$ recursive $\rightarrow S^\wedge$ RE

B.2) S RE $\rightarrow$ construct the enumeration $S = \{g_S(0), g_S(1), g_S(2), g_S(3), \ldots\}$
$S^\wedge$ RE $\rightarrow$ construct $S^\wedge = \{g_{S^\wedge}(0), g_{S^\wedge}(1), g_{S^\wedge}(2), g_{S^\wedge}(3), \ldots\}$
But $S \cup S^\wedge = \mathcal{N}$, $S \cap S^\wedge = \emptyset$

hence $\forall x \in \mathcal{N}$, x belongs to exactly one of the two enumerations $\rightarrow$
If one constructs the enumeration

$$\{g_S(0), g_{S^\wedge}(0), g_S(1), g_{S^\wedge}(1), g_S(2), g_{S^\wedge}(2), g_S(3), g_{S^\wedge}(3), \ldots\}$$

one can certainly find in it any x: if x is at an odd position, then $x \in S$, if it is at an even position then $x \in S^\wedge$. Hence c_S can be computed.

ALTRI RISULTATI IMPORTANTI:

S is RE $\leftrightarrow$ $\boldsymbol{S = D_h}$, with ***h*** computable and partial: $\boldsymbol{S = D_h} = \{x \mid h(x) \neq \perp\}$
and
S is RE $\leftrightarrow$ $\boldsymbol{S = I_g}$, with ***g*** computable and partial: $\boldsymbol{S = I_g} = \{x \mid \exists y \in \mathcal{N} :\ x = g(y)\}$

Proof is omitted here: it uses a quite useful and significant technique

The above theorem allows us to view RE sets as characterizing precisely the languages ***recognized/accepted*** by the Turing Machines

(NB not ***decided*** : **decide** and **recognize/accept** differ slightly)

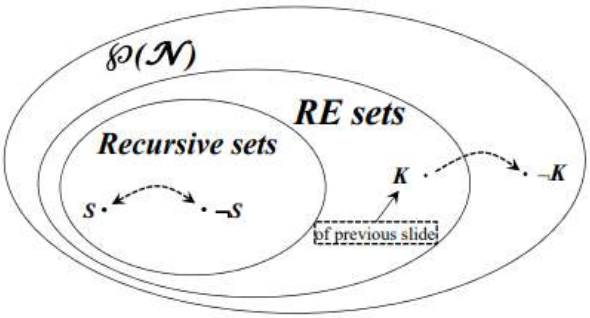
It can also serve as a Lemma to prove that:

There exist semidecidable sets that are not decidable:

$K = \{x \mid f_x(x) \neq \perp\}$ is semidecidable because $K = D_h$ with $h(x) = f_x(x)$.
We know however that the characteristic function of K,
($c_K(x) = 1$ *if* $f_x(x) \neq \perp$, 0 otherwise) is not computable (see function $h(x)$, slide 178)

$\Rightarrow K$ is not decidable

CONCLUSIONE:



BTW: notice that the relation «is the complement of» among sets is symmetric

Inclusions are all strict

Corollary: the class of RE sets (i.e., languages recognized by TM's) is *not* closed under complement

Why? Because if RE sets were closed under complement then all RE sets would also be recursive (also the complement would be RE), but we know (from the previous slide) that this is not the case

UN IMPORTANTE RIFLESSIONE SULL'HALTING PROBLEM COME CHIAVE DI R ≠ RE

Theorem: if Halting was decidable, then **R = RE**

- Suppose S is semidecidable: ⇒ (slide 189) ∃ a TM M_S that ***semidecides*** or ***recognizes*** it (slide 193), that is, for any x, M_S executed with input x
  - terminates and accepts if $x \in S$
  - does not terminate if $x \notin S$
- if Halting decidable ⇒ possible to tell whether M_S would halt on x or not ***in advance*** (without actually executing M_S)
  - if (we know that) M_S halts on x ⇒ accept x (determine that $x \in S$)
  - if (we know that) M_S does not halt on x ⇒ reject x (determine that $x \notin S$)
- therefore (under the assumption that halting is decidable) S would be not only RE but also R

