Domande “teoriche” di TAC - PARTE DI AUTOMI

DFA

Cos’è un DFA? Qual è il suo output? Definiscilo anche formalmente

è una macchina riconoscitore chiamata automa a stati finiti deterministico, che presa una stringa ritorna VERO se appartiene al linguaggio da essa descritto o FALSO altrimenti. Viene definita come M = (Q, q0, delta, F, SIGMA) = (stati, st.init, f.transiz., st.end, simboli)

Definisci la funzione di transizione

è una funzione totale (= esiste sempre una mossa possibile) da Q x SIGMA→ Q che definisce la mossa possibile dell’automa (delta(q, sigma) = q’)

Definisci alfabeto, stringa come giustapposizione e come funzione, simboli, ecc…

Un alfabeto SIGMA è un insieme di simboli, in genere finito e non vuoto. Le stringhe con giustapposizione (solita rappresentaz. da sx a dx) sono ambigue ({0, 1, 01} → come faccio a capire se “01” è “01” o “0” concat “1”?). Dunque si usa la funzione u: [1,..n] → SIGMA.
La stringa vuota è una stringa, non un simbolo (al max un META simbolo).

Definisci concat, linguaggi e operazioni sui linguaggi

La concat è definita come funzione “u:[1,n]→SIGMA” uv ([n+m]→SIGMA)
Oppure induttivamente come (uv)(i)={u(i) se 1<=i<=n, v(i-n) se n+1<=i<=n+m)
Un linguaggio è, invece, un sottoinsieme di SIGMA* (il più piccolo è l’insieme vuoto, poi {eps}, fino a SIGMA*). Le op. sono: U, concat, L^n (e L*) che deriva da u^n

Esibisci un automa che rifiuta tutto, che accetta tutto

1) tutte le transizioni in q0 (non finale), 2) tutte le transizioni in q0 (finale)

Definisci L(M) con il sistema di transizione e fai un esempio

Un modo di definire l’insieme delle stringhe accettate è con sist. di transizione (“a runtime”): si prendono in esame le configurazioni <q,u> e si descrivono le transizioni tra mosse come: <q, sigma.u> → <q’, u> aka delta(q, sigma) = q’. Ad esempio <q0,001> → <q1, 01> → … <q2, eps> (OK se q2 è F). Quindi le def è composta da: 1) config., 2) riduzione a un passo contenuta in Conf x Conf, 3) direttiva input, conf init <q0, u>, 4) direttiva finale (if è F)

Cosa vuol dire che un DFA è deterministico e terminante?

det: per ogni config c esiste al più un c’ in cui posso andare (1 e 1 sola mossa possibile)
term: non esiste una seq infinita, t.p.c cioè non arrivo a una config finale

Definisci L(M) con la funzione delta* e fornisci uno pseudocodice

delta*: Q x SIGMA* → Q / d*(q, eps) = q / d*(q, sigma.u) = d*(d(q, sigma), u)

```
State d*(Statr q, List[simb] u):
        if empty(u): return q
        <sigma, u’> = u // x :: t = sigma :: u’ stile Ocaml
        State q’ = d(q, sigma)
        return d*(q’, u’)
```

NFA

Cos’è un NFA? Da cosa è dato il non determinismo? Definiscilo anche formalmente

Un NFA è un automa non deterministico, cioè in cui possono esserci più mosse effettuabili a partire da uno stato. La def è uguale al DFA ma con l’insieme delle parti come codominio della delta, data la possibilità di finire in più stati diversi.

Parla dell’albero della computazione e delle configurazioni di arresto

L’albero ha radice <q0, u> e dà tutte le possibili computaz. fino alle config. d’arresto (foglie)
È accettata se almeno una computazione la accetta. Le config d’arresto (terminazione diversa da stato F) sono: “<q, eps> con str finita” o “<q, u> con u!=eps e d(q,sigma) vuoto”

Definisci L(M) con il sistema di transizione, come esplicare il non determinismo
<q, sigma.u> → <q', u> se q' ∈ delta(q, sigma) (come DFA ma con ∈ ad insieme)
Il non det. si esplicita scrivendo c → c1 != c2 ← c (la def. è uguale ai DFA perchè il non determinismo è implicito nel *, con significato di "esiste almeno una computaz. t.p.c).
Definisci L(M) con la funzione delta*
d*: Q x SIGMA* → P(Q) / d*(q, eps) = {q} / d*(q, u.sigma) = $U_{q' \in d^*(q,u)} d(q',sigma)$
L'NFA aumenta il potere espressivo? Che teorema consegue dalla risposta?
NO, sempre i regolari. Aka esiste DFA t.c L=L(Md) sse esiste NFA t.c L=L(Mn)
Ne consegue il teorema di Rabin-Scott: Dato Mn esiste Md | L(Md) = L(Mn)
Quale tra le due implicazioni è ovvia? Perchè la dimostrazione dell'altra è costruttiva?
Quella ovvia è da DFA a NFA (perchè il DFA è un NFA particolare senza determinsimo).
l'altra implicazione è costruttiva perchè fornisce anche un modo per costruire un DFA a partire da un NFA (costruendo tabella con gli insiemi degli stati)
Spiega la dimostrazione dell'implicazione complicata (4)
Proviamo il teorema di R-S per induzione su |u|, portando Mn = (Q, S, q0, $delta_N$, F_N) in Md = (P(Q), S, {q0}, $delta_D$, F_D), cioè che ∀ u ∈ SIGMA* $d^*_D(\{q0\}, u) = d^*_N(q0, u)$
P.B: u = eps → $d^*_D(\{q0\}, eps) = \{q0\} = d^*_N(q0, eps) = \{q0\}$
P.I: $d^*_D(\{q0\}, u.sigma) = d_D(d^*_D(\{q0\}, u), sigma)$ = *(HP)* $d_D(d^*_N(q0, u), sigma)$ = (per def) …
… = $U_{q' \in dN^*(q0,u)} d_N(q',sigma) = d^*_N(q0, u.sigma)$
ε-NFA
Cos'è un ε-NFA? Da cosa è dato il non determinismo? Definiscilo anche formalmente
è un NFA in cui posso anche fare mosse in cui non leggo (silenti). Il non-det, oltre che per i motivi negli NFA, è dato anche dalla scelta sul "leggere o non leggere". La def è uguale se non per delta: Q x SIGMA U {eps} → P(Q) (ho aggiunto "eps" come colonna nella tabella)
Definisci L(M) con il sistema di transizione, quali sono le configurazioni di arresto?
Stessa def ma si aggiunge la "regola" <q, u> → <q', u>, q ∈ d(q, eps). Oltre a non det. ho anche non terminazione perchè non ho la garanzia di "mangiare" 1 simbolo ∀ iterazione.
Le conf di arresto sono: 1) <q, eps> e d(q, eps) vuoto (str finita e no mosse silenti possibili) 2) <q, sigma.u> e d(q, eps) vuoto e d(q, sigma) vuoto (str non finita ma no mosse possibili)
Definisci L(M) con la funzione delta*, cos'è la ε-closure? Definiscila formalmente
Si completa la definizione con la "regola": $U_{q' \in d^*(q,u)}$e-closure(d(q',s)). La e-closure è, ∀ stato, l'insieme degli stati ottenibili con stati silenti (compreso sempre se stesso). La e-closure è definita con: se q' ∈ d(q, eps) allora q' ∈ e-c(q) → chiudo per transitività
Dimostra che con i ε-NFA non aumenta il potere espressivo
Ovvio, come prima, da NFA a e-NFA (NFA è un tipo di e-NFA senza transizioni silenti). Meno ovvio il contrario, che non abbiamo dimostrato ma che fornisce un metodo per costruzione.
Minimizzazione di DFA
Cosa significa minimzzare un DFA? Perchè si dice algoritmo di collassamento?
Significa passare da un M a un Mmin t.p.c L(Mmin) = L(M) e ha il min(#stati). Si dice alg. di collassamento perchè prima collassa tutto, e poi risale: 1) separo init e finali (perchè si comportano all'opposto con eps) / 2) controllo quali str differenziano, nel caso decompongo
Definisci la relazione di equivalenza tra stati, definisci M(min) e anche delta(min)
q ~ q' sse ∀u ∈ S* (d*(q,u) ∈ F sse d*(q',u) ∈ F)
cioè q ~ q' sse u è accettata a partire da q sse u è accettata a partire da q'
Fornisci uno pseudocodice per la minimizzazione
TBA

