Appunti di Automi e Linguaggi Formali

Nicola Baesso

June 5, 2022

Contents

Disclaimer

Questi appunti sono una raccolta parziale delle spiegazioni del prof. Bresolin, e sono state scritte con la gioia di poterle portare ai parziali e all'esame. Si consiglia comunque di utilizzare solo come eventuale ripasso e non come testo di studio.
Il materiale utilizzato sono: slide del docente, soluzioni fornite dal docente e soluzioni fornite dal tutor.

1 Introduzione

1.1 Definizioni utili alla comprensione

Per questo corso, ci si avvarrà di alcune definizioni riportate in seguito, utili per una migliore comprensione della materia.

1.1.1 Algoritmi e Problemi

Un problema è definito da 3 (tre) caratteristiche specifiche: l'insieme dei possibili input, l'insieme dei possibili output e la relazione che collega questi due insiemi.
Un algoritmo è una procedura meccanica che, eseguendo delle computazioni eseguibili da un calcolatore, risolve un determinato problema se per ogni input si ferma dopo un numero finito di passaggi e produce un output corretto.
Inoltre è composto da una complessità temporale, che indica il tempo di esecuzione, e una complessità spaziale, che indica la quantità di memoria utilizzata. Entrambe queste misure sono dipendenti dalla dimensione dell'input.

1.1.2 Linguaggi formali

Un linguaggio formale può essere definito come un'astrazione del concetto di problema.
Infatti, un problema può essere espresso sia come un insieme di stringhe (che da qui in avanti indicheremo con Linguaggio), con soluzioni che indicano se una stringa è presente nel Linguaggio o meno, oppure come una trasformazione tra vari Linguaggi, dove la soluzione trasforma una stringa di input in una stringa di output.
Quindi, ogni processo computazionale può essere ridotto ad una determinazione dell'appartenenza ad un insieme di stringhe, oppure essere ridotto ad una mappatura tra insiemi di stringhe.

1.1.3 Automi

Un automa è un dispositivo matematico (inteso in forma astratta) che può determinare l'appartenenza di una stringa ad un Linguaggio e può trasformare una stringa in una seconda stringa. Possiede ogni aspetto di un computer, poichè dato un input provvede a fornire un output, è dotato di memoria e ha la capacità di prendere delle decisioni.
La memoria per un automa è fondamentale. Sostanzialmente esistono automi a memoria finita e automi a memoria infinita, quest'ultimi con accesso limitato e non.
Chiaramente si hanno vari tipi di automi, ognuno di questi adatti ad una determinata classe di linguaggi, dove vengono differenziati per quantità di memoria e per il tipo di accesso ad essa.

1.1.4 Alfabeto e Linguaggio

Esattamente come nel caso del linguaggio naturale, un alfabeto è un insieme finito e non vuoto di simboli, ed è indicato con il simbolo Σ. Da esso si ha la stringa, ovvero una sequenza finita di simboli presi da un alfabeto. Inoltre, si definisce come stringa vuota la stringa senza alcun simbolo preso dall'alfabeto, e si indica con il simbolo ε. Infine la lunghezza di una stringa indica il numero di simboli presenti nella stringa, indicandola con |w|, con w una stringa qualsiasi .

Esempio Si consideri il codice binario, composto da 0 ed 1. Allora definiremo l'alfabeto come Σ={0,1}, una stringa valida come w=010110, e in questo particolare caso |w|=6.

La potenza di un alfabeto, espressa come Σ^k con k>0, esprime l'insieme delle stringhe composte da simboli dell'alfabeto di lunghezza k. L'espressione Σ^* indica l'insieme di tutte le stringhe sull'alfabeto.

Esempio Consideriamo nuovamente il codice binario. Per 0<k<2 (inclusi) si ha
$\Sigma^0=\varepsilon$ (la stringa vuota)
Σ^1={0,1}
Σ^2={00,01,10,11}
E così per ogni k positivo. Mentre $\Sigma^*=\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

Quindi la potenza di un alfabeto crea a sua volta un alfabeto, con stringhe di lunghezza k, e tale alfabeto è composto da 2^k stringhe.

Un linguaggio L è un sottoinsieme dell'insieme di tutte le stringhe dell'alfabeto. In simboli: $L \subseteq \Sigma^*$ per un certo Σ.

1.1.5 Operazioni sui linguaggi

Definiamo le operazioni che si possono fare sui linguaggi:

-Unione: $L \cup M=\{w : w \in L \text{ oppure } w \in M\}$
-Intersezione: $L \cap M=\{w : w \in L \text{ e } w \in M\}$
-Concatenazione: L.M={uv : u ∈ L e v ∈M}
-Complemento: $\overline{L}=\{w : w \notin L\}$
-Chiusura di Kleene (o Star): $L^*=\{w1,w2,...wk : k \geq 0 \text{ e ogni } wi \in L\}$

Tutte questa operazioni godono (e ci fanno godere) della proprietà di chiusura. In altre parole, se L e M sono linguaggi regolari, allora anche $L \cup M$, $L \cap M$, L.M, $\overline{L}$ e L^* sono linguaggi regolari.

Importante! Nelle prossime righe si parleranno di DFA, funzioni di transizioni e quant'altro. Se non si ha la più pallida idea di cosa significhino queste espres-

sioni, v'invito a passare per la sezione DFA del seguente file, e successivamente ritornare sull'esempio che segue.

Esempio Vogliamo (in realtà no, ma ci serve) dimostrare che la proprietà di chiusura vale per l'intersezione (dimostriamo la chiusura per intersezione).
Partiamo dal fatto (che vedremo in dettaglio più avanti) che, essendo L e M linguaggi regolari, esiste un DFA che accetta L e un DFA che accetta M. Quindi, essendo anche L $\cap$ M un linguaggio regolare, possiamo costruire un automa (più precisamente, un DFA) rappresentativo del linguaggio. In particolare, questo automa simula Al che Am in parallelo, e accetta una parola $\Leftrightarrow$ tale parola è accettata sia da Al che da Am.
La funzione di transizione dell'automa è formata da (p,q) $\rightarrow$ (s,t), con p $\rightarrow$ s funzione di transizione di Al, e q $\rightarrow$ t funzione di transizione di Am.
L'automa accetta una parola solo se Al e Am sono entrambi in uno stato finale. In altre parole, con p stato di Al e q stato di Am, l'automa accetta una parola solo se (p,q) è una coppia di stati finali. Formalmente:

AL$\cap$M=(Ql $\times$ Qm, Σ, δL$\cap$M, (ql,qm), Fl $\times$ Fm)
dove il simbolo $\times$ è il prodotto cartesiano.