27. Il potente Rice theorem

Definiamo F come una funzione computabile. L'insieme S degli indici di una TM che calcolano tale funzione sono definiti come

$$S = \{ x \mid f_x \in F \}$$

Tale insieme è decidable se e solo se $F = \emptyset$ o se F è l'insieme di tutte le funzioni calcolabili

Per determinare se un problema o insieme (set) è (semi)decidable o non lo è dobbiamo vedere le seguenti affermazioni:

- Se troviamo un algoritmo che termina sempre allora è decidable
- Se troviamo un algoritmo che potrebbe non terminare, ma termina sempre quando la risposta è positiva allora è semidecidable
- Usare il Rice Theorem
- Usare il **Problem Reduction**

28. Il Problem reduction

Se c'è un algoritmo che, data un'istanza di un problema P' calcola la sua soluzione trovando un'istanza di un algoritmo P che è risolvibile, e questa soluzione può essere trovata algoritmicamente da P, allora P' viene ridotto a P. Formuliamo così il problema:

I want to solve $x \in S'$
I can solve $y \in S$ for all possible y
If I have a computable, total function t such that $x \in S' \leftrightarrow t(x) \in S$
then I can answer algorithmically the question $x \in S'$
i.e., I have reduced the problem $x \in S'$ to the problem $y \in S$

Vale anche il viceversa:

I want to know if I can solve $x \in S$

I know I cannot solve $y \in S'$ (S' is *not* decidable)

If I find a computable total function t such that $y \in S' \leftrightarrow t(y) \in S$
then I can conclude that $x \in S$ is not decidable (otherwise I could show that $x \in S'$ is solvable by reducing it to $x \in S$)

Abbiamo già usato questo metodo implicitamente dalla undecidability dell'halting problem della TM dove abbiamo derivato la generalità della undecidability del problema della terminazione di ogni computazione di ogni computer. Consideriamo l'istanza della terminazione di un programma in C:

Consider a TM M_y and an integer x

I can build a C program, P, that simulates M_y and I can store x in an input file f

program P terminates its computation on file f if and only if $f_y(x) \neq \perp$

If I could decide if P terminates its computation on f then I could solve also the halting problem for the TM.

Per determinare se un problema è decidable utilizzando questa tecnica possiamo assumere, per assurdo (contraddizione) che è decidable e mostrare tramite riduzione che in quel caso l'halting problem sarebbe decidable, Consideriamo un'istanza P dell'halting problem e riduciamola al problema di non inizializzare una variabile, problema P^:

```
P^:
begin var   x, y: …
        P;
        y := x
end
```

making sure that identifiers x and y are "fresh variables", not used in P

Capiamo subito che se il programma restituisce un errore di variabile non inizializzata vuol dire che il programma P è terminato, risolvendo così l'healting problem.

La stessa tecnica può essere applicata ad altri programmi

Let us consider again the previous examples

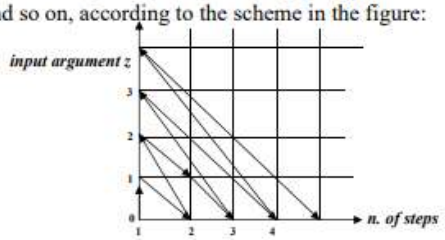
- halt of the TM
- Division by 0 and other run-time errors, …

The related sets are undecidable, but they are semidecidable:

a. if the TM stops, soon or late I find it;
b. if there exists any datum x, input of a program P, such that P, executed on input x, eventually executes a division by 0, then soon or late I can find it …

TEOREMA GENERALE REDUCTION:

- The set of values x for which $\exists\, z$ such that $f_x(z) \neq \perp$ is semidecidable
  (NB: here the "input parameter" of the problem is the TM index x)
- Sketch of the proof
  - If I compute $f_x(0)$ and find it $\neq \perp$ then there is no problem in providing an answer;
  - The problem arises if the computation of $f_x(0)$ does not terminate and $f_x(1) \neq \perp$: how can I find it?
  - Then I use the following trick, known as ***dovetailing***:
    - I simulate 1 execution step of $f_x(0)$: if it stops, then I have answered positively the question;
    - Otherwise I can simulate one computation step of $f_x(1)$;
    - Again if it does not stop I can simulate 2 steps of $f_x(0)$; next 1 step of $f_x(2)$; 2 steps of $f_x(1)$; 3 of $f_x(0)$; and so on, according to the scheme in the figure:



This way, if $\exists\, z$ s.t. $f_x(z) \neq \perp$, eventually I find it because eventually I will simulate enough computation steps of $f_x(z)$ to reach the stop

Conclusioni:

we have therefore a significant number of problems/sets (typically, related to run time errors in programs) that are not decidable, but are semidecidable.