Regex

Definisci i passi base delle regex e le operazioni possibili

PB: vuoto, eps, sigma / IND: r1 | r2, r1r2, r*

estensione ai linguaggi: vuoto, {eps}, {sigma}, L(r1) U L(r2), L(r1)L(r2), L(r)*

Descrivi i passaggi da RE a MinDFA. Vale l'opposto? Cosa ho implementato?

RE → e-NFA → NFA → DFA → MinDFA (e viceversa, e potendo skippare passaggi)

Ho implementato uno scanner/tokenizer/analizzatore lessicale

Come si passa da regex a ε-NFA? Passi base e induttivi?

vuoto: stato init + stato finale non raggiungibile / eps: solo stato finale iniziale senza mosse

sigma: stato iniziale non finale da cui poter raggiungere solo stato finale con sigma

→ IND → r1|r2: due diramazioni da q0 con transizione silente, che rientrano in stato finale

r1r2: sequenza di stati con in mezzo trans. silenti / r1*: posso passare nel finale con transizione vuota oppure in un insieme di stati da cui poi o rimanere o andare nel finale

Descrivi le proprietà di chiusura (L regolare → complemento, U, concat, intersect, *)

complementare: tutti i compl. di insiemi finiti regolari sono regolari (scambio F con Q\F)

L1 U L2 (prendo r1|r2) e uguale per la concat e per la *

L1 intersect L2 (uso che L1 intersect L2 = !(!L1 U !L2)).

Pumping Lemma

Dai una grammatica per {anbn|n>=0} e spiega perchè non basta un DFA

S ::= a S b | eps (non riesco con DFA perchè non ha memoria (se non per gli stati).

Introduci, spiega e definisci il pumping lemma. Dai una piccola prova.

Se L è reg $\exists n \in N \mid \forall z \in L, |z| >= n$, posso decomporre z come uvw …

… dati " $|uv|<=n, |v|>0$" → $\forall i>=0\ uv^iw \in L$

Sia n #stati di M, con m>n: q0 →(a) q1 → (a) qn → (a) qm (per forza stato ripetuto)

Prova: z = u.v.w con |uv| <= n e v != eps → prova z = s1…sm ∈ L (m > n)

q →(s1) … →(si) → qi=qj(ciclo !=eps) →(sm) q ∈ F / q0 →(u) !q (ciclo v) →(w) qf ∈ F

Idea: ipotizzo di avere |a| > |b|: non deve riconoscere, e invece riconosce

→ ES: ci sono 20k "a" e l'automa ha solo 10k stati: dovrò per forza ripetere il cappio(ciclo)

Come si usa il teorema?

L reg → prop. P(L) e quindi !P(L) → L !reg (riesco sempre a trovare ∀ decomp. una str fuori)

con $!P(L) = \forall n \in N\ \exists z \in L, |z| >= n$, posso decomporre z come uvw …

… dati " $|uv|<=n, |v|>0$" → $\exists i>=0\ uv^iw \notin L$

Grammatiche CF

Ricorda la definizione di Grammatica e spiega i suoi componenti. Quando un L è CF?

G = (T, N, S, P) con T=terminali={0,..9,a,..b,..}, N=!term=meta-simb={Exp,Num,...},

S = assioma (f.e. Exp), P=produzioni=coppie A ∈ N ::= alpha ∈ (T U N)*

Un linguaggio è CF sse L=L(G) per qualche grammatica CF (o alternativamente MPDA)

Parla delle derivazioni a un passo, della chiusura e degli alberi di derivazione

A un passo: passaggio da un N a un alpha ∈ (T U N)* applicando una produzione

Chiusura: $L(G) = \{u \mid u \in T^*, S \text{–}(*)\text{→} u\}$

Albero di derivazione: modo alternativo con S radice, nodi=produzioni e str=frontiera

Come si prova che L(G) è proprio il linguaggio cercato?