L'esempio prende in considerazione il linguaggio L$\cap$M, ma il concetto della costruzione dell'automa è valido anche per dimostrare la proprietà di chiusura nelle altre operazioni.

2 DFA, NFA, espressioni e Linguaggi (non) regolari

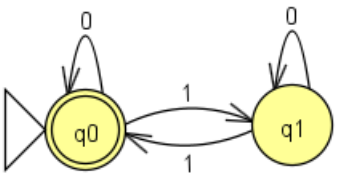
Dopo aver appreso le definizioni necessarie, andiamo a scoprire in questa sezione i primi automi (DFA ed NFA), cosa sono le espressioni regolari e cosa sono i Linguaggi non Regolari.

2.1 DFA

Gli Automi a Stati Finiti (in Inglese Deterministic Finite Automation, abbreviato in DFA) sono la forma più semplice di automa e dispongono di una quantità finita di memoria. Tali automi accettano una parola nel linguaggio unicamente se è in uno stato terminale.

Esempio Trovare ogni parola nel linguaggio binario che sia composta da un numero pari di 1.
Per creare questro DFA, abbiamo bisogno di due stati:
-Il primo sarà lo stato con il numero pari di 1. Tale stato sarà sia iniziale che terminale (e quindi accetterà la stringa).
-Il secondo sarà lo stato nel quale si ha un numero dispari di 1. In tale stato l'automa non accetterà la stringa.

Nel caso s'incontri uno 0, si deve rimanere nello stato attuale.
Segue il diagramma di transizione dell'automa, fatto con JFlap:



Si noti, nel esempio, che ogni stato esegue una transizione **per ogni simbolo nel linguaggio**. Inoltre, quello che fa l'automa è semplicemente "contare" quante cifre di 1 sono presenti, accettando la stringa solo quando questa cifra è pari.
In generale, possiamo dire che un DFA utilizza un numero di stati per "contare" ad un determinato scopo, ed ha una transizione per ogni simbolo del linguaggio che deve analizzare.

In maniera formale, un DFA si definisce con A=(Q,Σ,δ,q0,F), dove Q è un insieme finito di stati, Σ è un alfabeto finito (rappresenta i simboli dati in input al DFA), δ è una funzione di transizione (($q,a) \mapsto q'$), q0 è lo stato iniziale (e logicamente è nell'insieme di tutti gli stati dell'automa. In simboli: $q0 \in Q$), e F è un insieme di stati finali (ovvero gli stati dove l'automa accetta la stringa, in simboli $F \subseteq Q$).

Come già visto nell'esempio, un DFA può essere graficamente rappresentato con cerchi, ad indicare gli stati, e con frecce, ad indicarne le transizioni. Però un DFA può essere rappresentato anche tramite una tabella, chiamata tabella di transizione.

Esempio Riprendiamo l'esempio precedente.
Abbiamo già espresso l'automa che accetta ogni linguaggio con un numero pari di 1 come diagramma di transizione. Ora rappresentiamolo come tabella di transizione.
Per fare ciò, partiamo dallo stato iniziale, nel nostro caso q0, e ci segniamo le sue transizioni: se si ha uno 0, l'automa resta in q0, mentre se si ha un 1 l'automa va nello stato q1. Per q1 si fa la stessa cosa: se si ha uno 0, l'automa resta in q1, mentre con un 1 l'automa va (in questo caso, ritorna) in q0.
Di seguito, in forma tabellare, quel che ci siamo appena detti:

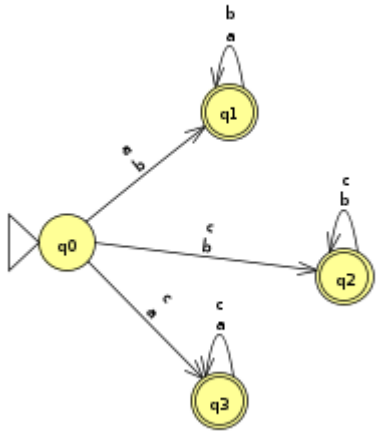
	0	1
$\rightarrow$ q0*	q0*	q1
q1	q1	q0*

Come si nota dall'esempio, nella tabella di transizione si segna con una freccia lo stato iniziale dell'automa, e con un asterisco lo stato finale dell'automa stesso (che nel nostro esempio coincide con lo stato iniziale, ma non è sempre così).
Importante! Un DFA accetta una parola w se la sua computazione accetta w, ovvero l'automa termina in uno stato finale. Inoltre, se il DFA accetta w, allora w appartiene ad un linguaggio regolare.
Più formalmente, si indica con L(A)={w $\in \Sigma^*$ |A accetta w} il linguaggio accettato, che è un linguaggio regolare.

2.2 NFA

Gli Automi a Stati Finiti non Deterministici (in inglese Nondeterministic Finite Automation, abbreviato in NFA) sono una forma di automi a stati finiti, che è utilizzato in contesti dove "contare" non è semplice, quindi si lascia spazio al non-determinismo.
Esempio Costruire un NFA che riconosca la parola che, sull'alfabeto {a,b,c}, non sia composto da tutti i simboli.
Per questo esempio, usare un DFA risulterebbe alquanto complesso, quindi ci avvarremo di un NFA per semplicità. Infatti, ci saranno sufficienti 4 stati e 6 transizioni in tutto. Di seguito troviamo sia tabella che diagramma di transizione per il seguente NFA:

	a	b	c
→ q0	{q1,q3}*	{q1,q2}*	{q2,q3}*
{q1,q2}*	q1*	{q1,q2}*	q2*
{q1,q3}*	{q1,q3}*	q1*	q3*
{q2,q3}*	q3*	q2*	{q2,q3}*
q1*	q1*	q1*	∅
q2*	∅	q2*	q2*
q3*	q3*	∅	q3*

