Immagine che contiene testo

Descrizione generata automaticamente

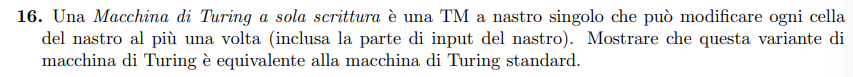
L’equivalenza tra un automa a coda ed una TM deve essere mostrata, quindi che il linguaggio può essere riconosciuto dall’automa e dunque dalla TM. Si dimostra creando una simulazione che fa capite che l’automa Q si comporti esattamente come la TM M.

*La simulazione di Q come M funziona nel modo seguente:*

si consideri l’intero nastro come una coda. Ciascun simbolo viene modificato e il movimento del nastro avviene a destra. Quando più di un numero viene pushato nella coda, si shiftano tutti i valori a destra. Se si raggiunge la fine del nastro, allora il simbolo più a sinistra viene considerato e si capisce di essere arrivati alla fine.

*La simulazione di M come Q funziona nel modo seguente:*

l’alfabeto della macchina M viene espanso aggiungendo un simbolo extra e un simbolo marcatore a sinistra viene inserito nella coda. I simboli sono pushati a sinistra e letti (pop) a destra.



*(Soluzione estesa del libro, tradotta e riscritta)*

Per prima cosa simuliamo una normale macchina di Turing con una macchina di Turing che scrive due volte.

La macchina a scrittura doppia simula un singolo passaggio della macchina originale copiando

l'intero nastro su una parte nuova del nastro sul lato destro della porzione attualmente utilizzata. La procedura di copiatura opera carattere per carattere, marcando un carattere mentre viene copiato. Questa procedura altera ogni quadrato del nastro due volte: una volta per scrivere il personaggio per la prima volta, e ancora per sottolineare che lo è stato copiato. La posizione della testina del nastro della macchina di Turing originale è contrassegnata sul nastro.

Quando si copiano le celle in corrispondenza o vicino alla posizione contrassegnata, il contenuto del nastro è aggiornato secondo le regole della macchina di Turing originale. Per eseguire la simulazione con una macchina scrivibile una volta, si opera come prima, eccetto che ogni cella del nastro precedente è ora rappresentata da due celle. La prima di queste contiene il simbolo del nastro della macchina originale e il secondo è per il simbolo utilizzato nella procedura di copiatura. L'input non viene presentato alla macchina nel formato con due celle per simbolo, quindi la prima volta che il nastro viene copiato, i segni di copiatura sono messi direttamente sopra il simbolo originale

*(Soluzione mia, che segue tipo quelle degli ε-NFA)*

Partendo dalla macchina originale, si copia l’intero nastro tra le transizioni. In questo nuovo nastro viene aggiunto un simbolo delimitatore (caso TM a nastro semi-infinito) ed un alfabeto espanso per registrare la posizione del nastro quando è stato copiato. In questo caso si hanno due scritture, la prima usata quando si copia il simbolo sul nuovo nastro, la seconda per indicato che sia stato copiato.

Avendo le transizioni uguali, ma TM a sola scrittura è equivalente a quelle standard.

Immagine che contiene testo

Descrizione generata automaticamente

1. Si considera un linguaggio CF, che viene riconosciuto da un PDA.

Ad esempio vediamo L={anbncn | n >= 0}

Usiamo il PL per linguaggi CF dimostrando che non si tratta di CFL.

Ad esempio consideriamo, per s=uvxyz, una stringa s=1p0p1p0p

Quindi consideriamo che s sia nella forma detta e:

* se v e v contengono al più un tipo di simbolo dell’alfabeto, la stringa sarà nella forma uv2xy2z e gli 0 ed 1 saranno di lunghezza diversa
* se v ed y contengono più di un tipo di alfabeto per simbolo e comunque questi non saranno nell’ordine corretto, portando ad una stringa sbilanciata

Dunque l’1-PDA (PDA regolare) non risconosce L.

Se siamo come visto nella forma uv2xy2z allora viene riconosciuto dal 2-PDA:

* si fa push di tutte le “a” che appaiono di fronte al nastro di input sul primo stack
* si fa push di tutte le “b” che seguono le “a” nella seconda pila. Se si vede qualche “a”, si rifiuta
* quando vede la prima “c”, si fa pop da entrambi gli stack allo stesso momento. Dopo questo, si rifiuta l’input se si vede qualche “a” o qualche “b”
* pop da entrambe le pile per ogni carattere “c” letto. Se l’input termina quando entrambi gli stack sono vuoti, accetta, altrimenti rifiuta.

