Immagine che contiene testo

Descrizione generata automaticamente

A,B sono regolari 🡪 Shuffle è regolare

a=|1|0|1|

b=|0|0|1|

w=|10|00|11

∑\* = (0+1)\* indica tutte le stringhe (compresa la vuota) ottenibili da linguaggio

Sappiamo quindi che esistono due automi del tipo:

DA = {QA, ∑, δA, qA, FA}

DB = {QB, ∑, δB, qB, FB}

Vogliamo produrre quindi:

DS = {QS, ∑, δS, qS, FS}

L’automa quindi sa l’ordine e che gli elementi sono invertiti, conseguentemente si va comunque almeno ad uno stato accettante.

L’idea informale è di costruire un automa che prende la posizione generica, richiama la procedura e poi non vedendo il funzionamento interno, richiamo i simboli avendo una dimostrazione generica:

Immagine che contiene lavagnabianca, testo

Descrizione generata automaticamente

Dim. formale

DS = {QS, ∑, δS, qS, FS}

Generica δ(qX, a) 🡪 q

Ad esempio metto come input il simbolo “a”:

δ( (x,y,A), a) 🡪 (δA(X,a),y, B)

L’idea quindi è che gli input si alternano e mi resta da aggiornare lo stato in cui si trova A, usando le sue funzioni di transizione.

x stato corrente DA

y stato corrente DB

A flag

δ((x, y, B), b) 🡪 (x, δB(y,b), A)

δ(q,x) 🡪 q

L’idea è che la tupla rappresenti gli stati dell’automa, ottenendo gli stati nuovamente e riorganizzandoli, costruendo automaticamente la funzione di transizione di δS.

Quindi l’idea è di avere un aggiornamento degli stati tali da avere le transizioni ricombinate, rimanendo con gli input uguali

- QS

QA x QB x {A, B}

Usiamo quindi “x” come prodotto cartesiano, tipo avendo

{a,b} x {c,d} eseguo il prodotto tra insiemi (prodotto cartesiano)

={(a,c), (a,d), (a,c), (b,d), (b,c)}

{qx, qy} x {qZ} x {A, B}

{(qX, qZ), (qY, qZ)} x {A, B}

(qX, qZ, A)…..

La risposta è: creare gli stati come prodotto (producendo tutto “brutalmente” con tutte le possibili combinazioni).

Stato iniziale qS

qS = (qA,qB,A)

δ[(qA,qB,A), a1] 🡪 (δ(qA, q1))

FA x FB x

bK

δ((qX, qy, B), bK)

L’={w | dehash(w), w ∈ L}

L regolare 🡪 L’ regolare

Immagine che contiene testo

Descrizione generata automaticamente

L linguaggio regolare su ∑

L’ = {y | xy ∈ L per x ∈ ∑\*}

w=1011, w ∈ L

1

11

011

1011

ε

(Q, δ, q, F, ∑)

Immagine che contiene testo

Descrizione generata automaticamente

Quindi:

{xz | xyz ∈ A, x, z ∈ Z\*, y ∈ Ʃ}

Sia M=(Q, Ʃ ,δ, qo, F) un DFA che riconosce A.

M’=(Q’, Ʃ , δ’, qo, F’)

In pratica si saltano a random tutti gli elementi, andando in uno, un altro o un altro ancora, senza un ordine definito. Per ognuno deve ricordarsi che ha saltato quel carattere (non ne salta più di uno):

w1w2 ……….wr

q0q1………….qn

Si introduce quindi un simbolo tale da capire se si è saltato o meno, eseguendo poi la transizione:

Q’=Q x {0,1} (q, 0) “non ho saltato”

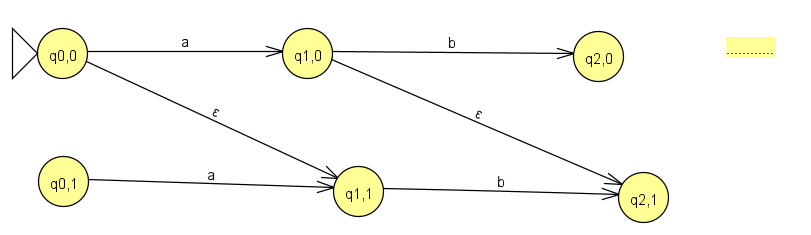
q’ = (q0, 0) (q,1) “ho saltato”

Qui ragiono che per qualche a ∈ Ʃ

δ’ ((q,0)a)={(δ(q0,a),0)} (quindi ho un simbolo e vado avanti con 0)

δ((q,0)ε)={(p,1)|δ(q,a)=p (salto un simbolo oppure rimango sulla transizione stessa perché vuota)