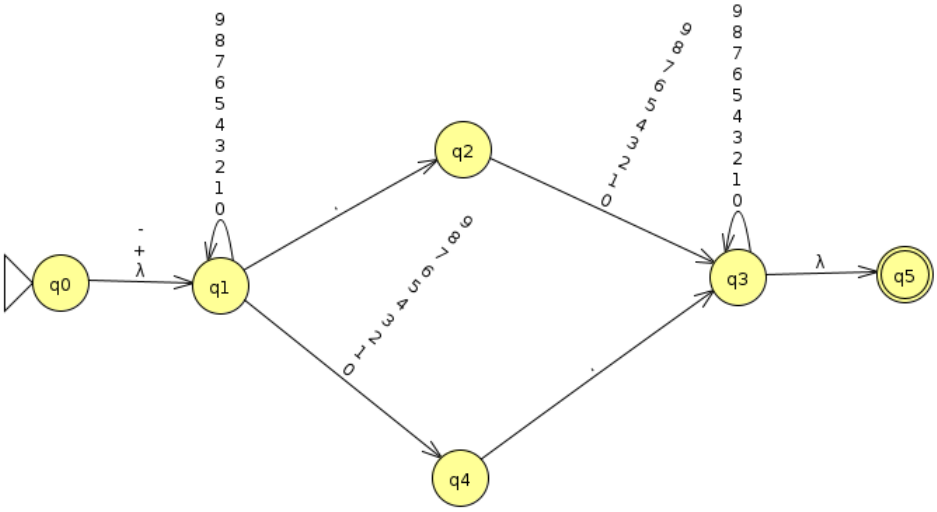


Esattamente come prima, nella tabella rappresentiamo lo stato iniziale con una freccia e con un asterisco rappresentiamo lo stato finale. Generalmente, può capitare che la tabella di transizione può essere "smagrita" ovvero che la tabella completa contiene transizioni a stati mai attraversati dall'automa (non è il caso dell'esempio), e quindi possono essere rimossi. Notiamo però che gli stati inclusi nella tabella non sono semplicemente solo gli stati semplici ma includo anche dei "gruppi" di stati, essendo le transizioni dirette in più stati (grazie al non-determinismo).
Formalizziamo: un NFA NA è descritto con NA=(Q,Σ,δ,q0,F), con l'unica differenza (rispetto ai DFA) che la funzione di transizione restituisce un sottoinsieme di Q.
Importante! Un NFA, a differenza di un DFA, può non arrivare alla fine della computazione ed interrompersi prima (come si nota bene dalle tabelle di transizioni), e quindi accetta una parola w se almeno una sua computazione accetta w, ovvero l'automa termina, in almeno una delle sue computazioni, in uno stato finale. Esattamente come nel DFA, se l'NFA accetta w, allora w appartiene ad un linguaggio regolare (la formalizzazione è la stessa).

2.2.1 ε-NFA

Gli ε-NFA sono NFA che utilizzano le ε-transizioni, ovvero utilizza transizioni che **non consumano il carattere**, quindi l'automa passa allo stato successivo mantenendo la stringa come prima.
Esempio Costruire un ε-NFA che accetti numeri decimali, riconoscendo il segno (opzionale),una stringa di decimali, un punto e un altra stringa di decimali, tenendo conto che una delle stringhe di decimali non può essere vuota.
In questo caso, usare un NFA semplice non è una buona idea, poichè non ci permette di dare opzionalità per quanto riguarda il segno. Inoltre non ci farebbe finire in uno stato finale al termine della lettura dei decimali. Di seguito troviamo sia tabella che diagramma di transizione per il seguente ε-NFA:

	ε	+,-	.	0,..,9
→ q0	q1	q1	$\emptyset$	$\emptyset$
q1	$\emptyset$	$\emptyset$	q2	{q1,q4}
q2	$\emptyset$	$\emptyset$	$\emptyset$	q3
q3	q5*	$\emptyset$	$\emptyset$	q3
q4	$\emptyset$	$\emptyset$	q3	$\emptyset$
q5*	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$



Essendo lo stato {q1,q4} non raggiungibile, si omette nella tabella.
Formalmente, un ε-NFA E è definito come E=(Q,Σ,δ,q0,F), dove l'unico parametro differente è la funzione di transizione, che prende in input uno stato in Q e un simbolo nell'alfabeto $\Sigma \cup \{\varepsilon\}$, restituendo un sottoinsieme di Q.
Essendo un particolare tipo di NFA, l'ε-NFA ha tutte le caratteristiche degli NFA.

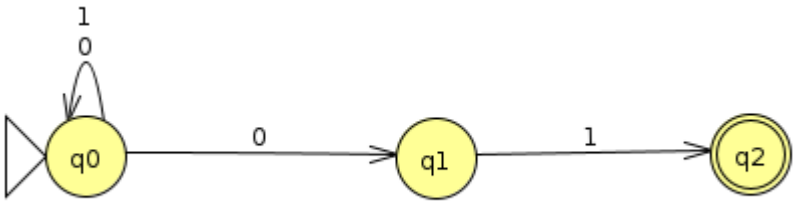
2.3 DFA to NFA (e viceversa)

Sebbene possa sembrare controintuitivo, sia NFA che DFA riconoscono gli stessi linguaggi cambiando unicamente il metodo con cui li riconoscono.
Infatti, $\forall$ NFA N $\exists$ un DFA D : L(D)=L(N) (e viceversa). Per far ciò, si usa un metodo chiamato Costruzione a Sottoinsiemi.

2.3.1 Costruzione a Sottoinsiemi

La Costruzione a Sottoinsiemi ci permette, dato un NFA, di ricavare un DFA che possa riconoscere il linguaggio del NFA. In particolare, ogni stato del DFA corrisponde ad un insieme di stati del NFA, il DFA inizia in {q0}, ovvero nell'insieme di stati con solo q0, nel DFA una stato finale corrisponde ad almeno uno stato finale del NFA, e la funzione di transizione del DFA si riferisce ad uno stato contenuto nel target della funzione di transizione di un NFA.