We must however pay attention to which is precisely the ***semi***decidable problem:

- detecting the ***presence*** of the error (i.e., if there is one I can find it)
  - NB: the semidecidable set is the set of erroneous programs
- Not its absence! The set in question is that of error-free programs

Notice however that, since the complement $\neg \mathbf{s}$ of a set $\mathbf{s} \in$ (RE – R) is not even RE (otherwise they both would be decidable),

The absence of errors (i.e., the ***correctness*** of a program with respect to an error) is not only not decidable, but it is not even semidecidable!

Important implications on verification by ***testing***

- Famous statement by Dijkstra: testing can prove the ***presence*** of errors, ***not*** their ***absence***

Hence, as an additional result, we obtain a systematic technique to prove that a (unsolvable) problem is not RE: by proving that its complement is RE.

29. The complexity of computing

Non siamo soddisfatti del sapere se un problema è risolvibile, vogliamo anche sapere quanto costa risolverlo.

Misuriamo il costo in base a:

- Costo di esecuzione (risorse fisiche necessarie):
  - Tempo
    - Tempo di compilazione
    - Tempo di esecuzione
  - Spazio

Time complexity of a TM

Per analizzare la complessità temporale, possiamo rappresentare una computazione come una sequenza di configurazioni (relazione indicata dal simbolo $\vdash$)

$$c = c_0 \vdash c_1 \vdash c_2 \vdash c_3 \dots \vdash c_r$$

$$T_M(x) = r \; se \; la \; computazione \; si \; ferma \; su \; c_r, \infty \; altrimenti$$

Space complexity of a TM

$$c = c_0 \vdash c_1 \vdash c_2 \vdash c_3 \dots \vdash c_r$$

$$\boldsymbol{S_M(x)} = \sum_{j=1}^{k} max\{|\alpha_{ij}|, i = 1, \dots, r\} \qquad \alpha_{ij} = valore \; del \; nastro \; j \; all'i - esimo \; step,$$

k è il numero di nastri, quindi non stiamo indicando altro che la somma delle massime occupazioni per ognuno

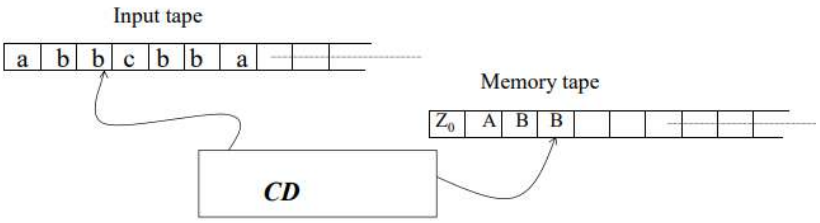
Relazione tra complessità di tempo e di spazio

$$\forall \boldsymbol{x}, \qquad \boldsymbol{S_M(x) \leq kT_M(x)}$$

Poichè $T_M(x)$ è anche la distanza che la testina può raggiungere

ESEMPIO:

A first example: accepting $\{wcw^R\}$, w∈{a,b}*



$T_M(x) = |x| + 1$ if $x \in L$

$T_M(x) = |w|+1$ if $x = saucu^Rb\,p$ (=> the TM rejects x upon reading the b after u^R)
and $w = saucu^R$, with $u,s,p \in \{a,b\}^*$

$T_M(x) = |x| + 1$ if $x \in \{a,b\}^*$ i.e., x does not contain any c

…

$S_M(x) = |x| + 1$ if $x \in \{a,b\}^*, \lfloor |x|/2 \rfloor + 1$ if $x \in L$, ...