L(G) ⊆ L: soundness (tutte le stringhe generate ∈ L, cioè non esistono str che !∈ L)

L ⊆ L(G): completezza (si generano tutte le stringhe corrette, niente rimane fuori)

Prova soundness e completezza sulla grammatica {u | #a = #b} (4)

Vedi quaderno

PDA

Cos'è un PDA? Quali azioni sono possibili? Definiscilo anche formalmente

I Push Down Automaton sono automi in cui, oltre a guardare lo stato, si guarda anche il top della pila. La mossa dipende da stato q, simbolo letto sigma, e simbolo top x. Quando si cambia stato si fa push di una sequenza di simboli e pop del top.

Definizione = (Q, SIGMA, q0, GAMMA (alfabeto pila), Z (init simb) ∈ GAMMA, delta, F)

con d = Q x (SIGMA U {eps}) x GAMMA → P(Q x GAMMA*)

stato x simbolo letto da stringa x simb. top → coppia stato x el. inseriti

pop: d(q, sigma, x) = q', eps (tolgo e non metto) / d(q, sigma, x) = q', x (lascio invariato)

Com'è fatta una computazione in un PDA?

La forma è <q ∈ Q, u ∈ SIGMA*, alpha ∈ GAMMA*>

passo lettura: <q, s.u, Xa> →(<q', b> ∈ d(q, s, x)) → <q', u, ba>

passo silente: <q, u, Xa> →(<q', b> ∈ d(q, eps, x)) → <q', u, ba> (solo insert in pila)

Quali sono le tre configurazioni di arresto possibili?

pila vuota: <q, u, eps> / str finita e no mosse silenti: <q, eps, xa> con d(q, eps, x) vuoto

str piena ma no mosse: <q, s.u, xa> con d(q, eps, x) e d(q, s, x) vuoti

Perchè un PDA non è in generale deterministico? (3 motivi). A quale condizione lo è?

1) + mosse possibili (|d(q, s, x)| > 1) / 2) + mosse silenti / 3) posso scegliere se leggere o no

Invece det se ∀ q,s,x |d(q, s, x)| + |d(q, eps, x)| <= 1

Quando non è terminante? Come si riconosce la terminazione di un PDA? (2 modi)

Sse esiste un ciclo di transizioni epsilon. Per riconoscere la terminazione di un PDA, invece:

1) Stati finali: L(M) = {u ∈ SIGMA* | <q0, u, z> →(*) <q, eps, ->, q ∈ F}

2) Pila vuota: L(M) = {u ∈ SIGMA* | <q0, u, z> →(*) <q, eps, eps>}

Perchè nei PDA il non determinismo aumenta il potere espressivo?

DPDA obv → PDA ma PDA →X DPDA : regola del prefisso (se str prefisso di altre: ∄ dpda)

1) <q0, aba|aba> → … <-, aba…

2) <q0, aba> → … <-, eps, eps… (se aba OK allora lo è anche sopra, ma sopra non è finito)

Come si prova che un PDA riconosca L? Fai esempio del "sse" con le str x = ucu^R

soundness + completezza: nell'es ogni str non in forma x è rifiutata + se accettata → forma x

→) <q0, ucu^R, z> →(*) <q0, cu^R, zu^R> → <q1, u^R , u^R> →(*) <q1, eps, eps>

←) <q0, ucv, z> →(*) <q0, cv, zu^R> → <q1, v, u^R> →(*) vuoto sse v = u^R → <q1, eps, eps>

Come si prova che L=L(Mpda) sse L=L(G) per qualche G?

→) ho M che accetta L → costruisco G che lo genera assumendo che posso sempre ridurre un PDA a 1 stato (Q = {q}, terminali: SIGMA, !term: GAMMA, assioma: Z, produzioni:

<q, a> ∈ d(q, s, x) → x := sa (con trans. silente d(q, eps, x) ∈ <q, a> → x := a)

←) Assumendo G sempre riducibile alla GreibachNF (A := sB1,...Bn, n>=0):

costruisco M_G con le corrispondenze di prima e produzioni GNF → <q, B1-Bn> ∈ d(q, s, A)

Trasforma una G in forma tabellare e mostra un esempio di parser

S := aB | bA		S	A	B
A := aS | bAA | a	a	B	S,eps	BB
B := bS | aBB | b	b	A	AA	S, eps

parseS(...) = { s = getSymb(); if(s = a) parseB(...) }

Mostra che valgono le prop di chiusura per solo alcuni operatori. Mostra esempi

L1 U L2: avrò L(G1)=L1 e L(G2)=L2 → S:=S1|S2 con renaming dei !term. (f.e N1 U N2)

uguale per la concat, mentre per lo * abbiamo L1* : S:= eps | S_1S (con S assioma totale)

L'intersect non va bene esibendo un controes.(→per legge allora neanche il complementare, perchè se è chiuso quello è chiuso anche l'altro)

L1 = {a^{n}b^{n}c^m|n,m>=0}, L2 = {a^{n}b^{m}c^m|n,m>=0}, L1intL2 = a^{n}b^{n}c^n NO CF