δ’ ((q,1)a)={(δ(q,a),1)} (quindi ho un simbolo e vado avanti con 1)



F’={(q1,1)|q0 ∈ F}



Immagine che contiene testo

Descrizione generata automaticamente



Per dimostrare che A/B è regolare, possiamo costruire un automa a stati finiti, in particolare un ε-NFA che, partendo dall’automa a stati finiti A=(Q U {q0}’, Ʃ, q0, δ, F) che ha lo stesso insieme di stati, stesso stato iniziale e gli stessi stati finali. Essendo che si deve sempre essere la produzione di “w”, lo stato iniziale conterrà, con l’aggiunta di una ε-transizione, un automa con tutti gli stati finali raggiungibili dal vecchio stato iniziale, con l’aggiunta della nuova stringa. Gli stati finali sono inoltre gli stessi.



Per dimostrare che A/B è regolare, possiamo costruire un automa a stati finiti, in particolare un ε-NFA che, partendo dall’automa a stati finiti A=(Q U {q0}’, Ʃ, q0, δ, F) che ha lo stesso insieme di stati, stesso stato iniziale e gli stessi stati finali. Essendo che si deve sempre essere una stringa in più, lo stato iniziale conterrà, con l’aggiunta di una ε-transizione, un automa con tutti gli stati finali raggiungibili dal vecchio stato iniziale, con l’aggiunta della stringa prefissa. Gli stati finali sono inoltre gli stessi.

Immagine che contiene testo

Descrizione generata automaticamente

(a) 3: si necessita di avere almeno tre caratteri per poter cominciare ad eseguire il pumping. Infatti parole come 00, 11 o similari non sono acettate.

(b) 3: parole come 11 (cioè xy0z) non sono accettate per costruzione, prendendo ad esempio 101 come xyz si nota che devono esserci entrambi per poter cominciare a pompare.

(c) 3: L’unione c’è ma non ci interessa, la proprietà vale comunque. Inoltre si nota che devo avere almeno due 1 ma l’idea è la stessa dell’esercizio precedente.

(d) 1: l’insieme vuoto deve avere almeno una stringa con cui operare e quindi pompare all’infinito una parola.

(e) 2: abbiamo bisogno di entrambi i caratteri per pompare.

(f) 3: le parole devono essere divise e definite in 3 pezzi anche qui per formare una parola valida e pompare.

(g) 4: in pratica la lunghezza deve essere 4 in quanto potrebbe esserci una suddivisione che non sbilancia la stringa avendo soli tre caratteri, avendo stringhe che pompate non sarebbero nel linguaggio. Sapendo che (11\*0)\* è l’espressione minima, avremo quantomeno bisogno di 0 oppure 1 per cominciare a pompare.

(h) 4: si vede infatti che ipotizzando 3 come lunghezza minima, potremmo avere una stringa del tipo 111 che rimane sempre regolare.

(i) 3: se sappiamo (penso io) che la stringa precedente non fa parte del linguaggio, significa considerando l’alfabeto precedente che necessitiamo di almeno tre caratteri per poter avere una stringa pompabile, considerando il caso complementare a quello descritto sopra.

(j) 4: potremmo banalmente avere la stringa 000 che non sarebbe pompabile.

(k) 2: anche qui, scegliendo 1 non sarebbe pompabile; con 2 almeno avremmo il possibile caso 01, regolarmente pompabile.

(l) 3: lasciando stare il caso dell’unione, comunque si nota che la stringa 001 non può avere 2 come lunghezza minima pompabile.

(m) 1: la stringa vuota deve necessitare di almeno un carattere qualsiasi.

(n) 3: si considera infatti che la minima stringa effettivamente pompabile sia 001

(o) 5: di fatto 1011 potrebbe non essere accettata dal linguaggio come pompata, perché già integralmente parte del linguaggio stesso.

(p) 2: considerando che la stringa vuota necessita di almeno un carattere e l’alfabeto la comprende di sicuro essendo star, potrebbe bastare per un generico alfabeto avere un solo altro carattere.

Immagine che contiene testo

Descrizione generata automaticamente

L’obiettivo come al solito è di costruire un automa in grado di riconoscere questo linguaggio presentato.

L’alfabeto Σ = {0, 1} e lo stato iniziale. Partiamo quindi dall’automa A=(Q, Ʃ, q0, δ, F) con una funzione di transizione costruita come:

* δ(x,y,01),a) = (δ1(x,a),y,01)
* δ(x,y,10),a) = (δ1(x,a),y,10)

perché devo avere ugual numero di occorrenze di 01/10.

