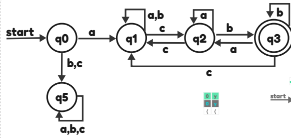
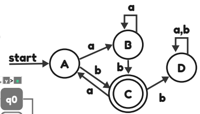
Random stuff





DFA conversion

!	Node	N->	a	a->	b	b->
*->	{q0}	A	{q1,q2}	B	∅	C
*	{q1,q2}	B	{q1,q2}	B	{q1,q3}	D
	∅	C	∅	C	∅	C
*	{q1,q3}	D	{q1,q2}	B	∅	C

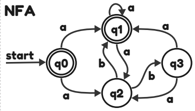
Coming soon! ;) *

* probably never :(

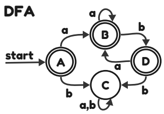
2 nov - 1

Esercizio 1

NFA



DFA

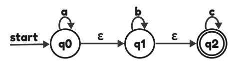


Conversione a DFA

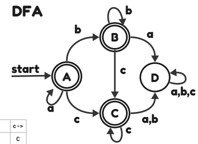
!	Nodo	N->	a	a->	b	b->
*->	{q0}	A	{q1,q2}	B	∅	C
*	{q1,q2}	B	{q1,q2}	B	{q1,q3}	D
	∅	C	∅	C	∅	C
*	{q1,q3}	D	{q1,q2}	B	∅	C

Esercizio 2

Ɛ-NFA



DFA



Conversione a ε-DFA

!	Nodo	N->	a	a->	b	b->	c	c->
*->	{q0,q1,q2}	A	{q1,q2,q3}	A	{q1,q2}	B	{q2}	C
*	{q1,q2}	B	∅	D	{q1,q2}	B	{q2}	C
*	{q2}	C	∅	D	∅	D	{q2}	C
	∅	D	∅	D	∅	D	∅	D

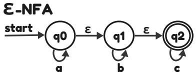
3 nov - 1

Definizione δ^

```
Caso base:
if the |w|  = 0 (length 0
δ^(T,w) = δ^(T,ε) = eclose(T)

T = stato iniziale
P = eclose(T)
S = δ(P,a)
R = eclose(S)

δ^(T,ax) = δ^(R,x)
(definizione ricorsiva)

Example for dummies:
δ^(q0,abc) = (R,bc)
P = eclose(q0) = {q0,q1}
S = δ(P,a) = δ(q0,a) ∪ δ(q1,a) = {q2}
R = eclose(S) = {q2,q3}
```

Esercizio 1

Ɛ-NFA



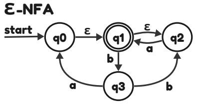
ER = a*b*c*

```
δ^(q0,aabc) =

P = eclose(T) = eclose(q0) = {q0,q1,q2}
S = δ(S,a) = δ({q0,q1,q2},a) = δ(q0,a) ∪ δ(q1,a) ∪ δ(q2,a) = {q0} ∪ ∅ ∪ ∅ = {q0}
R = eclose(S) = eclose({q0}) eclose(q0) = {q0,q1,q2}

= δ^({q0,q1,q2},abc) =

P = eclose({q0,q1,q2}) = eclose(q0) ∪ eclose(q1) ∪ eclose(q2) = {q0,q1,q2} ∪ {q1,q2} ∪ {q2} = {q0,q1,q2}
Q = δ(P,a) = δ(q0,a) ∪ δ(q1,a) ∪ δ(q2,a) = {q0} ∪ ∅ ∪ ∅ = {q0}
R = eclose(S) = {q0,q1,q2}

= δ^({q0,q1,q2},bc) =

P = eclose({q0,q1,q2}) = {q0,q1,q2}
Q = δ(P,b) = δ(q0,b) ∪ δ(q1,b) ∪ δ(q2,b) = ∅ ∪ {q1} ∪ ∅ = {q1}
R = eclose(Q) = {q1,q2}

= δ^({q1,q2},c) =

P = eclose({q1,q2}) = {q1,q2}
Q = δ(P,c) = δ(q1,c) ∪ δ(q2,c) = ∅ ∪ {q2} = {q2}
R = eclose(Q) = {q2}

= δ^({q2},ε) = δ^(q2,ε) = eclose(q2) = {q2}

Almeno uno degli stati è finale (q2): aabc è accettato.
```

3 nov - 2

Esercizio 1

Ɛ-NFA

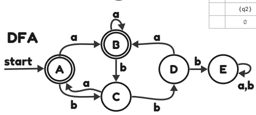



```
eclose(q0) = {q0,q1,q2}
eclose(q1) = {q1,q2}
eclose(q2) = {q2}
eclose(q3) = {q3}
```

DFA conversion

!	Nodo	N->	a	a->	b	b->
*->	{q0,q1,q2}	A	{q1,q2}	B	{q3}	C
*	{q1,q2}	B	{q1,q2}	B	{q3}	C
	{q3}	C	{q0,q1,q2}	A	{q2}	D
	{q2}	D	{q1,q2}	B	∅	E
	∅	E	∅	E	∅	E

DFA

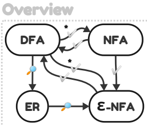



```
δD({q0,q1,q2},a) =
= eclose(δE(q0,a) ∪ δE(q1,a) ∪ δE(q2,a)) =
= eclose(∅ ∪ {q1} ∪ ∅) = {q1,q2}

δD({q0,q1,q2},b) =
= eclose(δE(q0,b) ∪ δE(q1,b) ∪ δE(q2,b)) =
= eclose(∅ ∪ ∅ ∪ {q3}) = {q3}
```

Altri esercizi disponibili nella lezione

4 nov - 1

Overview

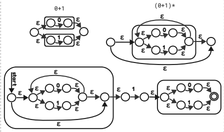


* Nota: per convertire da NFA/ε-NFA a DFA basta prendere la tabella del DFA e mettere gli stati in parentesi graffe. Gli insiemi risultanti saranno tutti "singoletti" (un solo elemento).

Esercizio 3.8 p. 99

ER = (0+1)* 1 (0+1)

(Tutte le stringhe binarie che terminano o con 10 o con 11)



ER -> Ɛ-NFA

1. Esattamente uno stato accettante (!)
2. Nessun arco entrante nello stato iniziale
3. Nessun arco uscente dallo stato accettante

Per induzione:

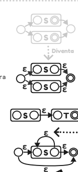
Base

1. Se R = ε (e quindi L(R) = {ε}), allora
2. Se R = ∅ (e quindi L(R) = ∅), allora
3. Se R = **a**, e quindi L(**a**) = {a}, allora

(Qualsiasi stringa gli diamo non va bene, perché non vuole niente)

Passo induttivo

1. **Unione**: se R = S + T, e quindi L(R) = L(S) ∪ L(T), allora
2. **Concatenazione**: se R = ST, e quindi L(R) = L(S)•L(T), allora
3. **Chiusura di Kleene**: se R = S*, e quindi L(R) = (L(S))*, allora
4. **Parentesi**: Se R = (S), e quindi L(R) = L(S), l'ε-NFA per S vale anche per R



Non va bene: ci sono due stati accettanti

Ci possono essere 0 o più concatenazioni di stringhe denotate da S

In questo modo accetta anche le stringhe vuote

DFA -> ER

Algoritmo per eliminazione degli stati: toglie tutti gli stati che non sono iniziali o finali. Eliminandoli, cambiamo le etichette degli archi, che diventeranno espressioni regolari. Alla fine avremo due casi:

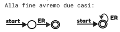
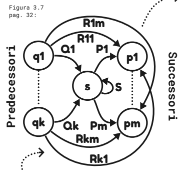
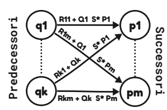


Figura 3.7 pag. 32:



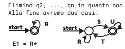
Cosa succede se **eliminiamo lo stato S**? Per andare da un predecessore a un successore, c'è la strada diretta, più tutte quelle che passano da S (facendo giri se c'è il cappio). Bisogna cambiare l'etichetta delle strade dirette:



Osservazioni:
1. Le etichette sono **ER**.
2. Predecessori e successori possono **coincidere**.
3. Se l'**arco non c'è**, lo mettiamo comunque, e lo etichettiamo ∅.
4. s può avere un **cappio**.

Ricapitolando:
Abbiamo un DFA con
- q0 -> **stato iniziale**
- q1,q2, ..., qn -> **stati finali**
- **altri stati**

Per prima cosa eliminiamo gli altri stati, con **eliminazione**. Poi mi concentro su q1, che considero come unico stato finale. Elimino q2, ..., qn in quanto non finali (**eliminazione**). Alla fine avremo due casi:



E1 = R*

E1 = (R + S U* T)* S U*

	(È uno dei modi possibili).
(R + S U* T)	Ciclo con **R**, oppure scelgo **S**, posso ciclare con **U**
(R + S U* T)*	e torno con **T**. Tutto questo può avvenire * volte.
	Alla fine però devo andare nello stato finale con
S U*	**S**, e posso ancora ciclare con **U**.

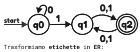
(R+SU*T)* SU*

Faccio questo con tutti gli stati finali q2, ..., qn, e otterrò: E2, ..., En.
L'ER finale è l'unione di tutte le ER:
E = E1 + E2 + ... + En

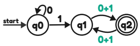
4 nov - 2

Esempio 1 DFA -> ER

DFA

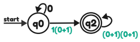


Trasformiamo **etichette** in **ER**:



Elimino q1. Devo vedere se ci sono:
Predecessori: q0, q2
  ER: q0 -> q1 = 1
    q2 -> q1 = 0+1
Successori: q2
  ER: q1 -> q2 = 0+1
Self-loop (no):
  ER: ∅
Collegamenti da ogni pred a ogni succ:
  R02 = ∅, R22 = ∅

Otteniamo:
R02 = ∅ + 1 ∅* (0+1) = 1(0+1)
R22 = ∅ + (0+1) ∅* (0+1) = (0+1)(0+1)



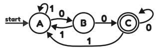
È come il secondo caso della teoria (4 nov-1):
Faccio il self-loop di **q0** * volte,
oppure vado in **q2**, faccio il self-loop di **q2** * volte, e torno in **q0**.
Infine vado un'ultima volta in **q2** e faccio il self-loop * volte.

(0 + 1(0+1) ((0+1)(0+1))* ∅)* 1(0+1) ((0+1)(0+1))*

  Sparisce perché c'è ∅

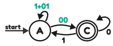
0* 1(0+1) ((0+1)(0+1))*
Oppure
0* 1(0+1) (00+01+10+11)*

Esempio 2 DFA -> ER



Le etichette sono già **ER**.
Elimino B.
Predecessori: A
  ER: A -> B = 0
Successori: A, C
  ER: B -> A = 1
    B -> C = 0
Self-loop: no
  ER: ∅
Collegamenti p->s:
  ER: RAA = 1
    RAC = ∅ (!)