Esiste un algoritmo riconoscitore da G a PDA? In che forma deve essere la G?
Come da RE a DFA SI, se in GNF: ∀ passo provo tutte le produzioni (alg exp) fino alla lunghezza della stringa, se non trovo la str tra tutte le possibili <= |u| allora non esiste.
S → sB1,...Bn →(*) u (con |B1, …Bn| = |u'|, e |sB1,...Bn →(*) u| <= |u|)
→ <= |u| perchè con la GNF so che a ogni passo toglie un simbolo s
Pumping Lemma (2)
Descrivi il pumping lemma per le G CF ed esibisci una prova
L CF → ∃ n ∈ N | ∀ z ∈ L, |z| >= n, posso decomporre z come uvwxy …
… dati " |vwx|<=n, |vx|>0" → ∀ i>=0 uv^iwx^iy ∈ L
Prova: Dato L=L(M) per qualche GCF (assumendo G in Chomsky con produz A:=BC | s)
Sia m il #!term di G e z ∈ L, cioè t.p.c ∃ un albero di derivazione: se z è abbastanza lunga ho che l'albero è piu alto di m (il linguaggio è infinito, sennò sarebbe regolare, quindi è sempre possibile arrivare a una z "abbastanza lunga") → ∃ un cammino in cui un !term si ripete: *Indichiamo con A(1) la prima occorrenza e con A(2) la seconda. Allora z puo essere decomposta come uvwxy dove w e la stringa ottenuta espandendo l'occorrenza A(2) e vwx e la stringa ottenuta espandendo l'occorrenza A(1). Ma dato che si tratta di due occorrenze dello stesso non terminale A, e chiaro che se sostituiamo il sottoalbero con radice A(1) con quello con radice A(2) otteniamo ancora un albero di derivazione valido, per la stringa uwy. Viceversa, sostituendo il sottoalbero con radice A(2) con quello con radice A(1), otteniamo un albero di derivazione per la stringa* uv^2wx^2y*, e iterando la sostituzione otteniamo alberi di derivazione per tutte le stringhe* uv^iwx^iy *con* $i > 2$.
Prova i tre dettagli tecnici del pumping lemma
1) dato albero alto m → frontiera.len <= 2^{m-1} (devo prendere una stringa >)
PB: S→sigma con m = 1 e $2^{m-1} = 2^0$ (solo radice)
PI: S → ramo B e ramo C, e so che è al max binario perchè sono in ChNF (rispettivamente stringhe ub e uc alte hb e hc): |ub| <= 2^{m-1} e uguale per uc (per HP) → |ubuc| <= 2^m → 2^{m-1} è l'altezza di B e C, mentre contando S ho $2^{(m-1)+1} = 2^m$
2) dato n = |!term|, prendo |z| >= n = 2^m
2.1) |vwx| <= n : vedi note pag 23
2.2) |vx| > 0 : vedi note pag 23
Come si usa il teorema?
Provo !Cond: L CF → ∀ n ∈ N | ∃ z ∈ L, |z| >= n, posso decomporre z come uvwxy …
… dati " |vwx|<=n, |vx|>0" → ∃ i>=0 uv^iwx^iy ∈ L
Parla del parallelismo con i vincoli contestuali
I vincoli contestuali (f.e. ling_programm.) sono inesprimibili con una GCF. Abbiamo:
{ BNF di ling_programm. { {Programmi sintatticamente corretti { Programmi ben formati } } }
Domande "teoriche" di TAC - PARTE DI CALCOLABILITÀ
INTRO e PR
Quali domande ci poniamo? Come cambiano se ci restringiamo a f: Nx..xN→N?
Dato un problema, posso risolverlo con una macchina/alg/..? Non sempre! Non è che dipende dal formalismo scelto? Riduciamoci a fun da naturali a naturali: anche così non posso calcolare qualsiasi funzione!
Descrivi il formalismo delle primitive ricorsive(PR). Come lo definisco?
Un candidato a esprimere tutte le f calcolabili è PR. Lo definisco come un programma composto da una lista di dichiarazioni di funzioni f_dec1,... f_decn t.p.c f(x1,..xn) = e con e = z (zero) | S(e) | f(e1,.. en) | x (input) → NB: se !ric in e posso avere solo funzioni dichiarate prima, mentre se ricorsivo posso avere anche altre f ma sempre primitive.
Come mostro che PR non è un candidato ad esprimere tutte le f calcolabili?

Non va bene per un controesempio: "∃ f intuitiv. calcolab. MA non esprimibili in PR" perchè permette solo f totali (cioè f che terminano per ogni input)

Quale schema deve seguire PR? Quali vincoli ho?

1) f(x1, ..xn, 0) = e (regola che mi dice che accade quando ho z)
2) f(x1, ..xn, S(y)) = e' (regola che mi dice che accade quando ho il successore) → limitato
→ posso chiamare f scalando su un argomento, convenz. l'ultimo: f(x1, ..xn, y)

Mostra l'esempio di somma, moltiplicazione, potenza, pred, differenza, min, max, mod

sum(x, 0) = x , sum(x, S(y)) = S(sum(x, y)) / mul(x, 0) = 0, mul(x, S(y)) = sum(x, mul(x, y))
pow(x, 0) = S(0), pow(x, S(y)) = mul(x, pow(x, y)) / pred(0) = 0, pred(S(y)) = y
x-0 = x, x - S(y) = pred(x-y) / min(x,y) = x-(x-y) / max(x,y) = x + (y - x) / |x - y| = (x - y) + (y - x)

Come NON posso scrivere eq? Perchè?

eq(0, 0) = 1, eq(0, S(y)) = eq(S(y), 0) = 0, eq(S(x), S(y)) = eq(x, y) NO (ho scalato su due)

Cosa vuol dire esecuzione in PR? Fai un esempio e poi formalizza con un disegno

Vuol dire, fissato un p, espandere e →…→n. Ad es. sum(2, 2) → S(sum(2,1)) → S(S…
Formalizzando con la sostituzione e[e1/x1, .. en/xn] →n [(Msum)] →s
→m [(valuto sum(n,m)]

PR è terminante? Perchè?

Si perchè per induzione: se non c'è ricorsione OK, se c'è ric. OK comunque perchè ∀ call diminuisco la complessità del problema.

PR è deterministico? Perchè, è un problema? Parla della confluenza

No, perchè posso scegliere cosa espandere. Ho e → e1 →(*) n (*)← e2 ← e con e1 != e2 e dunque non è un problema perchè è confluente (termina sempre, so che pur con espansioni diverse ottengo sempre lo stesso risultato) → se voglio scelgo una convenzione.

Data una f che ritorna sempre zero, che differenze ho tra PR e un L di programm.?

Ad esempio nei ling_progr valuto prima gli argomenti dato che potrei non terminare: nei PR potrei valutare un "ret 0" in 0[e1/n1]=0, nel l_prog se chiamo m(e) con e !term anche m !term

Che differenza c'è tra la frase "f è ricorsiva primitiva" e "conosco un alg PR per f"?

g(x) = 1 se "frase di cui non so la risposta", 0 "altrimenti": o è sempre costante 1 o 0 (PR) MA non esiste un algoritmo per calcolarla (es: cong. di Gold.: ∀ pari > 2 è somma di 2 primi)

Parla della fun sugli almeno "5" consecutivi in π

Appartiene a PR, ma se tolgo "almeno" (voglio esattamente x "5" consecutivi) non lo so

Cosa significa che i programmi in PR possono essere numerati effettivamente?

Posso ordinarli per lunghezza (e con LEX a parità di len) e poi generarli con un alg (effett.)