Possiamo semplificare la nostra analisi, passando dall'analizzare la complessità come $f(x)$ (x input string) alla complessità di $f(n)$ (n "size" del dato in input x)

$$n = |x|,$$

$$cioè\ lunghezza\ della\ stringa, numero\ di\ righe\ e\ colonne\ di\ una\ mtarice, ecc\ \ldots$$

Se x è uguale a n, scegliamo di analizzare il caso peggiore

Choice of the **worst case**:
$T_M(n) = \max\{T_M(x), |x| = n\}$ (idem for $S_M(n)$)

Il caso medio è calcolato invece come:

$$T_M(n) = \frac{\sum_{|x|=n} T_M(x)}{k^n}, \quad k = \text{cardinality of the alphabet}$$

We want a simple, concise, precise and practical way to indicate how a complexity function $f(n)$ grows with its argument n

Use of the Θ notation to estimate the ***dominant factor*** in the growth of a (complexity) function and its asymptotic behavior

$$f \Theta g \leftrightarrow \exists c\ such\ that \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty$$

Θ is an ***equivalence relation*** therefore…

For any function $f(n)$ we may say that T_M is $\Theta(f)$, or $T_M \in \Theta(f)$

($\Theta(f)$ is viewed as an equivalence class) …

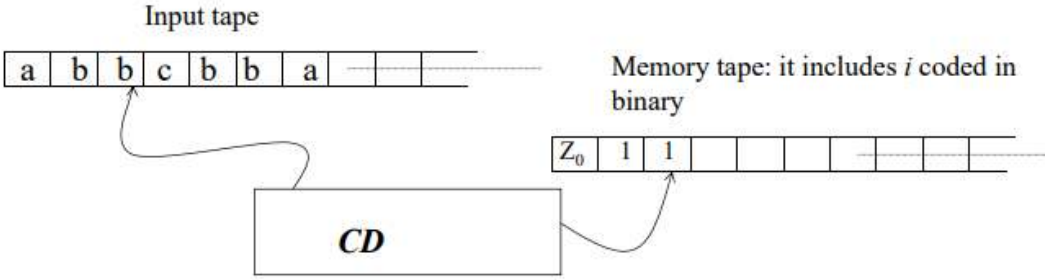
For instance, taking $T_M(n)=f(n)=5n^3+3n^2$, we may say T_M is $\Theta(n^3)$, or $T_M \in \Theta(n^3)$

(Saying that T_M is $\Theta(n)$ is like saying that $T_M(n)$ is linear?)

Successively we will see that the Θ notation not only shows the dominant growth factor, but it also, in a way, describes the part that is "independent from the power of the computing device"

Let's return to the example $\{wcw^R\}$

- $T_M(n)$ is $\Theta(n)$, $S_M(n)$ is also $\Theta(n)$
- Can one do anything better?
- Concerning $T_M(n)$ it is difficult (in general one has to at least read all the input string)
- Concerning $S_M(n)$:



Store in the memory tape only the position i of the symbol to examine; then move the scanning head in position i and n-i+1 to compare the two read symbols ===>

One gets: (let $n=|w|$)

$S_M(n)$: $\Theta(\log(n))$ (input is ***not*** copied to the memory tape)
but

$T_M(n)$: $\Theta(n^2 \cdot \log(n))$: use 2 tapes T_1 (main counter) and T_2 (auxiliary counter),

- $\forall i$, starting from 0
  - increment i (coded in binary on T_1) ($\Theta(\log(i))$ to implement $i := i+1$);
  - read next input symbol and store in the state
  - copy i on an auxiliary T_2 ($j := i$); go to opposite end of input
  - decrement (i times) j by 1 and move by 1 position the scanning head (towards the center) ($\Theta(i \cdot \log(i))$);
  - check current input with symbol stored in the state
- dominating factor is $\sum_{i=1}^{n} i \cdot \log(i) = n^2 \cdot \log(n)$

A typical ***time-space trade-off***

BTW: The example shows us why in the k-tape TM the scanning head can move in the two directions: otherwise one would lose important cases of sub-linear **spatial** complexity

For FA always $S_A(n)$ is $\Theta(k)$ and $T_A(n)$ is $\Theta(n)$, or even more precisely $T_A(n) = n$ (FA are ***real-time*** machines…);
For PDA always $S_A(n) \leq \Theta(n)$ and $T_A(n) \in \Theta(n)$ (number of ε-moves bounded *a priori*);
For ***single tape*** TM?