Otteniamo: (diretto + per B)
RAA = 1 + 0 ∅* 1 = 1 + 01
RAC = ∅ + 0 ∅* 0 = 00



È ancora come il secondo caso della teoria (4 nov-1).

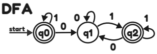
Self-loop A,
oppure vado in C, self-loop C * volte, torno in A.
Il tutto * volte.
Infine torno in C, e faccio self-loop in C * volte.

ER = (1 + 01 + 00 0* 1)* 00 0*

4 nov - 3

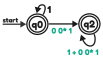
Esempio 3 DFA -> ER

DFA



Le etichette sono già **ER**.
Elimino q1.
Predecessori: q0, q2
  ER: q0 -> q1 = 0
    q2 -> q1 = 0
Successori: q2
  ER: q1 -> q2 = 1
Self-loop: sì
  ER: 0
Collegamenti p->s:
  ER: R02 = ∅
    R22 = 1

Otteniamo: (diretto + per q1)
R02 = ∅ + 0 0* 1 = 0 0* 1
R22 = 1 + 0 0* 1 = 1 + 0 0* 1



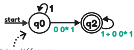
E1 = q0 finale, **elimino anche q2**.
Predecessori: q0
  ER: q0 -> q2 = **00*1**
Successori: nessuno
Self-loop: sì
  ER: **1+00*1**
Collegamenti p->s: nessuno
  Dato che non ci sono proprio i successori, e non solo l'arco come nel caso di R02, non ho nessuna ER.

Come in questo caso base:

ER = R*

E1 = **1***

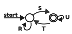
E2 = q2 finale, **q0 non finale**.



Unica differenza:
q0 **non** è finale

È come il secondo caso della teoria (4 nov-1).

ER = (R+SU*T)* SU*



E2 =
= (1 + 00*1 (1+00*1)* ∅)*00*1 (1+00*1)* =
= 1* 00*1 (1+00*1)*

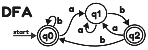
Risultato: E = E1 + E2 = **1* + 1*00*1(1+00*1)***

Se vogliamo, possiamo semplificare raccogliendo 1* all'inizio:

1* (ε + 00*1(1+00*1)*)

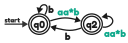
Esempio 4 DFA -> ER

DFA



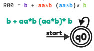
Le etichette sono già **ER**.
Elimino q1.
Predecessori: q0, q2
  ER: q0 -> q1 = a
    q2 -> q1 = a
Successori: q2
  ER: q1 -> q2 = b
Self-loop: sì
  ER: a
Collegamenti p->s:
  ER: R02 = ∅
    R22 = ∅

Otteniamo: (diretto + per q1)
R02 = ∅ + a a* b = a a* b
R22 = ∅ + a a* b = a a* b



E1 = q0 finale, **elimino anche q2**.
Predecessori: q0
  ER: q0 -> q2 = aa*b
Successori: q0
  ER: q2 -> q0 = b
Self-loop: sì
  ER: aa*b
Collegamenti p->s:
  ER: R00 = b

Otteniamo: (diretto + per q2)
R00 = b + aa*b (aa*b)* b

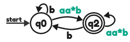


Come in questo caso base:

ER = R*

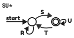
E1 = (b + aa*b (aa*b)* b)*

E2 = q2 finale, **q0 non finale**.



È come il secondo caso della teoria (4 nov-1).

ER = (R+SU*T)* SU*



E2 =
= (b + aa*b (aa*b)* b)* aa*b (aa*b)*

Risultato: E = E1 + E2 =
= (b+aa*b(aa*b)*b)* +
(b+aa*b(aa*b)*b)*aa*b(aa*b)*

Semplificato:

(b+aa*b(aa*b)*b)* (ε + aa*b(aa*b)*)

9 nov - 1

Correzione della simulazione del primo parziale disponibile nella lezione

9 nov - 2

Chiusura dei linguaggi regolari rispetto alle operazioni booleane

Supponiamo di avere due linguaggi regolari, L e M, definiti sullo stesso alfabeto.
L, M ∈ REG(Σ)

L'**unione** di questi linguaggi è regolare?
L ∪ M ∈ REG ? Sì.

L'**intersezione** di questi linguaggi è regolare?
L ∩ M ∈ REG ? Sì.

La **differenza** insiemistica è regolare?
L \ M ∈ REG ? Sì.

La **concatenazione** è regolare?
L · M ∈ REG ? Sì.

Il **complemento** di uno di questi linguaggi è regolare?
Γ = Σ* \ L ∈ REG ? Sì.

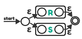
La **chiusura di Kleene** è regolare?
L* ∈ REG ? Sì.

Unione: dimostrazione 1

Se L e M sono regolari, allora ∃ due ER R, S tali che L(R) = L e L(S) = M.
Allora L(R+S) = L(R) ∪ L(S) = L ∪ M.

Unione: dimostrazione 2

Se L e M sono regolari, allora ∃ due ε-NFA R e S che li accettano.
Allora, l'ε-NFA



accetta L ∪ M.

Osservazione
Se **L** e **M** sono definiti su alfabeti diversi
L ∈ REG(**Σ1**), **M** ∈ REG(**Σ2**)
Σ = **Σ1** ∪ **Σ2**
allora **L** ∈ REG(**Σ**) **L** ⊆ **Σ***
  M ∈ REG(**Σ**) **M** ⊆ **Σ***

Concatenazione

Se L e M sono regolari, allora ∃ due ER R, S tali che L(R) = L e L(S) = M.
Allora L(R·S) = L(R) · L(S) = L · M.

Chiusura di Kleene

Se L è regolare, allora ∃ un ER R tale che L(R) = L.
Allora L(R*) = L*.

Complemento

Se L ∈ REG(Σ), allora ∃ un DFA A = {Q,Σ,δ,q0,F} tale che L = L(A).
Allora Γ = L(B), dove B è il DFA tale che A = {Q,Σ,δ,q0,Q\F} (scambio stati finali e non finali).

Osservazione
Se E è una ER tale che L(E) = L, come ricavo l'ER F tale che L(F) = L̄?

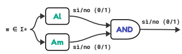
E -> ε-NFA -> DFA A per L -> DFA B per L̄ -> ER F

Intersezione: dimostrazione 1

Se L e M sono regolari, allora
L ∩ M = $\overline{\bar{L} \cup \bar{M}}$
L,M ∈ REG(Σ) -> L̄, M̄ ∈ REG(Σ)
-> L̄ ∪ M̄ ∈ REG(Σ) -> $\overline{\bar{L} \cup \bar{M}}$ ∈ REG (= L ∩ M)

Intersezione: dimostrazione 2 (automa prodotto)

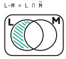
Se L e M ∈ REG(Σ), ∃ DFA Al = {QL,Σ,δL,qL,FL}
e Am = {QM,Σ,δM,qM,FM}:



Differenza

Se L, M ∈ REG(Σ), allora L \ M ∈ REG(Σ).

L-M = L ∩ M̄



M ∈ REG -> M̄ ∈ REG
L,M̄ ∈ REG -> L ∩ M̄ ∈ REG

9 nov - 3

L'automa prodotto

Due automi sincronizzati, che consumano la stessa stringa, e infine un controllo che ritorna true solo se entrambi gli automi la accettano.

Costruiamo l'automa prodotto A = Al ⊗ Am
A = {(Ql×Qm), Σ, δ, (ql,qm), (Fl×Fm)}
  = Q   = q0   = F

Nel prodotto cartesiano degli stati finali, sto già implicitamente facendo l'AND dei due automi, man mano.

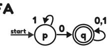
La funzione delta è δ((p,q),a) = (δL(p,a),δM(q,a))

Si può fare anche con altri gate logici.
AND: Intersezione
OR: Unione
XOR

Esempio 4.9 p. 129

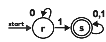
DFA

Al



Accetta stringhe binarie che contengono almeno uno 0

Am



Accetta stringhe binarie che contengono almeno un 1

Facciamo l'intersezione: stringhe binarie che contengono almeno uno 0 e almeno un 1.
Si può fare o con il diagramma o con la tabella.

Diagramma

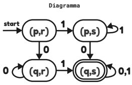


Tabella degli stati

Automa prodotto

!	Nodo	N->	0	0->	1	1->
->	(p,r)	A	(q,r)	B	(p,s)	C
	(q,r)	B	(q,r)	B	(q,s)	D
	(p,s)	C	(q,s)	D	(p,s)	C
*	(q,s)	D	(q,s)	D	(q,s)	D

(Si può fare lo stesso automa con i nomi invece delle coppie di stati.)

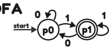
Altri esempi nella lezione

10 nov - 1

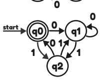
Automa prodotto che non accetta nulla

DFA

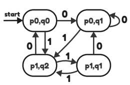
A



B



Diagramma



Nessuno stato finale: l'automa non accetta nulla (∅).

L(A) ≠ ∅   es: 1 ∈ L(A)
L(B) ≠ ∅   es: ε ∈ L(B)
L(A) ∩ L(B) = ∅

10 nov - 2, 3

Collegamento a laboratorio nella lezione (slide)

23 nov - 1

Minimizzazione dei DFA

- Riduzione al minimo del numero di stati del DFA
- Nessun DFA con meno stato di quello minimizzato può riconoscere lo stesso linguaggio regolare

DFA A = (Q,Σ,δ,q0,F)

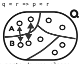
Definizione (equivalenza di stati)

p,q ∈ Q sono equivalenti (p ≈ q) se
∀ w ∈ Σ* vale δ^(p,w) ∈ F <=> δ^(q,w) ∈ F <=> δ^(q,w) ∈ F

I due DFA devono finire entrambi in uno stato finale/non finale data la stessa stringa, anche diverso.

≈ è **riflessiva** ∀ p ∈ Q, p ≈ p
simmetrica ∀ p,q ∈ Q, p ≈ q => q ≈ p
transitiva ∀ p,q,r ∈ Q, p ≈ q e q ≈ r => p ≈ r

≈ è una relazione d'equivalenza

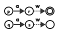


Se parto da una classe d'equivalenza A, finisco sempre nella stessa altra classe d'equivalenza B

Stati distinguibili (induzione):

Base:
se per F e q ∉ F, allora p ≉ q

Passo:



Se r ≉ s, allora p ≉ q

Bisogna vedere se i due stati non sono equivalenti
≉ -> **distinguibili**

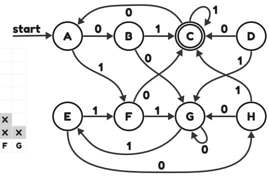
Deve esistere una stringa w che dà risultati diversi (uno finale e uno non finale) a partire da due stati: **distinguisher**.

Classi di equivalenza:
Non possono contenere stati finali o non finali: o tutti finali, o tutti non finali.

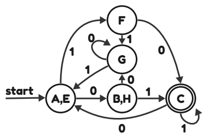
Esempio 1 minimizzazione DFA

Stati distinguibili

B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X
	A	B	C	D	E	F	G



Adesso parto da **A,E** e faccio tutti i collegamenti come nel DFA iniziale, solo con le classi di equivalenza al posto degli stati.