Cosa significa PR sse N? Dai due algoritmi per getProg e getIndex, a cosa serve?

C'è una corrispondenza biunivoca con N, da cui derivo l'idea di libreria di programmi indicizzati. Ho l'algoritmo getProg (da cui ottenere un pn dato n) e getIndex (da cui ottenere l'idx in numerazione di un dato p): entrambi generano tutte le stringhe (saltando i programmi non ben formati e incrementando i solo per quelli ben formati) per poi ottenere l'output.

Come provo che esistono funzioni intuitivamente calcolabili che però ∉ a PR?

Fissiamo una numerazione f0, ..fi, ..: g(x) = $f_x(x)+1$ è calcolabile. MA ∉ a PR. Supponiamo lo sia per assurdo: se lo fosse ∃ x che è proprio g (x-esima f nello scaffale), allora dato $g=f_x$ $g(x) = f_x(x)+1 = f_x(x)$ che è un assurdo (per diagonalizzazione: g non può essere i e j insieme)

Su che due caratteristiche di PR funziona la prova? Quale è irrinunciabile?

1) f totali (irrinunciabile, e dunque gioco sul togliere questo: se ammetto parziali scappo dall'assurdo) / 2) effettivamente numerabili (obv)

Tale prova è generalizzabile? ∃ un altro modo per provare che ∃ f intuit.calc. ∉ PR?

Si, a tutti i formalismi equivalenti. Si, si può provare per complessità elevata (> fPR ∀ fPR), come ad es fAckerm. (peraltro totale, quindi PR non sa neanche esprimere tutte le totali)

Esempi di sg, isZero, eq, <, IF, even, Ef, sum, allpos, max?
sg(0) = 0, sg(S(y)) = S(0) / !sg(x) = isZero(x) = 1 - sg(x) / eq(x, y) = !sg(|x-y|) / x < y = sg(y-x)
$IF_{p,f,g}(x)$ = (p(x) * f(x)) + (!sg(p(x)) * g(x)) / even(0) = 1, even(S(x)) = !sg(even(x))
$sum_f(0) = 0$, $sum_f(S(x)) = sum_f(x) + f(x)$ / $AP_f(x+1) = sg(f(x)) * AP_f(x)$
$MAX_f(0) = 0$, $MAX_f(S(x)) = max(f(x), MAX_f(x))$ con max = max numerico definito in preced.
INTRO A MACCHINE DI TURING e RICONOSCITORI
Spiega l'idea e il funzionamento delle macchine di Turing
C'è una testina su un nastro che viene fatta scorrere a destra/sinistra, in base allo stato attuale e al carattere puntato.
MdT come riconoscitori: definizione? delta? rappresentazione? deterministici?
Gli MdT come riconoscitori prendono in input un u ∈ SIGMA* e ritornano un bool o non terminano. Li definiamo MdT = (Q, q0, SIGMA, GAMMA, B (blank), delta, F)
con SIGMA contenuto propr. GAMMA (G ha almeno B in più: B ∈ GAMMA \ SIGMA)
con delta : Q x GAMMA (simb letto) →(parziale) Q x GAMMA (simb sostituito) x {L, R}
→ è deterministico perchè ho 0/1 mosse (parziale con al più una mossa)
→ rappresentazione delta: tabellare, grafica, lista di istruzioni a quintuple
Come sono fatte le configurazioni? Cos'è la porzione significativa?
<alpha, q, beta> con q ∈ Q e alpha, beta (sx, dx) ∈ GAMMA* (con porzione significativa la configurazione minimale t.p.c non esistono simboli diversi da blank fuori dai limiti dei simboli rappresentati (che sono in realtà infiniti).
Perchè non si può dire che la MdT "legge l'input"?
MAI dire che la Mdt "legge/mangia" la stringa, perchè vale solo per gli automi a stati finiti (la MdT può andare a sx, a dx, stare ferma, ecc).
Quali sono i 6 casi di transizione di configurazione? (4)
init per riconoscere u: c0 = <eps, q0, u> → NB: a e b stanno per alpha e beta
1) c → c' guardando d(q, x) = <q', Y, R>: <a, q, xb> → <aY, q', b>
2) c → c' guardando d(q, b) = <q', Y, R>: <a, q, eps> → <aY, q', eps>
3) c → c' guardando d(q, x) = <q', Y, L>: <aZ, q, xb> → <a, q', ZYb>
4, 5 e 6 analoghi speculari
Confronta il concetto di memoria tra DFA, PDA e MdT
DFA: finita / PDA: non infinita ma comunque non limitata a priori (usabile come stack) /
MdT: nastro infinito con porz. significativa finita (se voglio alloc metto un simbolo ∀ mossa)
Quali sono le configurazioni di arresto? Che convenzione assumiamo?
<a, q, xb> con d(q, x) undef e <a, q, eps> con d(q, _) undef (convenz.: no mosse da q ∈ F)
Quali sono le possibili computazioni? Quando la stringa è accettata? Quando no?
è una sequenza da c0 (init) a cn finita (di arresto) NON a forma di albero perchè è determ.
La str è accettata se la computaz. da c0 termina con cn= <-, q ∈ F, -> (indipend. dal nastro)
Definisci un linguaggio accettato da una MdT
L(M) = {u ∈ SIGMA* | c0 →(*) <-, q ∈ F, ->} (non accett. se q ∉ F | se non termina)
Definisci un linguaggio ricorsivamente enumerabile. Quando è anche ricorsivo?
Un L è r.e se ∃ MdT | L = L(M), cioè se ∃ M | output su u è OK sse u ∈ L. Oppure ancora se esiste alg/MdT | ∀u, a partire da c0, M termina in q ∈ F sse u ∈ L (SI, NO, !term).
Invece è anche ricorsivo se termina sempre (ric(decidibile) → r.e(semi-decidib.))
Cosa significa decidibile e semi-decidibile? Sapresti disegnare un diagramma?
Sono sinonimi di ricorsivo e r.e: nel diagramma abbiamo {other {r.e {ricorsivi} } }
Posso eliminare il caso di arresto in uno stato non finale? Come?
Si, posso farlo mandandolo semplicemente in non terminazione

MACCHINE DI TURING COME CALCOLATORI

MdT come calcolatori: def? analogie e diff. con i riconoscitori? posso ricondurmici?