Quindi in questo modo, L è riconosciuto dai 2-PDA.

Perciò i 2-PDA sono più potenti degli 1-PDA.

1. I 3-PDA non devono essere più potenti di una TM e quindi se si mostra i 2-PDA possono simulare una Turing Machine, allora i 3-PDA non sono più potenti dei 2-PDA.

Simuliamo una TM su 2-PDA:

* la prima pila memorizza i caratteri a sinistra della testa, con il fondo della pila che memorizza il carattere più a sinistra della TM
* la seconda pila memorizza i caratteri a destra della testa, con il fondo della pila che memorizza il carattere più a destra della TM

Vediamo quindi le transizioni:

* a sinistra, avendo δ(qi, wi)=(qj,wj,L) e poi wi sulla cima della seconda pila

Se ci sta il $ alla cima si rifiuta, altrimenti si procede a sinistra.

In altri casi, si fa pop del simbolo della prima pila facendo push sulla seconda pila. Si cambia poi stato sulla macchina qj. Se qj è stato accettante, allora accetta *w*; viceversa se qj rifiuta, allora rifiuta *w.*

* a destra, avendo δ(qi, wi)=(qj,wj,L) e poi wi sulla cima della seconda pila

Si fa il pop di wi dalla cima della seconda pila, pushando wj nella seconda pila. Se il $ è in cima alla prima pila, si fa il push del blank sulla seconda pila. In entrambi i cas, si camvia lo stato nella macchina qj. Se qj è stato accettante, allora accetta *w*; viceversa se qj rifiuta, allora rifiuta *w.*

In questo modo si dimostra che i 2-PDA riconoscono esattamente la classe dei linguaggi Turing-riconoscibili

1. Per dimostrare che i 3-PDA *non* sono più potenti dei 2-PDA, si considera di simulare una macchina a 4 nastri non deterministici con i 3-PDA.

* il primo nastro è quello di input, mentre gli altri 3 (secondo, terzo, quarto) corrispondono a 3 pile (prima, seconda, terza pila/stack).
* se lo stack si cambia con un push e non con un pop, allora si muove il nastro a destra senza cambia il simbolo corrente. Viene poi scritto il simbolo sul nastro pushato in cima alla pila. In questo caso tutte le cime del nastro sono tenuto eccetto gli stack affetti da push senza pop. Muove a destra per prendere il simbolo blank nel nastro
* se lo stack cambia con un pop e non con un push, N rimpiazza la cima del nastro con un simbolo vuoto e si muove a sinistra
* se lo stack è cambiato da entrambe le operazioni, N cambia il nastro con il simbolo scritto pushato nella cella corrente
* se lo stack non è cambiato, N scrive lo stesso simbolo all’inizio del nastro e poi di nuovo nel nastro stesso

Si usa un simbolo per capire quando N muove il primo nastro a destra. Se non viene usato, allora, N copia gli elementi all’inizio del primo nastro. Questa azioni sono parallele e fatte nello stesso momento in tutti i nastri, dato che possono cambiare una sola vola. N per questi motivi e si nota, è nondeterministica, capendo quale transizione prendere.

Dunque, alla fine di tutto questo, si prova che una TM non è potente quanto i 3-PDA, ma anche che i 3-PDA non sono pi potenti dei 2-PDA. Entrambi sono ugualmente potenti; ciò che cambia è la gestione effettiva dei valori. Ma entrambi sono affetti dal non-determinismo; semplicemente la TM per ciascuna variante numerica di PDA, aggiungerà un nastro, mantenendo di massima lo stesso comportamento dei nastri, risultando sempre ugualmente potente.



L’idea è di definire una TM M tale che accetti la stringa A e stia nel linguaggio. Essendo che ha un solo stato, semplicemente, esso sarà contemporaneamente lo stato iniziale e finale.

Immaginando quindi di avere una TM la simulazione procede in questo modo:

* Simula A sul nastro segnando lo stato iniziale
* Dato che è l’unico stato, la macchina avanza e si aspetta di non segnare nessun altro stato

ad eccezione di quelli che vanno verso lo stato iniziale, che saranno marcati

* Se la macchina non ha segnato altri stati, *accetta*, altrimenti *rifiuta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.