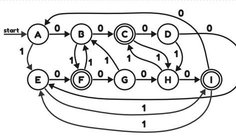
Proviamo a distinguere A e G.
ε: A,G: entrambi non finali
0: A->B, G->G : entrambi non finali
1: A->E, G->E : entrambi non finali
00: A->G, G->G: non finali
01: A->C, G->E: 01 è il distinguisher

{A,E} {B,H} {D,F}
Creiamo classi d'eq. con gli stati restanti: {C} {G}

Occhio: **D** ha solo archi uscenti e nessun arco entrante. Si può cancellare direttamente dalla tabella e dalle classi d'equivalenza.

23 nov - 2

Esempio 2 minimizzazione DFA



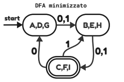
Stati distinguibili

B	X							
C	X	X						
D		X	X					
E	X		X	X				
F	X	X		X	X			
G		X	X		X	X		
H	X		X	X		X	X	
I	X	X		X	X		X	X
	A	B	C	D	E	F	G	H

Legenda:

ε	X	(Con 0 passaggi uno è iniziale, l'altro finale)
0	X	(Con 1 passaggio uno è iniziale, l'altro finale)
1	X	
0...	X	(Con 2+ passaggi uno è iniziale, l'altro finale)
1...	X	

DFA minimizzato



Classi di equivalenza:
{D,A}∪{G,A}∪{G,D} {E,B}∪{H,E}∪{H,B} {F,C}∪{I,F}∪{I,C} =

-> {A,D,G}
{B,E,H}
* {C,F,I}

23 nov - 3

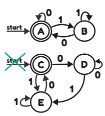
Equivalenza di DFA

Prendiamo due DFA:

Al = (Ql,Σ,δl,ql,Fl)
Am = (Qm,Σ,δm,qm,Fm)

Per capire se riconoscono lo stesso linguaggio, costruiamo un nuovo DFA:
A = (Ql ∪ Qm,Σ,δ,ql,Fl ∪ Fm)

Com'è fatto questo automa? Prendiamo un esempio.



B	X			
C		X		
D		X		
E	X		X	X
	A	B	C	D

Legenda:

ε	X
0	X
1	X
0...	X
1...	X

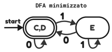
A e C sono equivalenti (non c'è la crocetta): i due automi **accettano lo stesso linguaggio**.

Cosa succede se minimizziamo il secondo automa?

D		
E	X	X
	C	D

Classi di equivalenza:
*->{C,D} {E}

DFA minimizzato

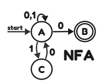


Se lo confrontiamo con il primo automa, è identico. Riducendo un automa B equivalente a un automa minimizzato A otteniamo nuovamente A.

Ricorda:
L'automa minimo ottenuto con l'algoritmo è **unico**.

Uniamo i due automi: formiamo un nuovo automa con l'unione degli insiemi degli stati dei due automi, stesso alfabeto, funzione delta che funziona come δl sul primo automa e δm sul secondo, stesso alfabeto, stato iniziale di uno dei due automi (**scegliamo il primo**), e stessi stati finali.
La tecnica consiste nell'applicare l'algoritmo riempi-tabella su {A,B,C,D,E} (anche se {C,D,E} non sono raggiungibili), e poi di valutare se i due stati iniziali A, C sono equivalenti tra loro.

Funziona con gli NFA? (no)



Proviamo a minimizzarlo con l'algoritmo riempi-tabella:

B	X	
C	X	X
	A	B

Risultano tutti e tre distinguibili: le nostre classi di equivalenza sono
{A},{B},{C}

Dunque, secondo l'algoritmo riempi-tabella l'automa è **già minimo**.
Tuttavia, se tolgo lo stato C posso osservare che l'automa continua ad accettare le stesse stringhe di prima, e quindi esiste un automa più semplice.

Il pumping lemma

Introduzione

Si usa per dimostrare che certi linguaggi **non** sono regolari.

Prendiamo un esempio:
L01 = {0^n 1^n | n ≥ 1} = {01, 0011, 000111...}

Noi sappiamo che dovendo avere lo stesso numero di 0 e di 1 non può essere regolare, ed è per forza context-free.

Per assurdo
Se L01 fosse regolare, esisterebbe un DFA A che lo accetta.
Supponiamo che A abbia k stati, e consideriamo la stringa 0^k.

Le prefisse di 0^k sono ε, 0, 00, 000, ..., 0^k

-> Significa che ho buttato via un certo numero di caratteri dal fondo. Con ε li ho buttati via tutti, con 0^k nessuno.

In totale sono **k+1** prefisse, e quindi una in più del numero di stati che ha il nostro automa. Gli stati dell'automa a stati finiti sono l'unico tipo di memoria a cui ha accesso, e il fatto che c'è una prefissa in più significa che ci sono almeno due prefisse diverse che portano nello stesso stato.

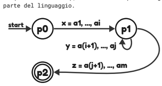
Formalmente:
∃ due prefisse 0^i e 0^j, con i =/= j, che portano allo stesso stato, quindi sono indistinguibili. Allora A accetta le stringhe 0^i 1^j e 0^j 1^i, anche se non hanno lo stesso numero di 0 e 1.

Il pumping lemma (teorema 4.1 a p. 120)

Sia L un linguaggio regolare.
∃ una costante n (dipendente da L) tale che ∀ w ∈ L con |w| ≥ n, w può essere scomposta come w = xyz, con i seguenti requisiti:

1. y =/= ε
2. |xy| ≤ n
3. ∀ k ≥ 0, anche x y^k z ∈ L

Quindi, tradotto:
Dato un linguaggio regolare, io posso trovare una costante n, dipendente dal linguaggio, tale per cui se considero una qualsiasi stringa del linguaggio lunga n, essa può essere spezzata in tre parti (xyz). La parte in mezzo (y) non può essere vuota, le prime due parti non possono essere troppo lunghe.
Se riesco a rispettare queste specifiche, non solo w = xyz, possiamo "iniettare" quante y vogliamo nella parte centrale (anche 0: xz ∈ L), e ottenere sempre stringhe che fanno parte del linguaggio.



Dimostrazione formale pumping lemma

Disponibile nella lezione

24 nov - 1

Esercizio 1 pumping lemma

Problema
Dimostrare che L01 = {0^n 1^n | n ≥ 1} **non** è regolare.

Per assurdo
Supponiamo che L01 sia regolare: allora vale il pumping lemma.
Chiamiamo n ∈ N la costante del PL.
Sia w = 0^n 1^n ∈ L01.
Scriviamo w = xyz tale che:

- x = 0^(n-1)
- y = 0
- z = 1^n

In questo modo rispettiamo i requisiti del pumping lemma:

- y =/= ε
- x e y non sono troppo lunghe (lunghezza ≤ n)

E quindi deve valere anche la terza parte:
x y^k z ∈ L01 ∀ k ≥ 0

Peccato che sia **falso**. Ad esempio, per k = 0, dovrebbe valere:
xz ∈ L01, con xz = 0^(n-1) 1^n --> ha un numero diverso di 0 e 1.

Esercizio 2 pumping lemma

Esempio 4.2 a p. 121
Dimostrare che Leq = {w ∈ {0,1}* | w contiene lo stesso numero di 0 e 1, senza vincoli sull'ordine} **non** è regolare.

Per assurdo
Supponiamo che Leq sia regolare: allora vale il pumping lemma.
Chiamiamo n ∈ N la costante del PL.

Come stringa w possiamo prendere qualunque stringa, che mi aiuta a fare la dimostrazione: usiamo la stessa stringa di prima.

Sia w = 0^n 1^n ∈ Leq

Scriviamo w = xyz tale che:
x = 0^(n-1)   y = 0   z = 1^n

E, per il PL, deve valere:
x y^k z ∈ L01 ∀ k ≥ 0

Invece è **falso**. Ad esempio, per k = 0, dovrebbe valere:
xz ∈ L01, con xz = 0^(n-1) 1^n ∉ Leq -> ha un n. diverso di 0 e 1.

Esercizio 3 pumping lemma

Esempio 4.3 a p. 122
Dimostrare che Lpr = {1^p | p primo} **non** è regolare.

Per assurdo
Supponiamo che Lpr sia regolare, e sia n ∈ N la costante del PL.

Sia w = 1^p ∈ Lpr tale che |w| ≥ n

Scriviamo w = xyz, con:
x = 1^(n-1)   y = 1   z = 1^(p-n)

E, per il PL, deve valere:
x y^k z ∈ Lpr ∀ k ≥ 0

Invece basta scegliere k tale che:
(n-1)+k+(p-n) non sia primo.

Esempio: k = 0, p = 5, n = 2
(2-1) + 0 + (5-2) = 1 + 3 = 4

Perciò Lpr **non** è regolare.

Esercizio 4 pumping lemma

Esempio 4.3 a p. 122
Dimostrare che Lpal = {w ∈ {0,1}* | w = w^R} (insieme delle stringhe palindrome composte da 0 e 1) **non** è regolare.

Per assurdo
Supponiamo che Lpal sia regolare, e sia n ∈ N la costante del PL.

```
Sia w = 0^n 1 0^n ∈ Lpal
        --- -----
        xy    z
```

Scriviamo w = xyz, con:
x = 0^(n-1)   y = 0   z = 1 0^n

Allora, per il PL, dovrebbe valere:
x y^k z ∈ Lpal ∀ k ≥ 0

Invece, xz (k=0) ∉ Lpal, poiché xz = 0^(n-1) 1 0^n
(numero di 0 diversi).

Perciò Lpal **non** è regolare.

Esercizio 5 pumping lemma

Parentesi bilanciate
Dimostrare che Lbal = {w ∈ {(,)}* | w è una stringa di parentesi bilanciate} **non** è regolare.

Per assurdo
Supponiamo che Lbal sia regolare, e sia n ∈ N la costante del PL.

Scegliamo w. Scegliere una **w = (()())** non aiuta, ma invece sceglierne **w = (^n)^n** ci rende tutto più semplice, perché è praticamente il linguaggio **0^n 1^n** ma con parentesi.

Scriviamo w = xyz, con:
x = (^[n-1]   y = (z =)^n

Allora, per il PL, dovrebbe valere:
x y^k z ∈ Lbal ∀ k ≥ 0

Invece, xz (k=0) ∉ Lbal, poiché xz = (^[n-1])^n
(numero di parentesi aperte e chiuse diverso).

Perciò Lbal **non** è regolare.

24 nov - 2

Esercizio 6 pumping lemma

Linguaggio palindromo
Dimostrare che L = {0^n 1 0^n} **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n 1 0^n ∈ L.

Scriviamo w = xyz, con:
x = 0^(n-1)   y = 0   z = 1 0^n

Possiamo notare che questo linguaggio è un sottolinguaggio di quello palindromo dell'esercizio 4.