Sono definiti come M=(Q, q0, SIGMA, delta, _) senza stati finali (per l'output mi serve leggere il nastro perchè non è solo un bool, anche se ovviamente posso ricondurmi al caso riconoscitore con f: SIGMA*(IN) → {T, F} sul nastro.

Cosa sono Cin e Cout? Che proprietà deve avere la codifica?

Cin e Cout sono le funzioni di codifica da IN/OUT a SIGMA*. Sono codifiche iniettive (i != j → c(i) != c(j)) ma non necessariamente surgettive.

Cosa vuol dire che M calcola f? Cosa può accadere? E nei linguaggi di programmaz.?

∀ i ∈ IN, a partire da <eps, q0, Cin(i)>: 1) se term: <a, q, b> → o sul nastro c'è la codifica di qualche output o ∄ out | a.b = Cout(out) (cioè non esiste un output t.p.c quello che c'è sul nastro corrisponde a una codifica, che possiamo comunque mandare in non terminazione) 2) non termina: f(i) undef

Quando f è T-calcolabile? Fai alcuni esempi standard

f è T-calcolabile se ∃ M (con opportune codifiche) | fM = f (ES: totalm undef, identità, ..)

Spiega la tesi di Church-T. Perchè non è proprio una tesi?

T-calcolabile sse "calcolabile intuitivamente" (è più una congettura perchè non è dimostrabile, data l'informalità della parte destra).

Quali 2+1 osservazioni si possono ricavare da tali tesi? Che conseguenze hanno?

1) Per ogni f intuitivamente calcolabile nota si può dare MdT, 2) Per ogni forma alternativa (Mu-ric, lambda-calcolo, RAM) sono stati dati algoritmi di conversione reciproca + 2.1) si è dimostrato che sono equivalenti. (1) e (2) ci dicono che le f_calc sono sempre le stesse, 2.1) che anche gli algoritmi lo sono → f calc(o ric) = calcolabili in ogni formalismo T-completo

Parla della numerazione effettiva delle funzioni ricorsive, cosa significa?

Supponendo "M0, … Mx, …" macchine e "phi0,.. phix,..." f_x calcolate dalla x-esima M
Un Mdt è rappresentabile come str, poi ordino le str effettivamente, dove "effettivamente" significa che si può dare un algoritmo per generarle (x→[Alg]→Mx; M→[]→x t.c. M = Mx)

Quali 5 osservazioni ricaviamo? Che conclusione ne è conseguenza?

1) è possibile che x!=y ma phix=phiy. Anzi, ho infiniti indici per ogni funzione calcolabile
2) Conoscere un algoritmo per calcolare f = avere un indice nella numerazione
3) Come per PR (intuitivam.): sapere che f è ric è != da conoscerne l'idx (saperla calcolare)
4) |{funz.calc.}| = |{calc.tot}| = N / 5) esistono (molte!) funzioni !calcolabili: deducibile dalla cardinalità, anche solo riducendoci alle f caratteristiche con card. $2^{\#N}$ (∀ subset: ∈ o no) (non sono numerabili)

NB: le f calcolabili totali hanno card N ma non posso numerarle effettivamente (come capisco se un progr. termina sempre generando le stringhe? → no algoritmico)
→ conclusione: MdT sse ℕ sse Str su SIGMA (useremo ℕ come dati & come idx di progr.)

Come uso la tesi di Church nelle dimostrazioni?

Se so descrivere un algoritmo, chances are che per church è ricorsivo (senza dare MdT)

Cos'è e com'è definita la macchina universale?

Esiste per Church, è un interprete, e a partire da <x, y> calcola $phi_x(y)$ ritornando il risultato o non terminando se è undef.

RISOLVIBILITA' di PROBLEMI

Descrivi l'Halting problem.

Lo diamo come esempio di insieme non ricorsivo: ritorna 1 se $M_x(y)$ termina ("se simulo so dire se termina, non viceversa").

Come si prova? (Halting Theorem) (4)

Ci riconduciamo per diagonalizzazione a $f_K(x)$ (1 se $M_x(x)$ termina (su se stesso)):

Definiamo g(x) (negator) come: non termina se $f_k(x) = 1$, 1 altrimenti. Per assurdo: supponendo che f_k sia ricorsiva; se g è calcolabile ∃ idx z t.p.c $g=phi_z$ → applico a se stesso: $g(z) =phi_z(z) = 1$ (term & !term: ASSURDO → !ric).

Se fK non è calcolabile (ric), perchè è un assurdo che lo sia fH?

Perchè sennò potrei definire f_K come $f_K = f_H(x,x)$ con riduzione

Come si mappano questi ragionamenti sugli insiemi?

H = {<x, y> | $phi_x(y)$ term} non è ric. (MA r.e perché posso eseguire e dare 1 se termina)

Fai un riassunto della terminologia vista finora (4)

Vedi appunti

Quando un insieme è ricorsivo? E r.e.? Dai anche la definizione alterntiva e provala

1) è ricorsivo quando la sua funzione caratteristica (1 se x ∈ X, 0 altrimenti) è ricorsiva
2) è r.e se la semicaratteristica SC (1 se x ∈ X, !term altrimenti) è calcolabile(ricorsiva)
Def. alternativa 2): ∃ f ric | f(x) term sse x ∈ X → prova di analogia tra le definizioni:
→) la semi-car. è un caso particolare della f ricorsiva (al contrario di sotto, se termina, cioè ci appartiene all'insieme, ritorno x). Ricordiamo che insieme X e' Imm e anche Dom di una f calcolabile. Aka esiste una bigezione tra insieme e se stesso
←) se ∃ f ric, per calc. SC_x: input x; $M_f(x)$; return 1; (se termina: 1)

Cos'è un problema? Quando è (o non è) decidibile? Quando è semi-decidibile?

Ci riduciamo a un problema di decisione. Dato un P sottoinsieme di ℕ, P è decidib. sse {P} = {x | P(x) = 1} è ricorsivo. P invece è semidecidibile sse {P} è r.e

Cos'è il teorema di Post? Prova tutte e 4 le implicazioni

Dati A e il complementare CA: A ric. sse CA ric. sse A e CA r.e. Proviamolo:
- A ric → CA ric (e anche viceversa (←) perchè C(C(A) = A): inverto gli output
- se ric → anche r.e: dunque → è ovvia
- A e CA r.e → ric? (←): esecuzione in interleaving. Non posso sequenzialmente perchè una delle due non termina (se il primo per caso non termina non arrivo al secondo), dunque termina quella delle due che termina (cioè sempre → ric).