Esempio Prendiamo il seguente NFA:



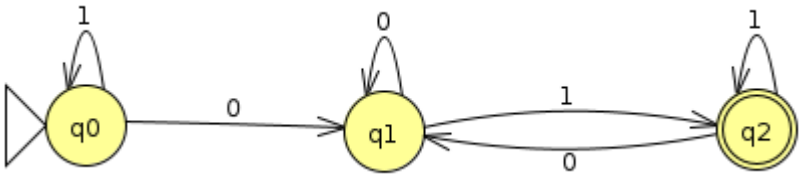
Per "tradurlo" in un DFA, dobbiamo fare una tabella di "conversione", come la seguente:

	0	1
$\rightarrow$ {q0}	{q0,q1}	{q0}
{q1}	$\emptyset$	{q2}*
{q2}*	$\emptyset$	$\emptyset$
{q0,q1}	{q0,q1}	{q0,q1,q2}*
{q0,q2}*	{q0,q1}	{q0,q1}
{q1,q2}*	$\emptyset$	{q2}*
{q0,q1,q2}*	{q0,q1}	{q0,q1,q2}*

Ma se osserviamo, alcuni stati sono inutili, ovvero non contribuiscono in alcun modo alla creazione del DFA, e l'automa non passerà mai per quei stati. Se "puliamo" la tabella otteniamo:

	0	1
$\rightarrow$ {q0}	{q0,q1}	{q0}
{q0,q1}	{q0,q1}	{q0,q1,q2}*
{q0,q1,q2}*	{q0,q1}	{q0,q1,q2}*

Che comprende gli stati che formano realmente il DFA. Se costruiamo il diagramma otteniamo:



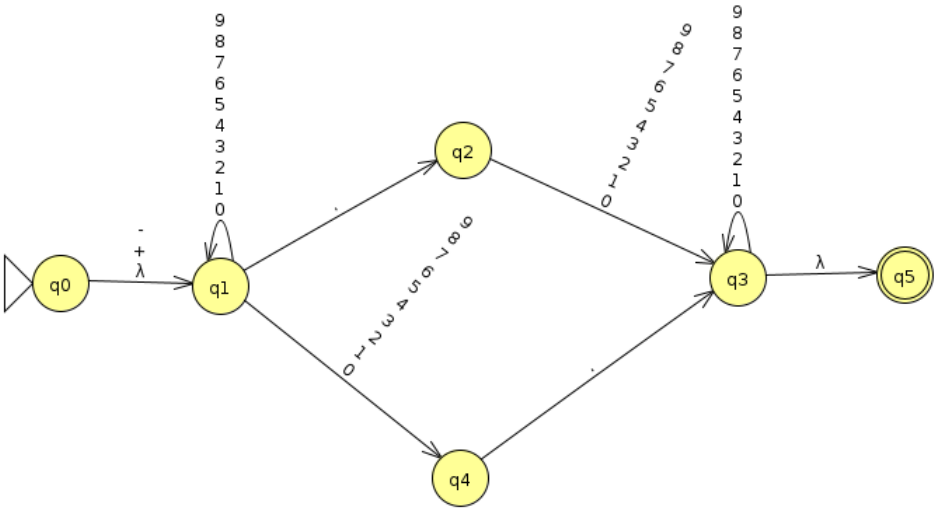
Che è effettivamente corrispondente agli stati e alle transizioni della seconda tabella.
Tale operazione è possibile grazie ad un teorema, che ora enunciamo.

Teorema di equivalenza
Un linguaggio L è accettato da un DFA ⇔ L è accettato da un NFA

2.3.2 ε-chiusure

Nel caso di ε-NFA, si utilizza una versione modificata della costruzione a sottoinsiemi, caratterizzata dalle ε-chiusure. Con ε-chiusura s'intende l'eliminazione delle ε-transizioni dagli stati che ne fanno uso.
A livello pratico. la costruzione a sottoinsiemi è uguale a quella usata partendo da un NFA, con la variante che lo stato iniziale del DFA è dato dalle ε-chiusure dello stato iniziale del ε-NFA.

Esempio Riprendiamo l'ε-NFA visto in precedenza:



Prima di effettuare la costruzione, dobbiamo effettuare le ε-chiusure:

ECLOSE(q0)={q0,q1}
ECLOSE(q1)={q1}
ECLOSE(q2)={q2}
ECLOSE(q3)={q3,q5}
ECLOSE(q4)={q4}
ECLOSE(q5)={q5}
Dopo questa operazione, possiamo applicare la costruzione. Riportiamo solo la tabella di transizione (per pigrizia):

	+,-	.	0,..,9
→ {q0,q1}	{q1}	{q2}	{q1,q4}
{q1}	∅	{q2}	{q1,q4}
{q2}	∅	∅	{q3,q5}*
{q1,q4}	∅	{q2,q3,q5}	{q1,q4}
{q2,q3,q5}*	∅	∅	{q2.q3,q5}
{q3,q5}*	∅	∅	{q3,q5}

Enunciamo anche il teorema per la conversione da ε-NFA a DFA:

Teorema
Un linguaggio L è accettato da un DFA $\Leftrightarrow$ L è accettato da un ε-NFA

2.4 Espressioni Regolari

Abbiamo visto come un linguaggio regolare possa essere espresso tramite un automa a stati finiti. Però non è l'unico modo per descriverlo. Infatti, possiamo esprimerlo anche tramite un'espressione regolare, che non è altro che un modo dichiarativo per descrivere un linguaggio regolare.
Gli ingredienti per costruire un'espressione regolare sono:
-un'insieme di costanti: ε indica la stringa vuota, $\emptyset$ indica il linguaggio vuoto e si usano i simboli presenti in un alfabeto Σ
-operatori: + viene usato per l'unione, . viene usato per la concatenazione e * viene usato per la chiusura di Kleene (* include anche ε)
-parentesi, per il raggruppamento

Con L(E) si indica il linguaggio creato dall'espressione regolare. Essendo una dichiarazione del linguaggio, possono esserci più espressioni corrette per un linguaggio.

Esempio Supponiamo di voler definire come espressione regolare il segeunte linguaggio:

L={w $\in$ {0,1}* : 0 e 1 alternati in w}

Possiamo descriverlo in due modi: o come unione delle possibilità, cioè unione di una alternanza 01, 10, 10101 o 01010, oppure come alternanza di 01, ma con la possibilità di avere un 1 davanti o uno 0.
Poniamolo come espressione, così che risulti più chiaro:

$$(01)^*+(10)^*+1(01)^*+0(10)^*$$
oppure
$$(\varepsilon+1)(01)^*(\varepsilon+0)$$

Si vede come, a livello pratico, non cambi nulla nel significato del linguaggio. Sta quindi a voi scegliere l'espressione che ritenete più opportuna.