Allora, per il PL, dovrebbe valere:
x y^k z ∈ L ∀ k ≥ 0

Invece, xz (k=0) ∉ L, poiché xz = 0^(n-1) 1 0^n
(il numero di 0 prima e dopo l'1 è diverso).

Perciò L **non** è regolare.

Esercizio 7 pumping lemma

Linguaggio con n e m
Dimostrare che L = {0^n 1^n 2^n | n, m } **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n 1^m 2^n ∈ L.

Scriviamo w = xyz, con:
x = 0^(n-1)   y = 0   z = 1^m 2^n

Allora, per il PL, dovrebbe valere:
x y^k z ∈ L ∀ k ≥ 0

Invece, xz (k=0) ∉ L, poiché xz = 0^(n-1) 1^m 2^n
(il numero di 0 e di 2 è diverso).

Perciò L **non** è regolare.

Esercizio 8 pumping lemma

Linguaggio con numero di 1 doppio rispetto a 0
Dimostrare che L = {0^n 1^2n | n ≥ 1} **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n 1^2n ∈ L.

Scriviamo w = xyz, con:
x = 0^(n-1)   y = 0   z = 1^2n

Allora, per il PL, dovrebbe valere:
x y^k z ∈ L ∀ k ≥ 0

Invece, xz (k=0) ∉ L, poiché xz = 0^(n-1) 1^2n
(c'è uno 0 in meno del dovuto).

Perciò L **non** è regolare.

Esercizio 9 pumping lemma

Linguaggio con meno 0 che 1
Dimostrare che L = {0^n 1^m | n ≤ m} **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n 1^m ∈ L (con n ≤ m).

Scriviamo w = xyz, con:
x = 0^(n-1)   y = 0   z = 1^m

Allora, per il PL, dovrebbe valere:
x y^k z ∈ L ∀ k ≥ 0

In questo caso, xz ∈ L, perché togliendo uno 0, n ≤ m resta valido.
Tuttavia, basta scegliere k tale che n-1+k > m --> k > m-n+1

Perciò L **non** è regolare.

Esercizio 10 pumping lemma

Linguaggio con più 0 che 1
Dimostrare che L = {0^n 1^m | n ≥ m} **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n 1^m ∈ L (con n ≥ m).

Scriviamo w = xyz, con:
x = 0   y = 0^(n-1)   z = 1^m

Allora, per il PL, dovrebbe valere:
x y^k z ∈ L ∀ k ≥ 0

Invece, xz (k=0) ∉ L, poiché xz = 0 1^m
(ci sono meno 0 che 1).

E se m fosse = 1?

Scriviamo w = xyz, con:
x = ε   y = 0^(n-1)   z = 1^m

xz (k=0) ∉ L, poiché xz = 1^m. Perciò L **non** è regolare.

Esercizio 11 pumping lemma

Linguaggio con n quadrato perfetto
Dimostrare che L = {0^n | n è un quadrato perfetto} **non** è regolare.

Per assurdo
Supponiamo che L sia regolare, e sia n ∈ N la costante del PL.

w = 0^n ∈ L.

Scriviamo w = xyz, con:
x = 0^(n-l)   y = 0^l   z = ε   con l > 0 (così y =/= ε)

Allora, per il PL, dovrebbe valere:
x y^k z = 0^(n-l) 0^kl ε = 0^(n-l+kl)

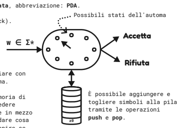
Scelgo k tale che 0^(n-l+kl) **non** è un quadrato perfetto.

Perciò L **non** è regolare.

Gli automi a pila

In inglese: **pushdown automata**, abbreviazione: **PDA**.

È un ε-NFA + una pila (stack).



La parte interessante della pila sta nel fatto che, contrariamente al numero di stati, la pila non ha una dimensione definita, e può quindi variare con il funzionamento dell'automa.

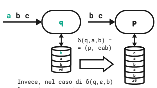
È possibile aggiungere e togliere simboli alla pila tramite le operazioni **push** e **pop**.

Tuttavia, non è come la memoria di un computer: non posso accedere direttamente a un carattere in mezzo alla pila. Posso solo guardare cosa c'è in cima alla pila, e capire se la pila è vuota oppure no.

L'automa deve sempre avere una pila non vuota, altrimenti si blocca: per questo useremo un carattere speciale, z0, che indica che sotto di lui non c'è più nulla.

Se la pila contiene solo z0 è **logicamente vuota**. Se invece non contiene nulla, è **fisicamente vuota** e l'automa si blocca.

Osserviamo il funzionamento del PDA.

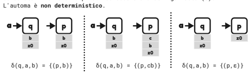


Invece, nel caso di δ(q,ε,b) la stringa non viene toccata, e il resto procede uguale a prima.

Definizione formale automa a pila

Un automa a pila (PDA) è una **settupla P = (Q, Σ, Γ, δ, q0, z0, F)** dove:
- Q è un insieme finito e non vuoto di stati
- Σ è l'alfabeto dei simboli di input
- Γ è l'alfabeto dei simboli di stack
- q0 ∈ Q è lo stato iniziale
- z0 ∈ Γ è il simbolo inizialmente presente nello stack
- F ⊆ Q è l'insieme degli stati finali
- δ è la funzione di transizione: Q × (Σ∪{ε}) × Γ -> P(Q × Γ*)

δ ha bisogno di 3 argomenti per funzionare: per questo, se la pila diventa fisicamente vuota, l'automa si blocca.
ε indica l'ε-mossa, e Γ* il la stringa che va a sostituire il simbolo consumato in cima alla pila. Può essere una stringa vuota (ε).
L'automa è **non deterministico**.



Posso farlo: ε ∈ Γ*

Osservazione: posso togliere (**pop**) dallo stack solo un simbolo alla volta, ma posso aggiungerne (**push**) quanti ne voglio.

Essendo non deterministico, la funzione δ di un automa a pila può essere così:

δ(q,a,c) = {(p,b),(r,ac)}
δ(q,ε,c) = {(s,c)}

30 nov - 1

PDA 1: stringhe binarie palindrome di lunghezza pari

Lwwr = {w w^R | w ∈ {0,1}* }

(Stringhe binarie palindrome di lunghezza pari)
Sono generate da questa CFG:
- **S -> 0S0 | 1S1 | ε**

Esempio: w = w^R -> 011 110 ∈ Lwwr

L'automa parte dallo stato iniziale.
Ci sono due fasi di computazione: una in cui legge w, e uno in cui legge w^R.

L'automa non sa se sta leggendo w o r^R, per cui, grazie al suo non-determinismo, ha un oracolo che suggerisce sempre la mossa giusta, o intraprende entrambe le strade a ogni passo di computazione.
Nel secondo caso, l'automa si incarterà molte volte, ma se almeno una serie di mosse porta a uno stato finale, la stringa è accettata.

Se si arriva a z0, la stringa è **accettata**.

L'automa **spinge** sulla pila i caratteri che legge per memorizzarli.

L'automa "**indovina**" grazie al **non-determinismo** che si trova nella **prima metà** della stringa.

L'automa inizia a eliminare gli elementi della pila corrispondenti alla **seconda metà** della stringa.

La computazione è composta da **tre fasi**:
- una in cui l'automa **riempie** la pila;
- una in cui l'automa **svuota** la pila;
- una in cui l'automa **accetta** la stringa.

Per questo l'automa avrà tre stati.

P = (Q, Σ, Γ, δ, q0, z0, F)
(Q: stati; Σ: alfabeto input; Γ: alfabeto stack; δ: funzione di transizione; q0: stato iniziale; z0: simbolo iniziale stack; F: stati finali)

P = ({q0,q1,q2},{0,1},{0,1,z0},δ,q0,z0,{q2})

Funzione δ:

δ(q0,0,z0) = {(q0,0 z0)}
δ(q0,1,z0) = {(q0,1 z0)}

Inizio della computazione: la **pila è vuota**, e leggo o uno 0 o un 1. Lo stato resta inalterato.

δ(q0,0,0) = {(q0,00)}
δ(q0,0,1) = {(q0,01)}
δ(q0,1,0) = {(q0,10)}
δ(q0,1,1) = {(q0,11)}

La **pila non è vuota**. Sono nella prima metà della stringa, e per questo resto in q0. **Rimpiazzo** il carattere della pila consumato con se stesso + quello della stringa consumato.

Riempimento pila

δ(q0,ε,z0) = {(q1,z0)}
δ(q0,ε,0) = {(q1,0)}
δ(q0,ε,1) = {(q1,1)}

ε-mosse: la prima serve in caso **l'intera stringa sia vuota**.* La seconda e terza sono per passare alla **seconda fase della computazione**, in cui si tolgono elementi dallo stack.

Transizione stati

δ(q1,0,0) = {(q1,ε)}
δ(q1,1,1) = {(q1,ε)}

Ci sono **solo due casi** per la seconda fase: se il primo elemento dello stack e l'ultimo della stringa **sono diversi**, l'automa **si blocca** e la stringa **non è accettata**.

Svuotamento pila

δ(q1,ε,z0) = {(q2,z0)}

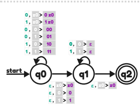
Se l'automa arriva **in fondo alla pila** (z0), la stringa è **accettata**.

Accettazione

* La **stringa vuota appartiene** al linguaggio: ε ∈ Lwwr.
L'automa passa subito da q0 a q1 con l'epsilon-mossa δ(q0,ε,z0) = {(q1,z0)}, in quanto è l'unica mossa disponibile.
Infine, fa un'altra epsilon-mossa e passa a q2, con δ(q1,ε,z0) = {(q2,z0)}. La stringa è accettata.

30 nov - 2

PDA 1: schema grafico



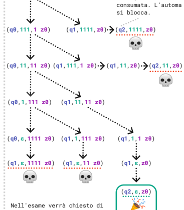
δ(q0,1,1) = {(q0,11)}
Si indica con un cappio (q0 -> q0), un 1, che indica il carattere della stringa consumata, e un 1/11, carattere consumato dello stack / caratteri che lo rimpiazzano (qui / è sostituito con > per rendere lo schema più chiaro).

Albero di computazione

```
input = 11 11     ∈ Lwwr
        w  w^R
```

L'automa parte dalla configurazione iniziale.
(q0,1111,z0)
Da qui posso applicare 1,z0/1 z0, oppure ignorare l'input e applicare ε,z0/z0.

È in q2, ma **non** accetta: la stringa non è stata consumata. L'automa si blocca.



Nell'esame verrà chiesto di seguire solo il ramo che porta allo stato accettante:

(q0,1111,z0) ⊢ (q0,111,1 z0) ⊢ (q0,11,11 z0) ⊢ (q1,11,11 z0) ⊢ (q1,1,1 z0) ⊢ (q1,ε,z0) ⊢ (q2,ε,z0)

Descrizioni istantanee

Le descrizioni istantanee (**ID**) sono **snapshot** che ci dicono com'è un PDA in un dato momento.