L’insieme degli stati finali sarà costituito da:

F= FA x FB x {0,1}

Avendo come stato iniziale (partendo da A):

q=(qA,qB,A)

e avendo come stato accettante

F= FA x FB x {A}

Immagine che contiene testo

Descrizione generata automaticamente

1) Assumiamo per assurdo il linguaggio sia regolare.

Ricordiamo che “u” è un simbolo della grammatica, quindi può essere sia 0 che 1

Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

w=0pu0p avendo almeno xy completamente piena di 0.

Il pumping xyiz non dimostra lo sbilanciamento, infatti semplicemente la stringa

rimane composta di egual numero di 0 e sempre la stringa u in mezzo.

Un qualsiasi pumping per k > 0, ad esempio.

w=0ku0p-k comunque non dimostra alcunché,

infatti abbiamo un numero di u sempre compreso tra un egual numero di zeri ed il linguaggio rimane

sempre regolare.

2) Assumiamo per assurdo il linguaggio sia regolare.

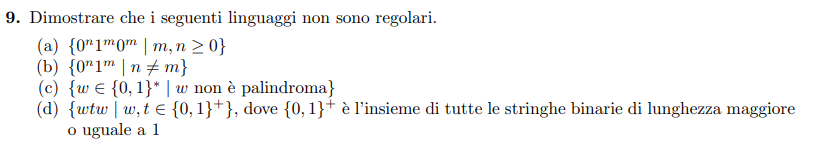
Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

u=0 e w=0p10q0k

In questo caso, siccome x=0k ed y=0q allora avremo in “z” la rimanente parte della stringa, quindi:

xy0z = xz=0k10k-p-q, quindi 02k-p-q1

Si vede quindi che il numero di 0 è sbilanciato rispetto al numero di 1 e il linguaggio quindi non può essere regolare.



a) Assumiamo per assurdo il linguaggio sia regolare.

Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

w=0k1p0k

e con pumping i=0

avremo 0k-p1k0k che chiaramente non appartiene al linguaggio e quindi non è regolare.

b) Assumiamo per assurdo il linguaggio sia regolare.

Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

w=0p1q con p > q o p < q, entrambi > 0 e < k

In questo caso è semplice perché basta letteralmente prendere x=ε, y=0p e z=1q

tale che pompando i avremo sempre un numero > di 0 rispetto agli 1.

Banalmente la cosa è dimostrata anche nel caso xy0z con n ≠ m e quindi

si dimostra che non è regolare.

c) Assumiamo per assurdo il linguaggio sia regolare.

Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

w=0p1q0r

In questo modo, eseguendo un pumping di i=0, avremo

0p-r1q0p che rivela la parola non sia nel linguaggio e dunque non è regolare.

d) Assumiamo per assurdo il linguaggio sia regolare.

Allora esisterà una suddivisione w=xyz tale da avere x ≠ ε, |xy| <= k con

0k1p0k. Eseguendo il pumping, ad esempio con i=2

e scegliendo y=0k1p avremmo una cosa del tipo 02k+2p0k che non rispetta |xy| <= k

nel numero di 0 ed il linguaggio non è regolare.



Sappiamo che L è context-free. Esiste quindi una grammatica G in forma normale del tipo::

A 🡪 BC

D 🡪 d

Se A 🡪 BC allora possiamo scrivere una parola w=bc

Il suffisso che è parte della parola, potrà essere:

* caso 1 (prefisso nullo) BC
* caso 2 (prefisso che è una parte di B): B’C
* caso 3 (prefisso che è esattamente B): C
* caso 4 (prefisso che è B più una parte di C): C’
* caso 5 (ho solo prefisso): ε

Ci sarà quindi la regola:

A 🡪 BC |B’C|C|C’|ε

D’ 🡪 d| ε

La grammatica G’ quindi avrà S’ come stato iniziale e condividerà gli stessi stati per le stesse regole.

Così descritta la grammatica G’ è context-free in quanto in CNF.



L’operazione di intersezione è definita liberamente per i linguaggi regolari; significa quindi che se eseguiamo l’intersezione di due linguaggi context-free, in questo caso A e B, anch’essa deve essere CF.

Considerando ad esempio due generiche parole del linguaggio w=apbqcp e w=aqbpcq

* caso base, con p=q=0 sono: abc & abc, dunque l’intersezione è fattibile
* caso induttivo, con generici esponenti p, q entrambi > 0 possiamo avere:

(es. p=2, q=3)

a2b3c2 & a3b2c3

avremo che l’intersezione genererà numero diverso di a, di b, e di c.

Quindi ad esempio potremmo avere stringhe del tipo:

a2b5c3 , a5b3c3, ecc.