In questo caso abbiamo una CFG che produce una stringa vuota. Optiamo per considerare invece una grammatica che appunto deriva ε:

* se la CFG deriva ε correttamente allora *accetta*
* se la CFG non deriva ε rifiuta

Definiamo M su input <G>, derivando quindi una codifica di una CFG che da un punto di vista costruttivo, sarà G’ ed è in forma normale di Chomsky. Questa grammatica accetta tutte le regole di G, come tale se G possiede una regola del tipo S 🡪 ε, allora anche G’ avrà una regola del tipo S’ 🡪 ε.

Dal punto di vista di una TM M che riconosce questa grammatica:

* marca lo stato iniziale
* si muove sul nastro seguendo le transizioni della CFG
* se essa arriva con le derivazioni ad una stringa vuota (quindi contiene una regola S 🡪 ε, allora *accetta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.



Sono entrambe due espressioni regolari tali che una contenga l’altra come sottoinsieme.

A livello pratico significa che una porzione di linguaggio di R è contenuta in S, quindi entrambe le espressioni hanno almeno una parte dello stesso linguaggio.

Si può quindi idealmente ragionare con una TM N che decide SREX in questo modo:

* trasforma R ed S in due ε-NFA equivalenti B e C
* si esegue N sugli input di R ed S
* se scorrendo tutto il linguaggio di R si arriva, marcando tutti i suoi stati, a coprire almeno una parte di stati di S, anche essi marcati allora *accetta*, altrimenti *rifiuta*

Conviene quindi avere N multinastro, tale da avere gli input da una parte, controllando R su un nastro e vedere sul risultante terzo nastro dove starebbe S se vi è un match del pattern precedente.

Poste queste condizioni, il linguaggio può dirsi *decidibile*.



X per poter essere decidibile deve ammettere che M sia macchina a nastro singolo e non modifichi tale porzione. Significa intuitivamente che potrà modificare tutti il nastro ad eccezione della porzione che contiene “w”. Ovunque sia “w” si marca e tutti gli altri simboli intorno non vengono marcati, mentre tutti gli altri sì. Identifichiamo quindi una TM che chiamiamo P che esegue questi passi:

* simula M su input w
* comincia a scorrere il nastro e, in questa fase, può decidere se marcare o meno i simboli se non vede “w”. Per essere completi diciamo che non marca nessun simbolo, quindi continua a scorrersi il nastro
* appena vede “w” lo marca e torna indietro nel nastro, avanzando fino a quando il nastro non termina

Idealmente, questa macchina potrebbe essere decisore. Ciò include però il fatto che vi sia sempre “w”; nel caso non vi fosse, la macchina andrebbe in loop molto facilmente.

La macchina può quindi essere riconoscitore ed X sarebbe *non decidibile*.



Assumiamo che G sia una TM che ammette un linguaggio Turing-riconoscibile.

A queste condizioni, notiamo anche che la macchina ammette un linguaggio vuoto, simile al test del vuoto, ma complementata. Ciò significa che accetterà tutte le stringhe presenti possibili presenti nel linguaggio, purché siano nell’ordine del linguaggio iniziale e siano complementate.

Se la macchina G però accetta almeno una stringa, la condizione richiesta non vale più, pertanto essa continuerà a scorrere il nastro marcando ogni simbolo terminale, fino a quando non vengono marcate nuove variabili; come per il test del vuoto, se la variabile iniziale non è marcata *accetta*, altrimenti *rifiuta*.

A questo punto sia pensa di costruire M tale che agendo sull’input <M>:

* segua l’ordine delle stringhe dell’altra macchina
* per garantire l’ordine, dopo la prima stringa non vengono più marcate stringhe
* sequenzialmente si scorrono tutte le stringhe della lista della TM M
* in questa macchina se si accetta almeno una stringa allora *accetta*, perché la variabile iniziata è stata marcata, complementalmente all’altra macchina, altrimenti *rifiuta*

Poste queste condizioni, il linguaggio può dirsi *decidibile*.



La notazione “≤m” indica che con un problema risolvibile da A riusciamo a risolvere ogni problema di A.