Definizione di ID
Una ID (o configurazione) è una tripla (q,w,γ), dove
- q ∈ Q è lo stato attuale
- w ∈ Σ* è l'input residuo (ciò che resta da leggere)
- γ ∈ Γ* è il contenuto attuale della pila.

Definizione di mossa
Sia **P = (Q, Σ, Γ, δ, q0, z0, F)** un PDA.
Supponiamo che la coppia (p,α) ∈ δ(q,a,x).
L'automa cambia stato in p, cancella a dalla pila e la sostituisce con α.
Allora,
∀ w ∈ Σ* e ∀ β ∈ Γ*, (q,aw,xβ) **⊢p** (p,w,αβ)

a è il primo carattere della stringa,
x è il primo carattere della pila.

⊢p indica una **mossa** del PDA **P**.
Se è evidente che ci riferiamo a P, scriviamo ⊢ anziché **⊢p**.
⊢p è una relazione binaria tra coppie di configurazione (ID).

Definizione di passaggio con 0 o + mosse (⊢*)
Base:
  I ⊢* I   ∀ ID I
Passo induttivo:
  I ⊢* J se ∃ ID K tale che **I ⊢ K e K ⊢* J**

Frammento/porzione di computazione

Si può indicare con ⊢p* per specificare il PDA.

Definizione non induttiva
al minuto 29 della lezione.

30 nov - 3

Teoremi 6.5 e 6.6 a pagina 215

Teorema 6.5

Sia P un PDA:

$P = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\text{stati finali}}{F})$

e $(q,x,\alpha) \vdash_P^* (p,y,\beta)$, allora $\forall\ w \in \Sigma^*$ e $\forall\ \gamma \in \Gamma^*$ vale:

$(q,xw,\alpha\gamma) \vdash_P^* (p,yw,\beta\gamma)$

Osservazione: noi sappiamo già che è possibile andare da (q,x,α) a (p,y,β) in 0 o più passi. Tuttavia, possiamo aggiungere w in fondo alla stringa, e il frammento di computazione rimane valido. La stessa cosa avviene aggiungendo γ in fondo alla pila: non vedendo oltre al primo carattere, la computazione non cambia.

Al contrario, se supponiamo di avere
$(q,xw,\alpha\gamma) \vdash_P^* (p,yw,\beta\gamma)$:
dire che $(q,x,\alpha) \vdash_P^* (p,y,\beta)$ è **falso**.

Il concetto non funziona con la pila: potremmo trovarci costretti a rimuovere α, e "contaminare" γ.
Invece, funziona anche al contrario con la stringa in input, per cui:

Teorema 6.6

Sia P un PDA:

$P = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\text{stati finali}}{F})$

e $(q,xw,\alpha) \vdash_P^* (p,yw,\beta)$, allora:

$(q,x,\alpha) \vdash_P^* (p,y,\beta)$

Linguaggi accettati dai PDA

Accettazione per stati finali

Sia P un PDA:

$P = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\text{stati finali}}{F})$

Il linguaggio accettato da P per stati finali è
$L(P) = \{w \in \Sigma^* \mid (q_0,w,z_0) \vdash_P^* (q,\varepsilon,\alpha)\}$
con $q \in F$ e $\alpha \in \Gamma^*$

Accettazione per pila vuota

Sia P un PDA:

$P = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\cancel{\text{stati finali}}}{\cancel{F}})$

Il linguaggio accettato per pila vuota è
$N(P) = \{w \in \Sigma^* \mid (q_0,w,z_0) \vdash_P^* (q,\varepsilon,\varepsilon)\}$
con $q \in Q$.

Osservazione 1: se si decide di accettare per pila vuota, F non serve a nulla: la nostra settupla diventa una sestupla.

Osservazione 2: si può convertire qualsiasi DFA che accetta per stati finali in DFA che accetta per pila vuota. Questo significa gli automi per pila vuota sono almeno potenti quanto gli automi per stati finali.
Inoltre, gli automi per stati finali sono potenti almeno quanto per pila vuota, perciò accettano gli stessi linguaggi.

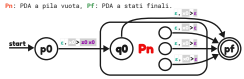
Teorema 6.9 a pagina 218

Da pila vuota a stati finali

Se $L = N(P_n)$ per un PDA: $P_n = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta_n}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0})$

allora esiste un PDA Pf tale che $L = L(P_f)$
$P_f = (Q\cup\{p_0,p_f\},\Sigma,\Gamma\cup\{x_0\},\delta_f,p_0,x_0,\{p_f\})$

Pn: PDA a pila vuota, Pf: PDA a stati finali.



Il rettangolo grosso è Pn, che noi dovremo andare a modificare per renderlo un automa a stati finali (Pf).

Andiamo ad aggiungere uno stato p_0, che inserisce in fondo alla pila x_0, nuovo e sconosciuto a Pn, che quindi non potrà toglierlo in alcun modo.

Colleghiamo ogni stato di Pn, compreso quello iniziale, allo stato finale di Pf, e sostituiamo x_0 con ε (o ancora x_0, non importa) per svuotare fisicamente la pila di Pn, ma solo logicamente quella di Pf (c'è ancora z_0).

Pf si accorgerà che Pn ha finito la computazione, perché Pn eliminerà z_0, che è il suo simbolo di fine pila. La pila è vuota per Pn, ma non Pf: Pf si sposterà in uno stato accettante.

δ_f è uguale a δ_n, con l'aggiunta di $\delta_f(p_0,\varepsilon,x_0) = \{(q_0,z_0 x_0)\}$,
e tutti i vari $\delta_f(q^*,\varepsilon,x_0) = \{(p_f,\varepsilon)\}$

1 dic - 1

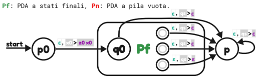
Teorema 6.11 a pagina 222

Da stati finali a pila vuota

Se $L = N(P_f)$ per un PDA: $P_f = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta_f}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\text{stati finali}}{F})$

allora esiste un PDA Pn tale che $L = L(P_n)$
$P_n = (Q\cup\{p_0,p\},\Sigma,\Gamma\cup\{x_0\},\delta_n,p_0,x_0)$

Pf: PDA a stati finali, Pn: PDA a pila vuota.



qs = qualsiasi: $\forall$ simbolo di $\Gamma\cup\{0\}$.

Il rettangolo grosso è Pf, che noi dovremo andare a modificare per renderlo un automa a pila vuota (Pn).

Andiamo ad aggiungere uno stato p_0, che inserisce in fondo alla pila x_0, nuovo e sconosciuto a Pf, che quindi non potrà toglierlo in alcun modo.

Colleghiamo ogni stato di Pf, che ora non sarà più finale, al nuovo stato p di Pn, e sostituiamo **qualsiasi elemento della pila** con ε per svuotare fisicamente la pila di Pn e terminare la computazione.

Che senso ha aggiungere x_0 se poi la pila verrà svuotata ugualmente?
Il senso è che Pf, durante la sua computazione, potrebbe svuotare la pila senza terminare, ma ciò porterebbe Pn a terminare prematuramente.
x_0 serve ad assicurarsi che la pila abbia sempre almeno un elemento, fino alla fine della computazione.

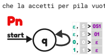
δ_n è uguale a δ_f, con l'aggiunta di $\delta_n(p_0,\varepsilon,x_0) = \{(q_0,z_0 x_0)\}$,
e tutti i vari $\delta_n(q^*,\varepsilon,qs) = \{(p,\varepsilon)\}$

Da CFG a PDA: introduzione

Sezione 6.3.1 p.225

Supponiamo di avere la CFG:
- **S -> 0S1|01**

Costruiamo un PDA che la accetti per pila vuota.



Non ci sono stati finali perché funziona a pila vuota, e il simbolo in fondo alla pila non è z_0, x_0... ma S, cioè il simbolo che rappresenta lo start symbol della nostra grammatica. Dopodiché, per ogni variabile Pn fa un'ε-mossa, toglie S dalla pila, e spinge sulla pila il contenuto del corpo della regola:

ε,S/0S1
ε,S/01

Lo fa per ogni regola di produzione, e per ogni variabile.
Infine, riguarda i simboli terminali e va a vedere se l'input matcha con il primo elemento della pila; se sì, lo rimuove:

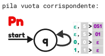
0,0/ε
1,0/ε

1 dic - 2

Da CFG a PDA: esempio

Continuiamo a lavorare sulla CFG:
- S -> 0S1|01

Convertita nel PDA per pila vuota corrispondente:



La procedura consiste nell'applicare le **regole di produzione** della CFG, rimuovendo la **testa** (lato sinistro della regola) dalla **pila** e sostituendola con il **corpo** (lato destro).

Legenda
- ε-mosse (⊢)
- modifiche avvenute
- <u>simboli da rimuovere</u>

Data la stringa 000111, seguiamo il ramo dell'albero di computazione che porta all'accettazione della stringa.

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$

Con una **ε-mossa sostituiamo S con 0S1**, come descritto dalla regola.

$(q,000111,\underline{S}) \vdash (q,\underline{0}00111,\boxed{\underline{0}S1}) \vdash$

Adesso però in cima alla pila non c'è più una variabile, ma un simbolo terminale (0), perciò non possiamo più applicare regole.

Quindi cosa fa l'automa? Va a vedere se il simbolo all'inizio della stringa ($\underline{0}$) matcha il simbolo in cima alla pila ($\underline{0}$). Se sì, **li cancella entrambi**.

$(q,00111,\underline{S}1) \vdash$

Adesso S è in cima alla pila, e possiamo **applicare la regola**.

$(q,\underline{0}0111,\boxed{\underline{0}S1}1) \vdash$

$(q,0111,\underline{S}11) \vdash (q,\underline{0}111,\boxed{\underline{0}1}11) \vdash (q,\underline{1}11,\underline{1}11) \vdash (q,\underline{1}1,\underline{1}1) \vdash (q,\underline{1},\underline{1}) \vdash (q,\boxed{\varepsilon},\boxed{\varepsilon})$

Continuiamo con questa procedura, finché non ci troviamo **solo con simboli terminali** da controllare ed eliminare.

La pila si svuota: dunque il PDA **accetta** la stringa 000111.

Da CFG a PDA: teorema

Data la grammatica $G = (\underset{\text{variabili}}{V}, \overset{\text{simboli terminali}}{T}, \underset{\text{regole di produzione}}{Q}, \overset{\text{start symbol}}{S})$, costruiamo il PDA $P = (\underset{\text{stato}}{\{q\}}, \overset{\text{alfabeto input}}{T}, \underset{\text{alfabeto stack}}{V\cup T}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q}, \overset{\text{simbolo iniziale stack}}{S})$.

La funzione δ ha due casi:
1. $\forall\ A \in V$, $\delta(q,\varepsilon,A) = \{(q,\beta) \mid A \to \beta$ è una produzione di $G\}$ ····> Nell'esempio di prima: ε, S > 0S1; ε, S > 01
2. $\forall\ a \in T$, $\delta(q,\varepsilon,A) = \{(q,\beta)\}$ ····> Nell'esempio di prima: 0, 0 > ε; 1, 1 > ε

I PDA accettano tutti i linguaggi regolari?