Importante! Esattamente come nel linguaggio naturale, le espressioni regolari sono soggette alle precedenze. In particolare, in assenza di parentesi ogni operatore si lega al simbolo alla sua sinistra. Quindi, per fare un esempio, una espressione del tipo 01*+1 viene raggruppata come (0(1)*)+1, e quindi è differente da (01)*+1.

2.5 Equivalenza tra FA e RE

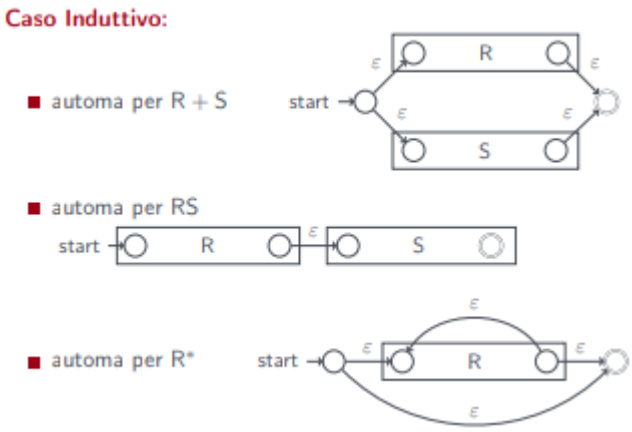
Abbiamo quindi visto finora gli automi a stati finiti (DFA, NFA, ε-NFA) e le espressioni regolari. Ma sappiamo anche che entrambi i modi rappresentano un linguaggio. Tramite questa conoscenza, possiamo passare da un automa (un DFA) ad una espressione regolare e viceversa (ma il quel caso si passerà ad un ε-NFA, come vediamo a breve).

2.5.1 Da RE a ε-NFA

Il primo modo è passare da una espressione regolare ad una ε-NFA. Usiamo il seguente teorema:

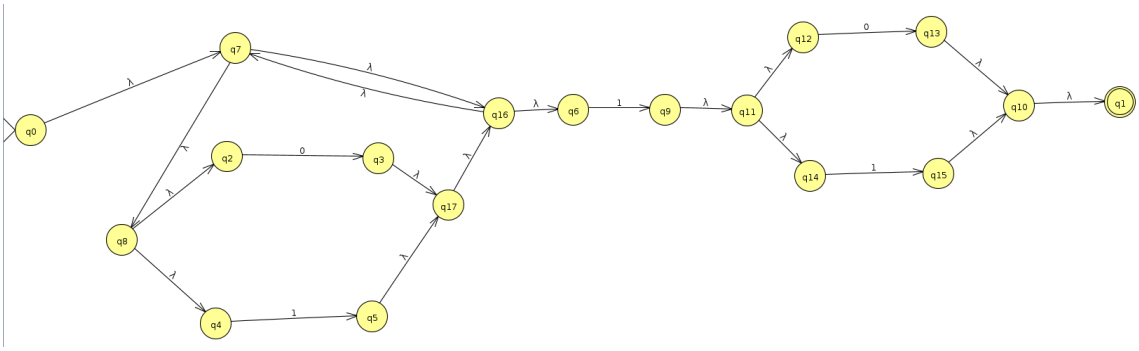
Teorema $\forall$ R espressione regolare possiamo costruire un ε-NFA A : L(A)=L(R)

Per far sì che il ε-NFA riconosca il linguaggio dell'espressione regolare, bisogna costruire un ε-NFA tale che abbia un solo stato finale, non abbia alcuna transizione entrante nello stato iniziale e non abbia nessuna transizione uscente dallo stato finale. Di seguito le regole per rappresentare unione, concatenazione e chiusura di Kleene.



Esempio Consideriamo l'espressione regolare (0+1)*1(0+1).
Notiamo che la prima unione è anche una chiusura di Kleene, quindi nel'automa per R*, il nostro R sarà l'automa che rappresenta l'unione 0+1. Seguirà una ε-transizione ad un automa che conti il primo carattere, il quale avrà un'altra

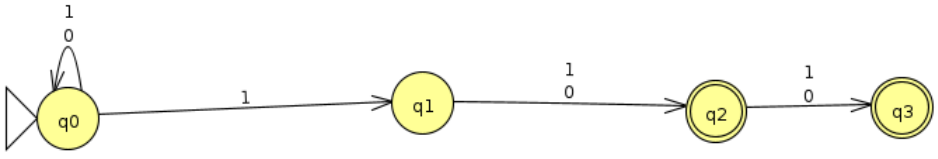
ε-transizione al secondo automa che rappresenta l'unione 0+1. Di seguito riportiamo il diagramma di transizione per l'ε-NFA appena descritto.



Notare che, fondamentalmente, si concatenano più pezzi di automi semplici tramite ε-transizioni.

2.5.2 Da DFA a RE

Se invece si ha un automa e si vuole passare ad una espressione regolare, si utilizza una tecnica chiamata **eliminazione di stati**.
In pratica, quando uno stato viene eliminato, vengono eliminati anche i cammini per quello stato. Al suo posto, si mettono nuove transizioni con espressioni regolari, che descrivono anche le transizioni perse. Si continua finchè non si ha un'unica transizione, contenente il linguaggio riconosciuto dall'automa.
Riassumendo, l'automa di partenza deve avere un unico stato finale (e in caso di più stati finali, se ne crea uno nuovo con ε-transizioni provenienti dai vecchi stati finali), s'iniziano a collassare le transizioni tra la stessa coppia di stati e si eliminano tutti gli stati tranne quello iniziale e quello finale. Se qr $\neq$ q0, ovvero stato finale e stato iniziale sono distinti, allora l'automa ha 4 transazioni: R(che rimane in q0),S(che da q0 va in qr),U(che rimane in qr) e T(che da qr ritorna in q0). L'espressione regolare che lo descrive è (R+SU*T)*SU*. Nel caso q0=qr, l'automa ha un unico stato R e la sua espressione regolare è R*.
Esempio Consideriamo il segeunte automa:



Guardando l'automa, possiamo già intuire che saremo nel caso in cui avremo q0 $\neq$ qr. La prima operazione è creare uno stato q4, che sarà il nostro stato

finale, essendoci 2 stati finali. Collassando le transizioni, otteniamo transizioni di unione, quindi nel nostro caso, in q0 per esempio, invece che una transizione per 0 ed una transizione per 1 otterremo un'unica transizione 0+1. Passando alla eliminazione degli stati, quando elimino q1 creo una transizione che va da q0 a q2, che in espressione regolare si esprime con 1(0+1), quando elimino q2 passo ad una transizione da q0 a q3 che vale 1(0+1)(0+1), mentre la transizione da q0 a q4, essendo 1(0+1) una transizione diretta, è 1(0+1)+1(0+1)(0+1).
Quindi l'espressione regolare relativa all'automa visto è (0+1)*(1(0+1)+1(0+1)(0+1)).

2.6 Linguaggi non regolari

Finora abbiamo visto tutti i possibili modi per mostrare un linguaggio regolare, dall'automa all'espressione regolare. Ma prendiamo in esempio il seguente linguaggio:

$$\text{L01}=\{0^n1^n : \text{n} \geq 0\}$$

Se un DFA con k stati legge 0^k, allora esiste un k+1 stato che è duplicato. Morale: se A legge 1^i e finisce in uno stato finale, accetta la parola 0^j1^i, che non è nel linguaggio, mentre se finisce in uno stato non finale, rifiuta la parola 0^i1^i, che è nel linguaggio, ingannando in ogni caso l'automa. Tali linguaggi sono definiti non regolari.
Il problema è che dire "Non riesco a costruirci un automa a stati finiti" non dice nulla, c'è bisogno di una prova oggettiva, che introduciamo con il teorema del Pumping Lemma.

Teorema Supponiamo che L sia un linguaggio regolare. Allora $\exists$ una lunghezza $\text{k} \geq 0$ tale che ogni parola $\text{w} \in \text{L}$ di lunghezza $|\text{w}|\geq\text{k}$ può essere spezzata in w=xyz, tale che $\text{y} \neq \varepsilon$, $|\text{xy}|\leq \text{k}$ e $\forall\ \text{i} \geq 0$, $\text{xy}^i\text{z} \in \text{L}$.

Quindi, per dimostrare che un linguaggio non è regolare, si usa il Pumping Lemma assumendo L regolare, portandosi in questo modo in una condizione di assurdità (non sempre funziona, ma nella maggior parte dei casi).

3 Grammatiche di Linguaggi liberi da contesto e PDA

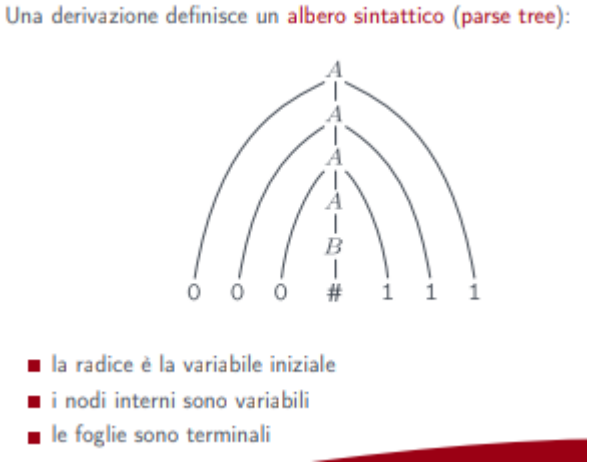
Passiamo ora ad una nuova parte, dove vedremo i inguaggi Context-Free.
Questi linguaggi, a differenza dei linguaggi regolari e non, non dipendono da un contesto, ma anche loro possono essere rappresentati in due modi, o come Grammatica Context-Free o come Automa a Pila, che vedremo successivamente.

3.1 Grammatiche Context-Free

Le Grammatiche Context-Free sono il primo modo per esprimere un linguaggio context-free. Esse utilizzano delle regole di sostituzione (generalmente hanno

forme tipo A ->B), e queste regole possono avere variabili (tra cui una iniziale) o terminali (ovvero simboli dell'alfabeto che si utilizza).
Esempio Vediamo la grammatica G1. Essa utilizza come alfabeto Σ={0,1,#} ed è composta da 3 regole:
-A->0A1
-A->B
-B->#
Un esempio di parola è 000#111.

Come si vede nell'esempio, si parte dalla variabile iniziale (nell'esempio, A), e si segue la regola, finchè non rimangono solo i terminali. Generalmente queste regole sono espresse anche come alberi sintattici, come il seguente:



Formalmente, una grammatica context-free è definita come (V,Σ,R,S), dove V è un insieme finito di variabili, Σ è un insieme finito di terminali disgiunto da V, R è un insieme di regole, S è la variabile iniziale (S $\in$ V).

Ma come si progettano le grammatiche context-free? Fondamentalmente non esistono processi meccanici, però ci sono delle tecniche che possono tornare utili. La prima riguarda l'unione di linguaggi più semplici, essendo molti linguaggi l'unione di linguaggi più semplici. L'idea è di costruire grammatiche separate per ogni componente, per poi unire le due grammatiche con una nuova regola iniziale.
La seconda, valida per i linguaggi regolari, consiste di prendere un DFA e trasformarlo in una grammatica context-free. L'idea sta nell'usare una variabile Ri per ogni stato qi, una regola ri ->aRj per ogni transizione e una regola Ri ->ε per ogni stato finale qi. Se q0 è lo stato iniziale, la variabile iniziale è R0.
La terza consiste nel vedere un linguaggio come formato da due sottostringhe collegate. L'idea in questo caso è usare regole nella forma R->uRv, che generano stringhe dove u corrisponde a v.