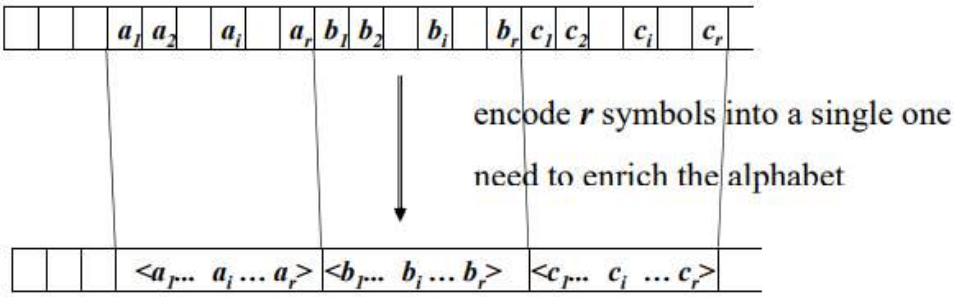
- Accepting $\{wcw^R\}$ requires, at first sight, $\Theta(n^2)$ (head goes up and back n times)
- Space complexity will never be $< \Theta(n)$
  (which provides an additional explanation of choosing k-tape TM as the principal model)
- Can one do better than $\Theta(n^2)$? NO: the proof is technically complex as it is often the case with non-trivial complexity lower bounds.
- (NB: a PDA accepts $\{wcw^R\}$ in $\Theta(n)$)

⇒

Hence, single tape **TM more powerful than PDA but sometimes less efficient**
What about von Neumann computers?
We'll see in a while ...

TEOREMA "SPEED UP" LINEARE (REDUCE SPACE COMPLEXITY E TIME COMPLEXITY)

- If L is accepted by a k-tape TM M with complexity $S_M(n)$, then for each $c > 0$ one can build a k-tape TM M' with complexity $\mathbf{S_{M'}(n) < c \cdot S_M(n)}$
  (=> one can "reduce space")



$r \cdot c \geq 2$ **e.g. to reach half the complexity (c=1/2) one must take $r \geq 4$**

If L is accepted by a **k-tape** TM M with space complexity $S_M(n)$, one can build a **1-tape** (NB: not a single tape) TM M' with complexity $S_{M'}(n) = S_M(n)$.
(=> one can "reduce #tapes")

If L is accepted by a k-tape TM M with space complexity $S_M(n)$, then for each $c > 0$ one can build a 1-tape TM M' with complexity $S_{M'}(n) < c \cdot S_M(n)$.
(=> one can "reduce space + #tapes")

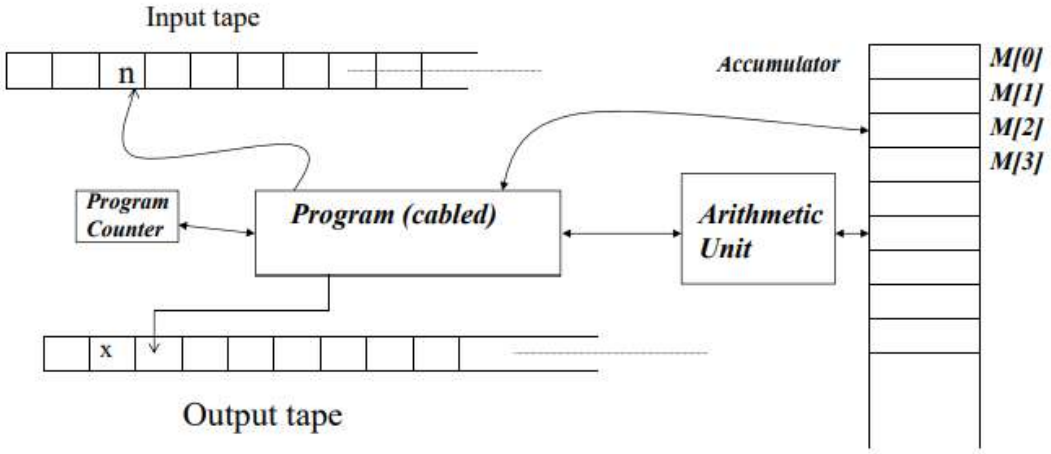
If L is accepted by a k-tape TM M with ***time*** complexity $T_M(n)$, then for every $c > 0$ one can devise a (k+1-tape) TM M' with complexity
$T_{M'}(n) = \max\{n+1, c \cdot T_M(n)\}$
(=> one can "reduce time")