Alg: inp x; k=0; while(4) {if $M_a^k(x) = 1$: ret 1; if $M_a^k(x) = 1$: ret 0; k++}

Fai l'esempio visto di insieme non r.e.. Come si prova che non lo sia?

Il complementare di H: CH = {<x, y> | $phi_x(y)$ undef} (o K analogamente)
Si prova per Post: perchè se fosse r.e, dato che H lo è, per Post sarebbero ric. (ASSURDO)

Esponi le proprietà di chiusura per ricorsivi e r.e

Per ric: 1) complementare: Si, per Post, 2) A U B e A intersect B: si costruiscono secondo la logica booleana (posso anche per intersect perchè so che terminano sempre)
Per re: 1) complementare: No, per Post 2) A U B: interleaving 3) A intersect B: va bene sequenzialmente affinchè terminino se entrambi terminano.

Esponi le proprietà di chiusura su linguaggi, dando un algoritmo per la concat su ric.

concat su A,B ric: provo tutte le decomposizioni della stringa per capire se esiste una decomposizione della stringa t.p.c il primo pezzo ∈ A e il secondo ∈ B.
A* su A ric: w è concat di pezzi che ∈ A ($w = u_1, \dots u_n$ con n>=0, u_i ∈ A): posso farlo perchè il #casi è finito e dunque terminerà (w è una stringa, quindi le decomposizioni sono finite)
concat e * sui r.e: come i ricorsivi ma provo tutte le decomposizioni in interleaving

Ricorda la prima definizione di insiemi r.e. Quale altre due caratterizzazioni diamo?

1) ∃ M | ∀x ∈ ℕ M(x) = 1 se x ∈ X, altrimenti !termina (aka può essere semidecisa da un alg)
2) X = insieme dei valori di una f ricorsiva=calcolabile (Immagine, aka X = {f(x) | x ∈ ℕ})
3) X = ∅ oppure insieme dei valori di una f ric totale (Immagine di alg che terminano sempre)

Caratterizzazione 1 => Caratterizzazione 2?

Si costruisce da 1) un alg il cui insieme output è proprio X: in X; if M(x) = 1: ret x; else !term

Cos'è il metodo a zig-zag?

Per numerare effettivamente gli elementi di un insieme r.e l'interleaving non basta, perchè dovrei mettere in parallelo infinite macchine, Dunque costruisco idealmente una matrice con sulle righe gli input i ∈ ℕ, sulle colonne il #passi j e in posizione (i, j) l'esecuzione di M su i per j passi. Non posso andare per righe perchè andrei sequenzialmente fissando un input, e non posso andare per colonne perchè dovrei andare in interleaving con infiniti input. Vado dunque a zig zag incrementando entrambi gli indici i e j, e se esiste un i t.p.c $M_j(i)$ termina allora ho finito (non sono ricaduto in uno dei due infiniti).

Caratterizzazione 2 => Caratterizzazione 3? (4)

Se X = ∅ ok, sennò, se ∃ un primo valore trovato a zig-zag all'indice n0, definisco una funzione g: ℕ → ℕ che ritorna il valore associato all'n-esimo passo a zig-zag (se ∃) così: g(n) = x se n <= n0, g(n+1) = {y se il passo n+1 dà y, g(n) altrimenti} Ad esempio se g(7) = 4: g(n) = 4 se n <= 7, g(n+1) = {y se il passo n+1 dà y, 4 *(o un valore diverso eventualmente trovato prima di n0)* altrimenti}: ergo inizialmente ritorno x = g(n) = g(7) = 4 . Quando dopo n0 = 7 passi trovo un primo valore x (che può anche proprio essere 4 = g(7)) quello diventa il nuovo valore ritornato dalla funzione. → ric e anche totale (ritorna sempre qualcosa) e anche PR → ho numerazione perchè ho un alg per generare tutti gli elementi di X (while(4) { if passo n ret x: ret x; n++ })

Caratterizzazione 3 => Caratterizzazione 1?

Per passare dall'immagine di un algoritmo totale a una macchina che semi-decida l'algoritmo, costruisco una funzione g che termini se esiste almeno un x t.p.c f(x) = y, data f la funzione totale con Immagine A. Otteniamo così che g termina solo se f trova l'input y come output, che terminando solo su valori in A mi fa ottenere che g ha come dominio A. input y; x = 0; while(4) { if (M_f(x) = y) return 1; ++x; }

Che conclusioni otteniamo? Fai qualche esempio

Dunque sono uguali: ∃ alg che semidecide X e ∃ alg che genera tutti gli el. di X (num effett.)

ES: A,B != ∅; A e B r.e sse A x B r.e?

→) è un AND dunque mi basta eseguire in sequenza ret M_a(x) && M_b(x)

←) so M_{axb}(x, y) = {1 se (x,y) ∈ A x B; else ! terminaz.}: dato che != ∅, fisso un y ∈ B e dunque, se termina su (x, y), so che x ∈ A.

Dato T = {x | o(x) è tot}, come proviamo che non è ricorsivo?

Se lo fosse potrei def f(x) = {phi_x(x)+1 se phi_x tot; 0 altrimenti} → essendo ricorsiva f corrisponde a una phi_xnella numerazione, e dunque phi_x(x)+1 = f(x) = phi_x(x): assurdo

Dato T = {x | o(x) è tot}, come proviamo che non è r.e? Cosa abbiamo ottenuto? (4)

Se lo fosse avrei una fun che realizza la numerazione. Sia f tale fun t.p.c ∀ y-esima f totale f(y) = x-esima funzione, posso mettere a paragone le due numerazioni (quella delle totali tot e quella di tutti i programmi prog, con f la funzione che mi mappa dalle totali alla numerazione normale). Pongo g(y=idx_tot) = $phi_{f(y) = idx\_prog}$(y)+1 → g è ric/calc e anche tot perchè y ∈ totali → so che esiste in prog e, sapendo che è totale, anche in tot → phi_x(y) = g(y) = $phi_{f(y)}$(y) + 1 = phi_x(y) + 1 → assurdo (nb: f(y) con y idx_tot mappa in x = idx_prog). L'assurdo sta negli estremi (senza+1 e con +1)

Qual è un metodo più "facile" per provare ipotesi simili? Definiscilo

Si usa la relazione di riducibilità P <= Q t.p.c ∃ f_riduz da Ip a Iq | x ∈ P sse f(x) ∈ Q

Come usiamo questo metodo? Ipotesi? Tesi?