3.1.1 Proprietà delle grammatiche context-free

Diamo la definizione di albero sintattico: data una grammatica G=(V,Σ,R,S), un albero sintattico è un albero i cui nodi interni sono variabili di V, le foglie sono simboli terminali o ε e se un nodo interno è etichettato con A e i suoi figli sono, da sinistra a destra, X1,X2...Xk, allora A ->X1X2...Xk è una regola di G.
Attenzione che un albero può generare una stringa in due modi diversi! Infatti, una grammatica genera ambiguamente una stringa se esistono due alberi sintattici diversi per quella stringa. Quindi, generalmente si effettua la derivazione a sinistra per evitare ambiguità.
Diamo una definizione: una grammatica è definita ambigua se genera almeno una stringa ambiguamente.
Inoltre esistono i linguaggi inerentemente ambigui, ovvero linguaggi che sono generati solo da grammatiche ambigue.
Ora, una grammatica context-free è migliore se semplificata, ed una delle forme più semplici è la forma normale di Chomsky, che rende ogni regola della forma A->BC, A->a, dove a è un terminale, B e C non posssono essere la variabile iniziale.
Inoltre, può esserci la regola S ->ε per la variabile iniziale S. Per trasformare una grammatica context-free in forma normale di Chomsky, si aggiunge una nuova variabile, si eliminano le ε-regole, si eliminano le regole unitarie A->B e si trasformano le regole rimaste nella forma corretta appena vista.
Segue un esempio, che non riporto a mano:

Trasformiamo la grammatica G_6 in forma normale di Chomsky:

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

Per trasformare $G = (V, \Sigma, R, S)$ in Forma Normale di Chomsky

1. aggiungiamo una nuova variabile iniziale $S_0 \notin V$ e la regola

$$S_0 \to S$$

In questo modo garantiamo che la variabile iniziale non compare mai sul lato destro di una regola

2. Eliminiamo le ε-regole $A \to \varepsilon$:
   - se $A \to \varepsilon$ è una regola dove A non è la variabile iniziale
   - per ogni regola del tipo $R \to uAv$, aggiungiamo la regola

     $$R \to uv$$

   - attenzione: nel caso di più occorrenze di A, consideriamo tutti i casi: per le regole come $R \to uAvAw$, aggiungiamo

     $$R \to uvAw \mid uAvw \mid uvw$$

   - nel caso di regole $R \to A$ aggiungiamo $R \to \varepsilon$ solo se non abbiamo già eliminato $R \to \varepsilon$
   - Ripeti finché non hai eliminato tutte le ε-regole

3. Eliminiamo le regole unitarie $A \to B$:
   - se $A \to B$ è una regola unitaria
   - per ogni regola del tipo $B \to u$, aggiungiamo la regola

     $$A \to u$$

     a meno che $A \to u$ non sia una regola unitaria eliminata in precedenza
   - Ripeti finché non hai eliminato tutte le regole unitarie

4. Trasformiamo le regole rimaste nella forma corretta:
   - se $A \to u_1 u_2 \dots u_k$ è una regola tale che:
     - ogni u_i è una variabile o un terminale
     - $k \geq 3$
   - sostituisci la regola con la catena di regole

     $$A \to u_1 A_1, \quad A_1 \to u_2 A_2, \quad A_2 \to u_3 A_3, \quad \dots \quad A_{k-2} \to u_{k-1} u_k$$

   - rimpiazza ogni terminale u_i sul lato destro di una regola con una nuova variabile U_i, e aggiungi la regola

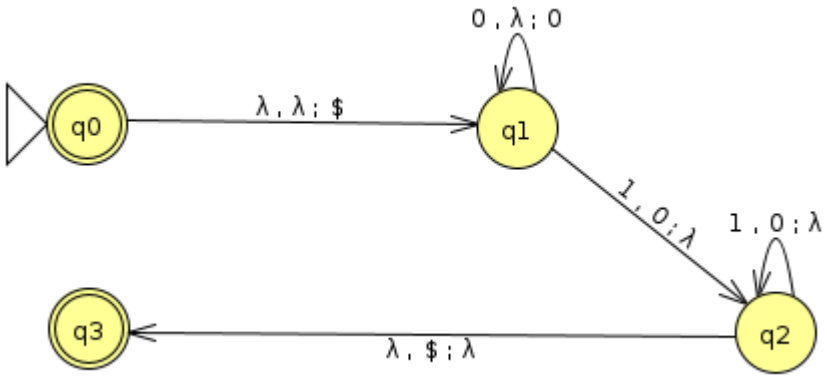
     $$U_i \to u_i$$

   - ripeti per ogni regola non corretta

3.2 PDA

Gli automi a pila (dall'Inglese Pushdown Automation, abbreviato in PDA) sono degli automi che, oltre ad utilizzare gli stati, utilizzano anche una pila. La funzione di transizione stabilisce quali possono essere gli stati successivi e i simboli da scrivere nella pila, dati stato corrente, simbolo in input e il simbolo in cima alla pila. Inoltre, essendo la pila un tipo di memoria LIFO, permette all'automa di avere memoria infinita ad accesso limitato.
Formalmente, un PDA viene definito come P=(Q,Σ,Γ,δ,q0,F), dove Q è un insieme finito di stati, Σ è l'alfabeto di input, Γ è l'alfabeto della pila, δ è la funzione di transizione, q0 è lo stato iniziale e F è l'insieme di stati accettanti.

Esempio Creiamo un PDA per il linguaggio L=$\{0^n1^n : n \geq 0\}$. Formalmente si definisce con P=({q0,q1,q2,q3},{0,1},{0,$},$\delta$,q0,{q0,q3}). Segue diagramma di transizione:



L'esempio appena visto accetta una parola per stato finale, ma un PDA può accettare anche per pila vuota. Infatti, un PDA accetta la parola w per pila vuota se $\exists$ una computazione che consuma tutto l'input e termina con la pila vuota. Inoltre, $\forall$ linguaggio accettato da un PDA per stato finale $\exists$ un PDA che accetta per pila vuota, e viceversa.

3.3 Da Grammatiche Context-Free a PDA

Enunciamo subito un teorema, che ci farà notare una somiglianza tra gli automi a stati finiti e i PDA:
Teorema Un linguaggio è context-free $\Leftrightarrow$ $\exists$ un PDA che lo riconosce
L'idea è che P simula i passi di derivazione in G, e P accetta w se esiste una derivazione di w in G.
La dimostrazione si fa creando un PDA P=({Qstart,Qloop,Qend},Σ,$\Sigma \cup V \cup$ {$},Qstart,Qend) e funzione di transizione ε,A->u per la regola A->u, a,a->ε per il terminale a (dove i vari stati sono presi dalla grammatica).