Proof schema is the same as the one for spatial complexity (collapse *r* adjacent cells of M into one cell of M'), with some additional technical detail:

- First, one must read and translate all the input (which requires *n* moves)
- (This will create some problems within the class $\Theta(n)$)
- Then M' can use a fixed number of moves (i.e., 6) to simulate groups of *r* moves of M
- then by suitably choosing *r* one can reduce the complexity by an arbitrary (linear) factor

30. Astrazione del modello di un computer: the RAM machine

Per calcolare una somma, una TM impiega $\theta(n)$, mentre un computer risolve questa operazione come un'operazione elementare e quindi in un unico step. Inoltre, un Computer può accedere a una qualsiasi porzione di memoria, mentre una TM ha un accesso alla memoria solo sequenziale. Queste differenze ci fanno capire che non possiamo approssimare un TM a un computer e quindi neanche approssimarne i costi in termini di spazio e di tempo. Ci viene in soccorso la **RAM machine** schematizzata come di seguito:



Every cell contains an integer, not a symbol (NB!)

Instruzioni:

LOAD [=, *] X		M[0] := M[X], X, M[M[X]]
i.e.,	LOAD X	M[0] := M[X]
	LOAD =X	M[0] := X
	LOAD * X	M[0] := M[M[X]]

STORE [*]	X	M[X] := M[0], M[M[X]] := M[0]
ADD [=, *]	X	M[0] := M[0]+M[X], …
SUB, MULT, DIV		…
READ [*]	X	
WRITE [=, *]	X	
JUMP	lab	PC := b(lab)
JZ, JGZ, ...	lab	
HALT		

Esempio di una RAM machine che calcola la funzione is_prime(n), restituisce 1 se n è primo, 0 altrimenti:

```
        READ   1      The input value n is stored in cell  M[1]
        LOAD=  1      If n = 1, it is trivially prime ...
        SUB    1
        JZ     YES
        LOAD=  2      the counter in M[2] is initialized to  2
        STORE  2
LOOP:   LOAD   1      If M [1] = M[2]  (i.e., counter = n)  then n is prime
        SUB    2
        JZ     YES
        LOAD   1      If M [1] = (M[1]div M[2] ) * M[2] then
        DIV    2      M[2] is a divisor of M[1];
        MULT   2      hence M[1] is not prime
        SUB    1
        JZ     NO
        LOAD   2      the counter in M[2] is incremented by 1 and the loop is repeated
        ADD=   1
        STORE  2
        JUMP   LOOP
YES     WRITE=        1
        HALT
NO      WRITE=        0
        HALT
```

Calcoliamo i costi del seguente algoritmo:

$$S_R(n) \text{ è } \theta(2)$$

$T_R(n)$ è $\theta(n)$, dove n non è la lunghezza della stringa in input poiché costante

Abbiamo un porblema di approssimazione che viene risolto con i logaritmi

COSTI LOGARITMICI DELLA RAM MACHINE (l(x) è la lunghezza di x):

LOAD=	x	l(x)
LOAD	x	l(x) + l(M[x])
LOAD*	x	l(x) + l(M[x]) + l(M[M[x]])
STORE	x	l(x) + l(M[0])
STORE *	x	l(x) + l(M[x]) + l(M[0])
ADD=	x	l(M[0]) + l(x)
ADD	x	l(M[0]) + l(x) + l(M[x])
ADD *	x	l(M[0]) + l(x) + l(M[x]) + l(M[M[x]])
…		
READ	x	l(value of current input) + l(x)
READ*	x	l(value of current input) + l(x) + l(M[x])
WRITE=	x	l(x)
WRITE	x	l(x) + l(M[x])
WRITE *	x	l(x) + l(M[x]) + l(M[M[x]])
JUMP	lab	1
JGZ	lab	l(M[0])
JZ	lab	l(M[0])
HALT		1