Se $L \in REG$, allora $\exists$ PDA Pf (non-deterministico) tale che $L = L(P_f)$ (**accettato per stati finali da** Pn).
Inoltre, esiste anche un PDA Pn (non-deterministico) tale che $L = L(P_n)$.

Dimostrazione:
Infatti, un **PDA non-deterministico** è un **ε-NFA** che usa una **pila**, e gli ε-NFA accettano i linguaggi **regolari**. Se lasciamo la pila **sempre vuota** nella computazione, abbiamo un **semplice ε-NFA**.
Inoltre, i **PDA** per **pila vuota** accettano **gli stessi** linguaggi dei **PDA** per **stati finali**.

Negli automi deterministici, la situazione cambia.

Teorema 6.17 a pagina 234

Se $L \in REG$, $\exists$ un DPDA Pf tale che $L = L(P_f)$.

Dimostrazione:
DPDA per stati finali ≈ **DFA** + **pila**

Come prima, se **non uso la pila**, il controllo finito simula il **DPDA**, e accetta il linguaggio.

Osservazione:
Non è detto che $\exists$ **DPDA** Pn tale che $L = N(P_n)$ (**accettato per pila vuota da** Pn).

Quindi, per pila vuota i DPDA fanno **meno** di quelli per stati finali, e non fanno tutti i REG.

PDA deterministico (DPDA)

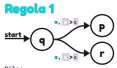
Condizioni affinché il PDA diventi deterministico

Definizione: Un PDA $P = (\underset{\text{stati}}{Q}, \overset{\text{alfabeto input}}{\Sigma}, \underset{\text{alfabeto stack}}{\Gamma}, \overset{\text{funzione di transizione}}{\delta}, \underset{\text{stato iniziale}}{q_0}, \overset{\text{simbolo iniziale stack}}{z_0}, \underset{\text{stati finali}}{F})$ è **deterministico** (**DPDA**, **Deterministic PushDown Automata**),

se valgono queste due condizioni.
1. $|\delta(q,a,x)| \le 1 \quad \forall\ q \in Q,\ \ \forall\ a \in \Sigma\cup\{\varepsilon\},\ \ \forall\ x \in \Gamma$ ← La cardinalità può essere 0: in quel caso, l'automa si incarta
2. Se $|\delta(q,a,x)| \ne 0$ per qualche $a \in \Sigma$, allora $|\delta(q,\varepsilon,x)| = 0$

Questi sono i casi di non-determinismo che le regole permettono di escludere:

Regola 1

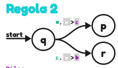


Pila:

x
c
z0

Funzione δ:
$\delta(q,a,x) = \{(p,\varepsilon)\}$ ✓
$\delta(q,a,x) = \{(r,\varepsilon)\}$ ✗

Regola 2



Pila:

x
c
z0

Funzione δ:
$\delta(q,a,x) = \{(p,\varepsilon)\}$ ✓
$\delta(q,a,x) = \{(r,b)\}$ ✗

Così facendo, il nostro automa è **deterministico**, e ad ogni passo può fare **una sola mossa**.
L'albero di computazione diventa **un ramo solo**.

Osservazione: ci possono essere delle **ε-transizioni** nei DPDA? Sì, basta che non creino una delle situazioni di non-determinismo sopra.

Questi automi **non** riconoscono tutti i CFG: con il determinisimo si **perde** qualcosa.
Tuttavia, dato che sono essenzialmente DFA + una **pila**, accettano tutti i linguaggi regolari.

Esempio 6.16 a pagina 234

Stringhe palindrome dispari formate da 0 e 1

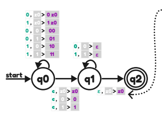
DPDA che accetta Lwcwr = {w c w^R | w ∈ {0,1}, con c ≠ 0,1}.

Attenzione
Lwcwr ≠ Lwwr (esempio delle stringhe palindrome **pari** del PDA del 30 nov - 1)

Lwwr è un CFL **non può essere accettato** da **DPDA**, mentre è **accettato** dai **PDA**.
Perché? Perché la **separazione** tra w e w^R **non è definita** da un carattere in particolare, e non si può sapere la lunghezza della stringa.
L'unico modo per capire dove inizia w^R è tramite il **non-determinismo** (**molteplici rami/oracolo**).

Con il simbolo c, che non è 0 o 1, aiutiamo il DPDA a capire quando è il momento di cambiare stato.

La notazione con il diagramma degli stati è identica, perché non abbiamo fatto altro che imporre dei vincoli per la funzione δ.



Se tolgo z_0 nell'ultimo passaggio, accetto sia per pila vuota che per stato finale. Altrimenti, accetto solo per stato finale.

14 dic - 1

Linguaggi prefix-free

Definizione: proprietà del prefisso (prefix-free)

Attenzione: avere la proprietà del prefisso vuol dire essere **senza** prefisse!

Un linguaggio L ha la proprietà del prefisso (cioè è **prefix-free**) se ∄ due stringhe x e y ∈ L tali che x ≠ y e x è prefissa di y.
Tutti i CFL sono prefix-free.

Quindi, se prendo una stringa y ∈ L, tutte le sue prefisse non devono appartenere al lingaggio. Quindi, se tolgo uno o più simboli dal fondo, non deve appartenere al linguaggio.

Prendiamo degli esempi.

1. **Lwcwr** è prefix-free? 01c1~~0~~ 000c00~~0~~

   Sì. Eliminando uno o più simboli dal fondo, la stringa non appartiene più a L: la c non è più in mezzo.

2. **L0** è prefix-free? L0 = **{0}*** = {ε,0,00,000,...}

   000~~0~~ Sia 000 che 0000 ∈ L0

   No. Eliminando uno o più simboli dal fondo, la stringa appartiene comunque a {0}*.

Osservazione: Per dimostrare che un linguaggio **non** è prefix-free, bastano due stringhe d'esempio. Invece, per dimostrare se **è** prefix-free, bisogna fare un ragionamento che si applica a tutte le stringhe.

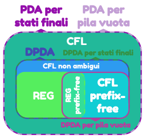
Teorema 6.19 pag. 235

Un linguaggio L è N(P) per un DPDA P **se e solo se** L ha la proprietà del prefisso (cioè è prefix-free) ed è L(P') per un DPDA P'.

Questo vuol dire che io prendo l'insieme di linguaggi accettanti per stati finali (L(P')) e pongo una restrizione: se il linguaggio è accettato dall'automa a **stati finali**, e inoltre è anche **prefix-free**, allora è accettato da un altro automa deterministico per **pila vuota**.

Viceversa (dato che c'è il "se e solo se"), se ho che un linguaggio è accettato da un DPDA per **pila vuota**, allora ho sicuramente un altro DPDA che lo accetta per **stati finali**, ma certamente il linguaggio è anche **prefix-free**.

Di conseguenza, i DPDA per **pila vuota** sono meno potenti, perché accettano una sottoclasse di quelli per **stati finali**: solo linguaggi **prefix-free**.



Questo schema illustra la potenza in termini di linguaggi riconosciuti dei vari tipi di PDA.

CFL non ambigui = non prefix-free / con prefissa

Dimostrazione

Supponiamo di avere un DPDA P per **pila vuota**.
Supponiamo che L = N(P).
Per assurdo, supponiamo che L **non** sia **prefix-free**.

Allora ∃ x,y ∈ L tali che x ≠ y e x è **prefissa** di y.

Pongo y = xw.

Se è vero che P accetta x, dopo aver consumato x, la pila è vuota.
P però **non** può accettare y: nel primo pezzo (x) la pila si svuota, ma ci sono ancora caratteri da leggere (w), e quindi si incarta.

Per questo, se un linguaggio ha delle prefisse, **non** può essere accettato per **pila vuota**.

Osservazione: Prima abbiamo detto che i DPDA per **stati finali** accettano tutti i lingaggi regolari, dato che sono semplicemente dei **DFA** + una **pila**.
Invece, i DPDA per **pila vuota** **NON** accettano tutti i linguaggi regolari.
Banalmente, il linguaggio L0 = **{0}*** = {ε,0,00,000,...} che abbiamo visto prima **non** è **prefix-free**, e quindi **non** è accettato.

14 dic - 2

CFG non ambigue

Teorema 6.20 a pagina 236

Se L ∈ N(P) per un DPDA P, allora L è una CFG **non ambigua**.

Teorema 6.21 a pagina 237

Se L = L(P) per un DPDA P, allora L ha una CFG **non ambigua**.

Esercizio 3

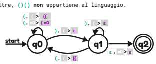
Scrivere un DPDA per L = linguaggio delle parentesi bilanciate (Lpal), generato dalla CFG:
- **B -> (BB)|(B)|()**

L è **prefix-free**? Sì: se y ∈ L, e x ≠ y è prefissa di y, allora x contiene delle parentesi aperte ma non chiuse.

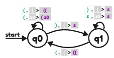
Definiamo 3 stati: q0, che impila parentesi aperte sullo stack; q1, che rimuove parentesi dallo stack ogni volta che ne legge una chiusa, e q2, che accetta quando la pila è vuota (funziona sia per L(P) che per N(P)).

Notare che è necessario poter tornare da q1 a q0 in caso ci siano altre parentesi aperte, come nel caso di (()()).

Inoltre, ()() **non** appartiene al linguaggio.



Solo per **pila vuota**:



Esercizio 1

Scrivere un DPDA per L = {0^n 1^n | n ≥ 1}.

Possiamo scegliere di fare una di queste due cose:

1. Utilizzare un DFA per stati finali, per essere sicuri di poterlo fare
2. Stabilire per prima cosa se è prefix-free, e poi scegliere il tipo di DFA.

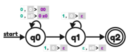
Facciamo la 2: **è prefix free?**

Sia y ∈ L, allora y = 0^n 1^n con n ≥ 1.
Se tolgo un 1 dal fondo della stringa, la stringa che ottengo non ha più lo stesso numero di 0 e 1, e quindi x ∉ L. **È** **prefix free**.

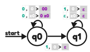
Quindi, volendo, possiamo fare un PDA per pila vuota.

L'automa funzionerà aggiungendo 0 alla pila ogni volta che li legge, e togliendoli man mano che legge 1. Se la pila è vuota e la stringa è consumata, la stringa è accettata; altrimenti, se la pila non si svuota o la stringa non è completamente consumata, la stringa non è accettata.

Il PDA seguente funziona sia per **stati finali** sia per **pila vuota**:



Questo invece funziona **solo per** **pila vuota**, ma ha uno stato in meno.



Esercizio 2

Scrivere un DPDA per L = {0^n 1^n | n ≥ 0}.

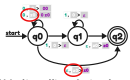
Rispetto al linguaggio di prima, c'è anche la stringa vuota.