Un approndimento utile sulla notazione: <https://www.ccs.neu.edu/home/rjw/csu390-f06/HowTo/How-to-Create-Mapping-Reductions.pdf>

Ad esempio, se A fosse definito come linguaggio regolare, come ad esempio 0n1n. A sarà 1n0n, quindi di fatto entrambi riconoscono lo stesso linguaggio, solo “a parti invertite” intuitivamente parlando.

Più formalmente, entrambi sono definiti sullo stesso linguaggio Σ e possiamo esprimerli come linguaggi context-free. Immmaginando quindi di poterli definire come una CFG tale che partendo da L si arrivi ad L’ per mezzo di una grammatica di Chomsky.

Idealmente, partendo da G per A:

S 🡪 01

S 🡪 0S1

possiamo definire una grammatica G’ per A, tale che ne sia esattamente il complemento:

S’ 🡪 10

S’ 🡪 1S0

Essendo in questo modo entrambi i linguaggi context-free ed essendo A un linguaggio con un insieme di stringhe riconoscibili anche da A, allora possiamo costruire una TM tale da poter riconoscere questo linguaggio.

Banalmente ciò si riconosce con una TM che permette, partendo dallo stato iniziale, di riconoscere tutte le derivazioni perché la CFG è in Forma normale di Chomsky, quindi riconosciuto dalla macchina (Turing-riconoscibile). Alla luce di tutto ciò, il linguaggio può dirsi decidibile, tale che la macchina precedente accetti tutte le computazione conducono (per forma di Chomsky) alla derivazione finale.

A è un linguaggio definito da B, ma questo viene definito solo tramite alcune condizioni su A.

Partendo da A dobbiamo dimostrare che presenta una stringa x; se questa stringa è presente nelle transizioni che la TM di riferimento scorrendo trova, allora accetta, altrimenti rifiuta.

B invece viene descritto partendo da una descrizione simile di A, ma ammette anche una stringa “y”.

Quindi oltre ad ammettere una TM di riferimento partendo dalle transizioni percorse per “x”, similmente avremo n passi per “y”. La “y” esiste a condizione di “x”, pertanto se la TM accetta entrambe le stringhe, accetterà il linguaggio. Essendo A un riconoscitore, di sicuro accetterà almeno “x”; la condizione di accettazione su “y” poco cambia per definizione rispetto alla TM precedente. Se x ∈ B allora accetterà “x” con un numero sufficientemente lungo di passi tali da poter accettare anche “y”, pertanto se x ∉ B allora

y ∉ B per tutte le “y”.



Non implica che A sia linguaggio regolare. Per esempio come sopra, definendo A={0n1n|n>=0} e B = {1}

Similmente a sopra, notiamo che A si pone come linguaggio non regolare (ma CF perché esprimibile come visto prima da apposita grammatica, pertanto anche Turing-decidibile), mentre B lo è, quindi coerentemente con la notazione descritta sopra, B è un linguaggio regolare, finito che risolve tutti i problemi di A, ma non si ha necessariamente che se B è regolare allora A lo è.

(Il 26 è uguale al 23, si veda quello per riferimento)

Immaginiamo di avere una TM che riceve in input una stringa w tale che sia un simbolo generico oppure 0x.

A queste condizioni possiamo immaginare una macchina <M, w> che, attraverso una simulazione, scorre tutto il nastro e trova w, in modo tale che accetti quando la trova, rifiuta altrimenti. A livello di output, si constata che <M, w> ∈ ATM, sempre appartenente al linguaggio.

Se è definita la stringa w sul linguaggio ATM similmente è definito il complemento; è facile dimostrare che con una simulazione che agisce esattamente all’opposto avremmo che <M, w> ∈ ATM, sempre appartenente al linguaggio.

L’altro caso invece considera una TM che scrive 1 seguita da una stringa ∈ <M, w>. L’idea è totalmente uguale a prima, come descritto, pertanto la funzione di input considera sempre <M, w> ∈ ATM  🡪 output di

R ∈ J. Dato che ATM ≤m J, similmente abbiamo ATM ≤m J.

Essi non possono essere Turing riconoscibili in quanto già ATM non è riconoscitore (solamente in un caso accetterà esattamente la stringa w, che deve essere 0x e si ferma. Ciò è molto limitato da un punto di vista applicativo; il complemento ragiona nella maniera contrapposta). Tanto ci basta per indicare che J e J non sono Turing-riconoscibili

(Il 28 è uguale a questo appena fatto, si veda questo per riconoscimento)