Possiamo ricalcolare quindi il costo della funzione is_prime come segue:

M[1] stores *n*
M[2] stores the counter

LOOP:	LOAD	1	1+l(n)	
	SUB	2	l(n) +2 + l(M[2])	
	JZ	YES	l(M[0])	
	LOAD	1	1+l(n)	
	DIV	2	l(n) +2 + l(M[2])	
	MULT	2	l(n/M[2]) +2 + l(M[2])	($< 2 \cdot l$(n))
	SUB	1	l(M[0]) +1 + l(n)	($< 2 \cdot l$(n) +1)
	JZ	NO	… ≤ l(n)	
	LOAD	2	… ≤ l(n) + k	
	ADD=	1	...	
	STORE	2		
	JUMP	LOOP		

In conclusion, one can easily put an upper bound on the cost of the individual loop iteration as $\Theta(\log(n))$

Hence the overall time complexity is $\Theta(n \cdot \log(n))$

Le varie machine che abbiamo visto hanno costi differenti e non ce n'è una che è migliore di tutte le altre in tutti i casi. Possiamo però stabilire a priori una relazione tra le complessità dei vari modelli computazionali

POLYNOMIAL CORRELATION THEOREM (in analogy with the Church Thesis):

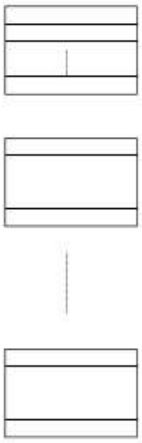
Under "reasonable" cost criterion hypotheses (the constant criterion for the RAM is not "reasonable" in all cases!) if a problem can be solved by a computation model M_1 with (space/time) complexity $\mathbf{C_1(n)}$ then there exists a a suitable ***polynomial*** $\mathbf{P_2}$ such that the same problem can be solved by ***any*** other computation model M_2 with complexity $\mathbf{C_2(n) \leq P_2(C_1(n))}$

Thanks to this result –and other important theoretical facts– the following analogy has been adopted:

- Class $\mathcal{P}$ of practically "tractable" problems = class of problems solvable in polynomial time
- $\mathcal{P}$ includes also the problems with complexity n^{1000} (better, at any rate, than the exponential ones), but practical experience shows that the problems with any practical applications (searches, paths, optimizations, …) that are in $\mathcal{P}$ also admit solutions with an acceptable degree (exponent) in the polynomial
- (similarly we will see shortly that the complexity relation between TM and RAM is "close")

1: How a RAM can simulate a (k-tape) TM

The RAM memory simulates the TM memory:
1 RAM cell for each cell of the TM tape
However, instead of using blocks of adjacent RAM memory cells to simulate each tape, we associate a -k cells- block to each k-tuple of cells taken for each position of the tape, + a "base" block:



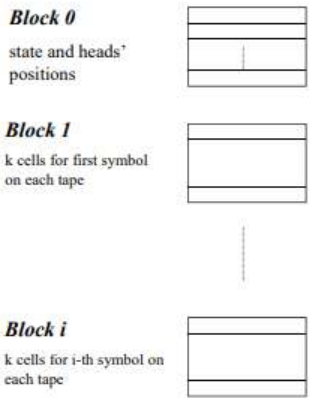
Block 0 — ***K+1 cells to store the state and the position of the k heads***

Block 1 — ***K cells to store the first symbol of every tape***

Block i — ***K cells to store the i-th symbol of every tape***

- A move of the TM is simulated by the RAM:

235



Block 0
state and heads' positions

Block 1
k cells for first symbol on each tape

Block i
k cells for i-th symbol on each tape

- ***Reading:***
- The content of the 0 block is checked (a packet of k+1 accesses, c*(k+1) moves)
- k indirect accesses to k blocks to examine the content of the cells corresponding to the heads
- ***Writing:***
- The state is changed (in block 0)
- The content of the cells corresponding to the head positions are updated through indirect STORE operations
- The values of the k heads positions are updated (in block 0)

A move of the TM requires h*k RAM moves:

With a constant cost criterion: $T_R \in \Theta(T_M)$

With a logarithmic cost criterion [the "serious" one]: $T_R \in \Theta(T_M.log(T_M))$ (an indirect access to *i* costs log(i))

2:

Time correlation between TM and RAM: (Theorem 3.17 on the textbook)

2: How a TM can simulate a RAM

(in a simple but central case: accepting languages (hence no output tape) without using MULT nor DIV: the generalization is trivial)

- The TM has 3 tapes: the first tape encodes the content of the RAM memory:

----------	$	i_j	#	$M[i_j]$	$	----------	$	i_k	#	$M[i_k]$	$	----------

- NB:
  - The various RAM cells are kept ordered (i.e., $i_j<i_{j+1}$)
  - Initially the tape is empty ---> at a generic time it includes only the cells that have been assigned a value (through a STORE)
  - i_j and $M[i_j]$ are represented in binary encoding
- 2 further tapes:
  - A tape includes the "accumulator register" M[0] (in binary)
  - A "service" tape

- How a RAM step is simulated by the TM:

----------	$	i_j	#	$M[i_j]$	$	----------	$	i_k	#	$M[i_k]$	$	----------

- Let us consider a representative example:
- LOAD h:
  - Look for the value of h on the main tape (if it is not found: error)
  - The part next to h, M[h], it is copied into (the tape that stores) M[0]
- STORE h:
  - Look for h. If it is not found, "make a hole" using the service tape
  - Store h and copy M[0] in the part next to it (M[h]); copy the successive part from the service tape
  - If h already exists copy M[0] in the part next to it (M[h]); this can require the use of the service tape if the number of cells already occupied is not equal to that of M[0].
- ADD* h:
  - Look for h; look for M[h]; …
- With an easy generalization:
- ***Simulating a RAM move can take the TM a number of moves with an upper bound c*(length of the main tape that stores the content of the RAM memory).***

- Now a Lemma (Lemma 3.18 on the textbook):
  the length of the main tape has an upper bound equal to a function not greater than $\Theta(T_R)$

----------	$	i_j	#	$M[i_j]$	$	----------	$	i_k	#	$M[i_k]$	$	----------

Each "i_j-th cell" of the RAM requires in the tape $l(i_j) + l(M[i_j])$ (+2) tape cells.

Each "i_j-th cell" exists in the tape if and only if the RAM has executed at least a STORE on it.

The STORE costs for the RAM is $l(i_j) + l(M[i_j])$ ---->

To fill r cells, of total length

$$\sum_{j=1,r} l(i_j) + l(M[i_j])$$

the RAM needs a time (w.r.t. logarithmic cost criterion) that is at least proportional to the same value.

Hence to simulate a RAM move, the TM needs a time at most $\Theta(T_R)$;

a RAM move costs *at least* 1;

if the RAM has complexity T_R, the TM executes *at most* T_R moves ---> the complete simulation of the RAM by the TM costs *at most* $\Theta(T^2_R)$.

UNO DEI SETTE PROBLEMI DEL MILLENNIO

$\mathcal{NP}$: the class of problems that can be solved ***non***deterministically in polynomial time

$\mathcal{P}$: the class of problems that can be solved deterministically in polynomial time (the ***tractable*** problems)

A momentous question : $\mathcal{P} = \mathcal{NP}$?

Most likely not. However, surprisingly … this crucial question has not been answered!

If $\mathcal{P} = \mathcal{NP}$ we might solve efficiently an enormous quantity of problems that now are intractable, and must be addressed by means of heuristics, by special cases, etc.

The notion of ($\mathcal{NP}$) ***completeness***: a "representative" of a class incorporates the essence of all problems of the class: if we find the solution for it we have it for all!

- $\mathcal{NP}$ completeness is based on the existence of ***polynomial-time reductions***: a problem is reduced to another one by means of a polynomial-time transformation (total computable function)

It is interesting to notice that in the enormous set of $\mathcal{NP}$ problems, a great number is also $\mathcal{NP}$-complete: it would suffice to solve one of then in (deterministic) polynomial time and we would have $\mathcal{P} = \mathcal{NP}$; it would suffice to prove that one of them is intractable and all other ones would also be so!

Finally: nondeterministic is not a synonym for random, but … random computations are very effective and promising.