Di conseguenza, L è **prefix-free**?

Prendiamo una stringa y ∈ L, con n > 0.
Allora x = ε è prefissa di y, x ≠ y, e x ∈ L.
Togliendo da una stringa tutti i caratteri, otteniamo ε ∈ L.

Quindi basta che un linguaggio contenga ε per renderlo non prefix-free.

L **non** è prefix-free, e quindi sarà necessario accettare per **stati finali**.



Bello questo DPDA eh? Semplice e efficace. Passiamo al prossimo esercizio...
SIKE! **È SBAGLIATO**! Infatti, da q0 partono due frecce, una epsilon mossa e una mossa normale, che dipendono dallo stesso carattere nella pila (z0). Questo infrange la seconda condizione che abbiamo imposto per i DPDA, perché aggiunge non-determinismo.
È possibile fare un DPDA per questo linguaggio? È difficile da dimostrare, ma la risposta è **no**. Si può solo costruire con PDA non-deterministici.

Esercizio 4

Scrivere un DPDA per L = linguaggio delle parentesi bilanciate, generato dalla CFG:
- **B -> BB|(B)|()**

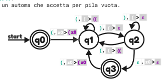
Non esistono parentesi che racchiudono BB.
(()()) **non** è più valida, ma ()() adesso lo è:

B -> BB -> ()B -> ()()

L è **prefix-free**? No (basta un controesempio):

() è prefissa di ()(), e ∈ L.
Quindi **non** esiste un automa che accetta per pila vuota.

Per **stati finali**:

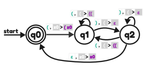


14 dic - 3

Esercizio 5

Scrivere un DPDA per L generato dalla CFG:
- **B -> (BB)|(B)|ε**

L è **prefix-free**? No: ε ∈ L.
Ad esempio, () è prefissa di ()(),
e ε è prefissa di entrambe le stringhe.



Esercizio 6: linguaggio nidificato

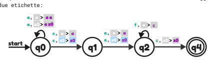
Scrivere un DPDA per L = { a^n b^m c d^m e f^n | n,m ≥ 0}

L è **prefix-free**? Sì: con n > 0, togliendo un simbolo dal fondo, la stringa non è accettata.
E se **n = 0**? Si possono verificare in totale 4 casi tra m e n.
Esaminiamo quali dono **prefix-free**:

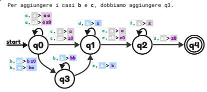
1. n > 0, m = 0 -> **a^n c e f^n** -> ✓
2. n = 0, m > 0 -> **b^m c d^m e** -> ✓
3. n > 0, m > 0 -> **a^n b^m c d^m e f^n** -> ✓
4. n = 0, m = 0 -> **c e** -> ✓

Volendo, si potrebbe accettare anche per pila vuota, ma faremo per **stati finali** per maggiore convenienza.

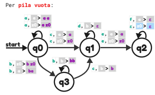
In questo modo funziona il caso **a**. Includiamo anche il caso **d** aggiungendo due etichette:



Adesso riconosce **a** e **d**.
Per aggiungere i casi **b** e **c**, dobbiamo aggiungere q3.



Per **pila vuota**:



Le macchine di Turing

Nel corso di una conferenza matematica che si è tenuta nell'anno 1900, Hilbert ha esposto la lista dei problemi da risolvere.

Uno in particolare chiedeva se era possibile stabilire qualsiasi teorema vero in maniera meccanica, anche se non esistevano i computer.

In quegli anni, esistevano dei computer meccanici usati nell'ambito della crittografia: vennero in pochi decenni migliorati e miniaturizzati sempre di più, fino a che nella Seconda Guerra Mondiale diedero origine alla macchina Enigma, un gioiello ingegneristico.

L'idea era: posso prendere una di queste macchine che nasceranno nei prossimi anni, e farle dimostrare tutti i problemi possibili? Ciò sostituirebbe i matematici.

Due persone lavorarono a questa cosa: **Gödel**, un logico, e **Turing**, un matematico.
Gödel dimostrò che non è possibile trovare una procedura automatica per dimostrare i teoremi veri, perché **la dimostrabilità e la verità non coincidono**.
Perciò ci sono degli enunciati che non è possibile né dimostrare né smentire: questo è il famoso **teorema di incompletezza di Gödel**.

Per la parte del problema riguardante la dimostrazione meccanica, Turing provò a creare un modello a partire da quello che fa un matematico umano per dimostrare un teorema.
Prende un foglio, pensa, legge, scrive.

Come posso automatizzare questa cosa?

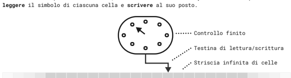
Bisogna prendere un meccanismo di controllo, in poche parole un **automa a stati finiti**, in cui gli stati rappresentano gli stati mentali del cervello del matematico.

Poi c'è il foglio: immaginiamolo come una striscia, divisa in celle, ciascuna contenente un simbolo di un alfabeto finito.

In base alla dimensione della tabella, potrò dimostrare un teorema di un certo calibro: se ho l'equivalente di un foglio, potrò dimostrare solo teoremi che stanno su un foglio.

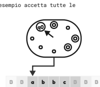
Quindi immaginiamo un **nastro composto da infinite celle**, da entrambi i lati.

Poi ci serve la penna: nel modello, assume la forma di un **dispositivo** che mi permette di **leggere** il simbolo di ciascuna cella e **scrivere** al suo posto.



In questo modo posso creare un modello di calcolo che ad esempio accetta tutte le stringhe di un linguaggio.

Prendiamo un esempio: abbiamo in input la stringa abbc.
Stavolta non diamo l'input direttamente alla macchina, ma lo scriviamo sul nastro.
La testina di lettura/scrittura si trova sul primo simbolo (a), e il controllo finito si trova in uno stato iniziale, q0.



Le altre celle sono vuote: in realtà sono solo logicamente vuote, perché contengono un simbolo speciale, solitamente segnato con **B** o con **ƀ**.

Inoltre, la macchina di Turing ha una un **alfabeto di input**, un **alfabeto per il nastro**, degli **stati finali**, e una funzione delta.

La funzione delta prende uno stato attuale, il simbolo sul nastro sotto la testina, e restituisce in modo deterministico * una tripla, che contiene un nuovo stato, un simbolo del nastro che va a sostituire quello sotto la testina, e L/R, che sposta la testina a sinistra (L) o a destra (R).

$$\delta(\underbrace{q}_{\text{stato di partenza}},\underbrace{a}_{\text{simbolo sul nastro}}) = (\underbrace{p}_{\text{stato di arrivo}},\underbrace{b}_{\text{nuovo simbolo}},\underbrace{L}_{\text{direzione della testina}})$$

*-> esistono macchine di Turing non-deterministiche

Quando si arresta? Quando delta non è definita per quel caso: ancora una volta è una funzione **parziale**.
Se lo stato in cui si ferma è finale, la stringa è **accettata**.
Se è **non finale**, **è rifiutata**.

Osservazione: la stringa di input non è consumata come negli altri automi, ma si può leggere e riscrivere quante volte si vuole.

Ad esempio, posso copiare la stringa da un'altra parte, e poi cancellarla.
Si possono cominciare a fare procedure che ricordano sempre di più dei programmi.

Con questo modello, Turing va al nucleo di quello che vuol dire "computazione", e dimostra che è impossibile avere una procedura automatizzata per prendere in input un programma della macchina di Turing e capire se si fermerà oppure no.
Questo è il cosiddetto **halting problem** (problema dell'arresto), ed è **indecidibile**, **non calcolabile**.

Siamo sicuri che questa sia la risposta definitiva? Turing ha scelto il modello giusto?
Sì: un altro logico, Alonzo Church, creò il "lambda calcolo" per simulare delle computazioni, e scoprì che le cose calcolabili con il lambda calcolo sono le stesse calcolabili con le macchine di Turing. Emil Post scoprì la stessa cosa con il suo modello.

La risposta alla domanda della conferenza è no, i matematici non saranno sostituiti da macchine. Accidenti. *Sad Wolfram Alpha noises*

15 dic - 1

Definizione della macchina di Turing

Una macchina di Turing (MdT) è una **settupla** M = (Q, Σ, Γ, δ, q0, B, F) dove:

(Q: stati; Σ: alfabeto input; Γ: alfabeto nastro; δ: funzione di transizione; q0: stato iniziale; B: simbolo di blank; F: stati finali)

- Q è un insieme finito e non vuoto di stati
- Σ è l'alfabeto (finito) dei simboli di input (Σ ⊂ Γ)
- Γ è l'alfabeto dei simboli del nastro
- δ è la funzione parziale di transizione: Q × Γ -> Q × Γ × {L,R}
- q0 ∈ Q è lo stato iniziale
- B ∈ Γ è il simbolo di blank (B ∈ Γ \ Σ)
- F ⊆ Q è l'insieme degli stati finali

Facciamo un esempio di mossa.

w = abcc   δ(q0,a) = (p,b,L)

L(M) = {a ∈ Σ* | M accetta a}

Un linguaggio L è **ricorsivamente enumerabile** se esiste una macchina di Turing M tale che L = L(M).

Come si pongono questi linguaggi nella gerarchia di Chomsky?

- Tipo 0: **Ricorsivamente enumerabili**
- Tipo 1: Contestuali
- Tipo 2: Context-free
- Tipo 3: Regolari

Sono generati da una grammatica che fondamentalmente **non ha vincoli**: nella testa e nella coda delle regole di produzioni può esserci qualsiasi sequenza di **simboli terminali** e **non terminali**, senza restrizioni.

Abbiamo detto che la macchina di Turing sembra essere il modello universale di calcolo: infatti Alonzo Church disse che **ogni linguaggio di calcolo** sensato è **Turing-computabile**.

Infatti è quello che è successo finora, con un'unica eccezione notevole: quella dei **modelli di calcolo quantistici**, perché sono in grado di produrre bit **casuali**. Posso preparare un qubit in una sovrapposizione di 0 e 1 in misura equipesata, e quando vado ad effettuare una misurazione, cade nello stato classico 0 o nello stato classico 1 con esattamente la stessa probabilità.

Le descrizioni istantanee (ID)

Disegnare la macchina di Turing ogni volta è scomodo: come negli automi a pila, possiamo rappresentarlo in modo da avere tutte le informazioni sulla MdT in quell'esatto istante.

ID: x1 x2 ... xi-1 q xi xi+1 ... xn

Per evitare ambiguità, si richiede che Q ∩ Γ = Ø (nessun simbolo del nastro può essere interpretato come stato del controllo finito).

⊦ = relazione binaria tra ID
I ⊦ J se e solo se c'è un caso della δ che consente di farlo.