Usiamo questo metodo 1) in positivo: Q ric(re) → P ric(re) 2) neg: P !ric(re) → Q !ric(re)
HP: Q ric: ∃ Mq | x→Mq→1 se x ∈ Q / TH: ∃ Mp analogo?: x→Mf→Mq→ 1 se x ∈ P
NB: è riflessiva (P riducib. a se stesso con Id) e transitiva

Fai due esempi, uno con Z= le f costanti 0 e uno con EQ= l'equivalenza tra programmi.
1) Do una f per K <= Z: input y; if $M_x(x)$: ret 0; else !term} → Z ! ric
2) Dato K <= Z, vediamo Z <= EQ: avere EQ (x ass.) significherebbe poter dare in $input_x$ sempre z ∈ Z e in input y un programma casuale → ma ciò vuol dire che avrei un alg che mi dice se un y appartiene a Z… MA K <= Z e dunque assurdo → EQ ! ric

Quali sono le premesse al teorema di Rice? Cos'è una prop. semantica?
Dati i programmi in corrispond. biunivoca con ℕ e le proprietà di un programma con sottoinsiemi di ℕ, si dice che ogni proprietà semantica (non banale) dei programmi è non decidibile. Le proprietà semantiche, o estensionali, sono relative solo al comportamento del programma (e non al suo codice o altre caratteristiche non a runtime). Le proprietà banali sono invece l'insieme vuoto ⌀ e ℕ.

Ccome sono definite le proprietà estensionali?
PI estensionale sse x ∈ PI e phi_x = phi_y → y ∈ PI (cioè se, trovata un'altra funzione "uguale" nella numerazione, anch'essa appartiene a PI)

Esponi il teorema di Rice. Come possiamo usarlo? Possiamo usarlo su K?
"Una proprietà estensionale dei programmi è decidibile/ric sse è banale. Dunque se una PI è estensionale e non banale →(Rice)!ric. Non possiamo usarlo su K perchè non è estens.

Dai una riduzione generica K <= PI. Come si prova?
Data PI estens. e ! banale, diamo una riduz. generica K <= PI. Mappando K in PI e !K in !PI, scegliendo un elemento z ∈ PI (che sappiamo esistere perchè essendo PI non banale non è = ⌀) e considerando TU={x | phi_x tot. undef}: abbiamo che gli el. di TU sono tutti contenuti in PI o nessuno lo è (perchè TU è estensionale). Assumiamo ora che non siano contenuti (e si potrebbe provare anche con l'assunzione opposta) e scriviamo g(x):
input y; if $M_x(x)$ return $M_z(y)$; else ! term} → Abbiamo 2 possibilità:
1) "x ∈ K → g(x) ∈ PI": $phi_{g(x)}(y)$ = $phi_z(y)$ → g(x) ∈ PI (sotto l'assunzione che $M_x(x)$ termina, g(x) = z perchè calcolano la stessa funzione ($M_z(y)$ è la macchina z sull'input passato).
2) "x !∈ K → g(x) ∈ TU" → g(x) ∈ !PI (Se Mx(x) non term, allora non appartiene a K, ma allora g(x) è tot. Undef, quindi appartiene a TU, e quindi appartiene a !PI che contiene tutte le TU).
Al contrario, se ipotizzo che TU contenuti in PI, uso la stessa riduzione ma da !K e prendendo uno z in !PI.

Perchè le prop. banali sono decidibili? Perchè sono estensionali?
Sono ovviamente decidibili (⌀: sempre F, ℕ: sempre V). Perchè sono estensionali?:
⌀: perchè l'antecedente (in x ∈ PI) è sempre F (!∃ x ∈ ⌀)
ℕ: perchè y ∈ sempre a PI se PI = ℕ (è sempre un programma)

EXTRA: Teorema del parametro
Indicante la specializzazione del sw, mi dice che: se ho una f ricorsiva da $ℕ^{m+n}$ a ℕ, ∃ f ric totale s^m_n t.p.c da f(x1,.. xm, y1,.. yn) ottengo s_n(x1,.. xm) e phi_m(y1,.. yn) (f parziali)
Ho due conseguenze, dando f ricorsiva da $ℕ^{m+n}$ a ℕ:
1) fissando i primi m parametri, la nuova funzione che prende solo g(y1,.. yn) è ric
2) posso creare in modo algoritmico la f che fissa i primi parametri (provabile ∀ formalismo)
→ Ad es. posso fissare un x segnato per la riduzione a k (quando scrivo $M_x(x)$)

EXTRA: f Mu-Ricorsive
Per rendere TC le PR si aggiunge l'op. di minimizzazione, o mu ricorsione (ciclo non limitato a priori, aka while). Data f: $ℕ^{n+1}$→ ℕ ric/tot def. (con un param in meno) g: $ℕ^n$→ ℕ t.p.c
g(x1,.. xn) = mu^0(f(x1,.. xn, y) = {min{y | f(x1,.. xn, y) = 0} se ∃ ; undef altrimenti}
NB1: ovvio che mu_ric → ric per tesi di Church

NB2: Provo con alg che sia TC: (calcolo g a partire da f) (Ip fondamentale: f totale)

```
input x1, ..xn; y = 0;
while(4)
        if f(x1,.. xn, y) = 0: ret y
        y++
```

NB3: TC → mu_ric, e si prova anche che è nella forma t.p.c uso l'operatore 1 sola volta (KleenNF) e per il resto solo iterazioni limitate a priori (aka for):

$\forall z\ phi_z^k(x1,..\ xk) = p(mu_y^0(t_k(z, x1,..\ xk, y))$

ES: $floor(log_a(x)) = M_y^0(a^{y+1} <= x)$

ESERCIZI FINALI: Vedi