Eseguendo l’intersezione si nota che otterremmo stringhe diverse e, se volessimo dimostrarlo con il PL per linguaggi CF, otterremmo sempre stringhe sbilanciate. Dunque l’operazione di intersezione non può essere chiusa per il linguaggi CF.

Immagine che contiene testo

Descrizione generata automaticamente

Per dimostrare che i linguaggi *sono* context-free dobbiamo necessariamente fare delle derivazioni.

Se dovessimo dimostrare che *non* sono context-free allora dovremmo usare il PL per CFL.

a) S 🡪 1X1X1X1X

X 🡪 0X|1X|ε

b) S 🡪 0A|1A

A 🡪 0S|1S|ε

c) S 🡪 0A|1B

A 🡪 0|ε

B 🡪 1|ε

d) S 🡪 0S0|1S1|0|0S1|1S0

e) S 🡪 0|1|1S1|0S0|ε

f) S 🡪 T0T

T 🡪 1T0|0T1|0T0|ε

g) S 🡪 1A|0A

A 🡪 A0|

h) S 🡪 A

A 🡪 #B|0A0|1A1

B 🡪 0B|1B|ε

i) S 🡪 A#B|B#A|ε

A 🡪 TAT|0

B 🡪 TBT|1

T 🡪 0|1

j) S 🡪 AB|BA

A 🡪 0|0A0|0A1|1A0|1A1

B 🡪 1|0B0|0B1|1B0|1B1

k) S 🡪 S1|S2

S1 🡪 aA

A 🡪 bAb|aA|ε

S2 🡪 Bb!aBb

B 🡪 Bb|aaBb|aBb|ε



Se A e B sono linguaggi regolari, allora sono descrivibili mediante le medesime operazioni dei linguaggi regolari e successivamente almeno da un automa DFA/NFA.

Sia M=(Q, Ʃ ,δ, qo, F) un DFA che riconosce A.

M’=(Q’, Ʃ , δ’, qo, F’)

Descriviamo quindi A, composto evidentemente da stringhe “x”, poi B, composto da stringhe di tipo “y”

Significa quindi che per questi dettagliamo:

(rA, L) 🡪 (sA,M) se δL(rA, x)=sA

(rB, L) 🡪 (sB,M) se δL(rB, y)=sA

Similmente è definita (rA, L, M) 🡪 (sb,M, L) se δL(rA, x)=δL(rB, y).

A queste condizioni notiamo che il linguaggio context-free necessariamente richiede che gli stati finali corrispondano tra i due linguaggi.

Articoliamo quindi come idea di stato finale F (δ0, x), F(δ1, y) = F(X **.** Y)

Necessariamente l’idea è che, nel caso non esista uno dei due stati,

lo stato finale deve essere:

F(X **.** Y) = F (δ0, x)

oppure

F(X **.** Y) = F(δ1, y)

per la caratterizzazione di termini di cui sopra. Dunque il linguaggio è CF.



Sapendo che G è in forma normale di Chomsky, quindi nella forma

A🡪 BC

A 🡪 a

Prendiamo l’esempio di una grammatica di un es. fatto in questo file:

S’ 🡪 AX|YB|a|AS|SA

S 🡪 AX|YB|a|AS|SA

A 🡪 b|AX|YB|a|AS|SA

B🡪 b

X 🡪 SA

Y 🡪 a

Consideriamo una stringa di lunghezza 1, possibilmente composta da una situazione del tipo

A 🡪 BC

Sono tutti simboli terminali e quindi, ciascuno è in forma normale di Chomsky, non avendo regole unitarie/transizioni vuote/più di tre transizioni per regola.

Induttivamente, nel caso di n > 1, avremo una situazione in cui, per poter applicare Chomsky abbiamo bisogno di almeno 2 regole, una terminale e non terminale ed eventuali altre regole.

Quindi:

A 🡪 BC

B 🡪 b

C 🡪 b|c|d

Ogni singola derivazione, per Chomsky, richiede esattamente l’applicazione di una sostituzione in altra regola e ogni regola può essere composta fino ad un massimo di 2 regole.

Essendoci almeno una regola terminale, per ipotesi, il numero di stringhe necessarie a comporre tale situazione sarà 2n.

Data appunto la presenza di almeno una regola terminale, il numero di stringhe necessarie a mostrare tale situazione in una derivazione si compone di passi.

In una situazione reale come la grammatica ottenuta sopra, si nota che per ogni simbolo di partenza ne corrispondono almeno 2 derivati, con l’aggiunta della regola terminale.

Consideriamo quindi S’ 🡪 AB ed S 🡪 XY. Tale situazione comporta due regole non terminali ed un’altra possibilmente terminale. Dunque generalmente Chomsky richiede questo tipo di derivazione.