Esempio 1
Supponiamo che δ(q,xi) = (p,y,L). Allora:

x1 x2 ... xi-1 q xi xi+1 ... xn ⊦ x1 x2 ... xi-2 p xi-1 y xi+1 ... xn

Abbiamo due casi particolari da considerare:

1. i = 1 (la testina punta sul primo simbolo non blank a sinistra)

   q x1 x2 ... xn ⊦ p B x1 x2 ... xn

2. i = n e y = B (la testina punta sull'ultimo simbolo non blank a destra, e sostituisce un carattere con blank)

   x1 x2 ... xn-1 q xn ⊦ x1 x2 ... xn-2 p xn-1

Il blank viene "inglobato" dai blank successivi, non c'è bisogno di scriverlo.

Esempio 2: casi speculari
Supponiamo che δ(q,xi) = (p,y,R). Allora:

x1 x2 ... xi-1 q xi xi+1 ... xn ⊦ x1 x2 ... xi-2 p xi-1 y xi+1 ... xn

1. i = n:
   x1 x2 ... xn-1 q xn ⊦ x1 x2 ... xn-1 y p B

2. i = 1 e y = B:
   q x1 x2 ... xn ⊦ p x2 ... xn

Esempio di computazione

Esempio 8.2 a pagina 303
Rappresenta la MdT che accetta **L = {0^n 1^n | n ≥ 1}**.

Funzione δ

!	Q Γ	0	1	X	Y	B
->	q0	(q1,X,R)	-	-	(q3,Y,R)	-
	q1	(q1,0,R)	(q2,Y,L)	-	(q1,Y,R)	-
	q2	(q2,0,L)	-	(q0,X,R)	(q2,Y,L)	-
	q3	-	-	-	(q3,Y,R)	(q4,B,R)
*	q4	-	-	-	-	-

Esempio di computazione con stringa accettante

Prendwiamo una stringa che fa parte del linguaggio: w = 0011

Attenzione: qui q4 è lo stato accettante, ma se non è indicato nessuno stato accettante, nell'esame bisogna solo mostrare la computazione, perché non conosciamo il linguaggio.

In generale, la configurazione iniziale, data una stringa w, è q0 w.
In questo caso è q0 0011.

Ecco la computazione.

q0 0011	⊦	X q1 011	⊦	X0 q1 11	⊦	X q2 0Y1	⊦	q2 X0Y1
X q0 0Y1	⊦	XX q1 Y1	⊦	XXY q1 1	⊦	XX q2 YY	⊦	X q2 XYY
XX q0 YY	⊦	XXY q3 Y	⊦	XXYY q3 B	⊦	XXYYB q4 B	✓	

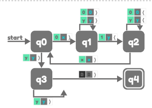
Esempio di computazione con stringa NON accettante

w = 0010

q0 0010	⊦	X q1 010	⊦	X0 q1 10	⊦	X q2 0Y0	⊦	q2 X0Y0
X q0 0Y0	⊦	XX q1 Y0	⊦	XXY q1 0	⊦	XXY0 q1 B	✗	

15 dic - 2

Diagramma degli stati



Definizione: linguaggi accettati

Come nei PDA, si può indicare una computazione in 0 o più passi utilizzando ⊦*

I ⊦* J

Definizione: linguaggio accettato da una macchina di Turing

Sia M = (Q, Σ, Γ, δ, q0, B, F) una MdT.

(Q: stati; Σ: alfabeto input; Γ: alfabeto nastro; δ: funzione di transizione; q0: stato iniziale; B: simbolo di blank; F: stati finali)

Allora il linguaggio accettato da M è

L(M) = { w ∈ Σ | q0 w ⊦* αpβ , con p ∈ F e α,β ∈ Γ* }

Osservazioni

Osservazione 1: calcolo di funzioni
Adesso stiamo utilizzando le MdT per accettare linguaggi,ma in realtà le MdT si possono anche utilizzare per calcolare funzioni.

fm: Γ* -> Γ*

Osservazione 2: accettazione
In realtà Turing ha definito l'accettazione **per arresto**: se la macchina si arresta, la stringa è accettata.
Questo perché lui era interessato a dimostrare che è impossibile determinare se la macchina si ferma, è un problema indecidibile.
Con il passare degli anni, si è preferito passare a una forma di accettazione con stati finali.

Varianti migliori?

Esistono delle varianti che possono fare un po' di più?

Testina ferma

A   δ: Q × Γ -> Q × Γ × {L,R}

Se io definisco un nuovo modello in cui:

B   δ: Q × Γ -> Q × Γ × {L,R,S}

con S = stay. La testina non è più obbligata a muoversi.
Questa macchina fa qualcosa in più? In realta, no.

Questo si dimostra in modo semplice, perché A può simulare B.

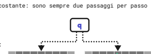
Ad esempio, proviamo a simulare che la macchina resti ferma in un passo. Allora posso mandarla in uno stato ausiliarlo nuovo, e la faccio spostare ad esempio a sinistra, e questo nuovo stato ausiliarlo la rimanda a destra.

In due passi è simulato il modello che sembrava più potente.

È più lento, ma costante: sono sempre due passaggi per passo fermo.

Più nastri

La delta sarebbe:



δ(q,a,b) = (p,b,L,c,R)

Tuttavia, può essere simulato dal modello base.

Perché uno potrebbe volere più nastri? Alcune computazioni potrebbero risultare più semplici, specialmente se si può assegnare uno specifico compito a ciascun nastro.

Macchina di Turing con solo bit

Questa a dire il vero non è proprio una estensione ma una restrizione.
I simboli sul nastro provengono da un alfabeto finito e non vuoto.
In realtà non cambia nulla in termini di potenza di calcolo (è solo rallentata) se ci limitiamo a usare 0 e 1 come simboli sul nastro, e usando 0 al posto del simbolo di blank.

Variante non-deterministica

Supponiamo di definire un nuovo modello in cui la delta è:

δ: Q × Γ -> P (Q×Γ×{L,R})

Per esempio:

δ(q,a) = {(p1,b1,L),(p2,b2,R),...,(pn,bn,L)}

Quindi in una macchina di Turing non-deterministica possiamo:

1. Avere un oracolo che prevede la mossa corretta;
2. Dividere il ramo dell'albero di computazione ogni volta.

Basta che esista almeno un ramo della computazione che venga accettato per dichiarare la stringa valida.

Il numero dei possibili rami di computazione è esponenziale, dove ad ogni livello corrisponde un passo di computazione, e dopo n passi di computazione abbiamo un numero esponenziale di mosse (ad esempio 2^n).

Questo modello, che sembra saper fare così tante cose,
è più potente della macchina di Turing deterministica?

La risposta è **NO**: fondamentalmente, esiste una macchina di Turing deterministica con due nastri che fa la stessa cosa. Utilizza il secondo nastro come una coda, e ogni volta che il ramo si divide accoda il nuovo ID, per poi estrarlo e continuare la computazione.
Fa praticamente la visita dell'albero di computazione in ampiezza (non in profondità: potrebbe finire in un ramo infinito). Il tempo sarebbe ovviamente maggiore: benvenuto nel modno dell'emulazione.

Sia MdT deterministica che non-deterministica riconoscono linguaggi ricorsivamente enumerabili.

Complessità computazionale

Ci sono due classi di complessità:

- P:
  Classe dei problemi di decisione che possono essere determinati da una macchina di Turing deterministica in tempo polinomiale.
  È un albero di computazione formato da un solo ramo, che ha lunghezza polinomiale p(n), dove n è la lunghezza del mio input.
- NP:
  È la stessa classe, ma con macchine non-deterministiche.

Sono uguali?
Dimostrare se   P ⊆ NP,   P ⊇ NP, oppure   P ⊂ NP
è un problema da un milione di dollari.

Questo è uno dei nuovi problemi del millennio, una lista di problemi come quella proposta da Hilbert nel 1900, da cui dipendono anche altri problemi.

Esercizio 13

Problema & logica

Problema

Costruite una macchina di Turing che accetti il linguaggio

L = {0^n 1^n 2^n con n ≥ 0}

Possibile soluzione

Questo è il mio ragionamento, e sarà probabilmente completamente sbagliato.
La computazione inizierà con una stringa di input scritta sul nastro. Ad esempio: **w** = 000111222.

In linea generale, cercherò di creare un automa che vada avanti e indietro nella stringa, cambiando 0 in X, 1 in Y e 2 in Z.

Verranno implementati gli stati:

- **qi**: stato **i**niziale; cambia 0 in X. Se trova Y va in **qy**
- **q1**: scorre avanti finché non trova il primo **1**, e lo cambia in Y
- **q2**: scorre avanti finché non trova il primo **2**, e lo cambia in Z
- **qx**: scorre indietro finché non trova una **X**, per poi tornare a qi
- **qc**: scorre a destra finché non trova un blank, e va in **qy**; si inceppa se trova qualunque cosa oltre a Y e Z (**c**heck)
- **qy**: stato accettante (**y**es)

Inizialmente, la testina sarà sicuramente posizionata su 0. Ogni altro valore non può essere accettato, con una eccezione: blank (n=0). In quel caso, si passa direttamente allo stato accettante e la computazione si ferma.

Funzione δ

!	Q Γ	0	1	2	X	Y	Z	B
->	qi	(q1,X,R)	-	-	-	(qy,Y,R)	-	-
	q1	(q1,0,R)	(q2,Y,R)	-	-	(q1,Y,R)	-	-
	q2	-	(q2,1,R)	(qx,Z,L)	-	-	(q2,Z,R)	(qc,B,L)
	qx	(qx,0,L)	(qx,1,L)	-	(qi,X,R)	(qx,Y,L)	(qx,Z,L)	-
	qc	-	-	-	(qc,X,L)	(qc,Y,L)	(qc,Z,L)	(qy,B,R)
*	qy	-	-	-	-	-	-	-

Computazione

qi 000111222	⊦	X q1 00111222	⊦	X0 q1 0111222	⊦	X00 q1 111222	⊦
X00Y q2 11222	⊦	X00Y1 q2 1222	⊦	X00Y11 q2 222	⊦	X00Y1 qx 1Z22	⊦
X00Y qx 11Z22	⊦	X00 qx Y11Z22	⊦	X0 qx 0Y11Z22	⊦	X qx 00Y11Z22	⊦
qx X00Y11Z22	⊦	X qi 00Y11Z22	⊦	XX q1 0Y11Z22	⊦	XX0 q1 Y11Z22	⊦
XX0Y q1 11Z22	⊦	XX0YY q2 1Z22	⊦	XX0YY1 q2 Z22	⊦	XX0YY1Z q2 22	⊦
XX0YY1 qx ZZ2	⊦*	X qx X0YY1ZZ2	⊦	XX qi 0YY1ZZ2	⊦	XXX q1 YY1ZZ2	⊦
XXXY q1 Y1ZZ2	⊦	XXXYY q1 1ZZ2	⊦	XXXYYY q2 ZZ2	⊦	XXXYYYZ q2 Z2	⊦
XXXYYYZZ q2 2	⊦	XXXYYYZZZ q2 B	⊦	XXXYYYZZ qc ZB	⊦*	qc XXXYYYZZZ	⊦
qc BXXXYYYZZZ	⊦	B qy XXXYYYZZZ	✓				