3.4 Da PDA a Grammatiche Context-Free

Per questo passaggio, andremo a fare una grammatica che fa un po' di più rispetto al PDA, ovvero generiamo una variabile Apq per ogni coppia di stati p,q di P, dove Apq genera ogni stringa che porta da p con pila vuota a q con pila vuota.
Occorre però che il PDA da trasformare rispetti 3 condizioni, ovvero ha un unico stato accettante qf, svuota la pila prima di accettare, e ogni transizione effettua il push o il pop, ma non entrambe le azioni. In particolare, quest'ultima azione prevede due casi, la prima riguarda l'inserimento all'inizio e la rimozione alla fine, la seconda vede un inserimento all'inizio ma una rimozione prima della fine

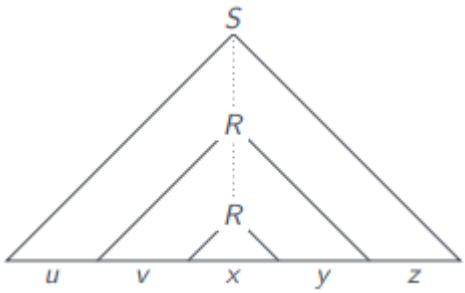
(in r, quindi invece che solo Apq, si avrà sia Apr che Arq).
Formalmente, dato un PDA P=(Q,Σ,Γ,δ,q0,{qf}), generiamo una grammatica G=(V,Σ,R,Aq0-qf) tale che
V={Apq : p,q ∈ Q},
∀ p,q,r,s ∈ Q, u ∈ Γ e a,b ∈ Σε, se δ(p,a,ε) contiene (r,u) e δ(s,b,u) contiene (q,ε), aggiungiamo la regola Apq->aArsb,
∀ p,q,r ∈ Q, aggiungiamo la regola Apq->AprArq,
∀ p ∈ Q, aggiungiamo la regola App->ε.

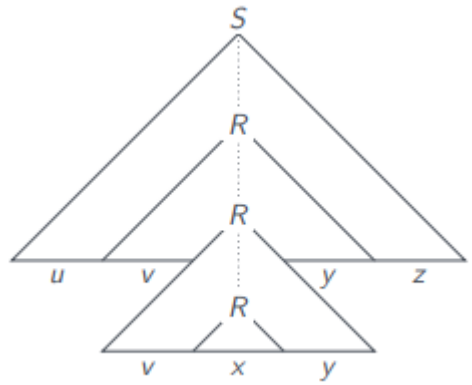
3.5 Linguaggi non context-free

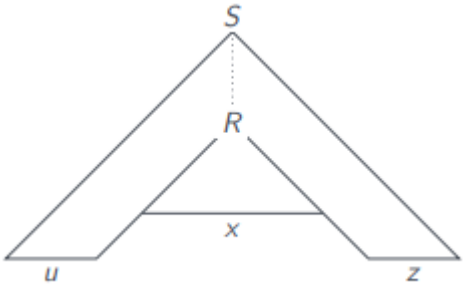
Abbiamo visto, per i linguaggi regolari, che il Pumping Lemma è il modo per dimostrarne la regolarità. Similmente, esiste il Pumping Lemma per linguaggi context-free, di cui ora diamo l'enunciato.

Teorema Sia L un linguaggio context-free. Allora ∃ una lunghezza k ≥ 0 tale che ∀ w ∈ L di lunghezza |w|≥ k può essere spezzata in w=uvxyz tale che |vy|>0, |vxy|≤ k, ∀ i ≥ 0, uv^ixy^iz ∈ L.

L'unica differenza, rispetto ai linguaggi regolari, è che in questo caso la parola viene divisa in 5 parti, e le parti "pompate" sono 2. Questo perché, per i linguaggi context-free, si replica il sottoalbero (ripetuto) di R rimanendo nel lingauggio (a parità di w "molto lunga" e albero sintattico relativo "molto alto"). Di seguito una dimostrazione grafica.





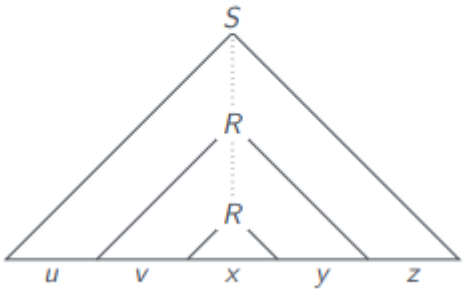


Includiamo inoltre proprietà degli alberi e dimostrazione, sempre in forma grafica.

- Sia G una grammatica per il linguaggio L
- Sia b il numero massimo di simboli nel lato destro delle regole
- In un albero sintattico, ogni nodo avrà al massimo b figli:
  - al più b foglie per un albero di altezza 1
  - al più b^2 foglie per un albero di altezza 2
  - al più b^h foglie per un albero di altezza h
- Un albero di altezza h genera una stringa di lunghezza minore o uguale a b^h
- Viceversa: una stringa di lunghezza maggiore o uguale a $b^h + 1$ richiede un albero sintattico di altezza maggiore di h

- Prendiamo come lunghezza del pumping $k = b^{|V|+1}$ ($|V|$ numero di variabili in G)
- Presa una stringa $w \in L$, se $|w| \geq k$ allora ogni albero sintattico per w deve avere altezza maggiore o uguale a $|V|+1$
- Prendiamo più piccolo albero sintattico τ per w
- Prendiamo il cammino più lungo in τ: sarà di lunghezza $\geq |V|+2$
- Quindi ci sono almeno $|V|+1$ variabili nel cammino
- Qualche variabile R si ripete nel cammino
- Scegliamo R in modo che si ripeta nei $|V|+1$ nodi più in basso nel cammino

- La variabile R ci dà la suddivisione $w = uvxyz$:
  - il sottoalbero più in alto genera vxy
  - il sottoalbero più in basso genera x



- possiamo sostituire i due sottoalberi tra loro, ottenendo di nuovo un albero sintattico corretto:
  - sostituire ripetutamente il più piccolo con il più grande ci dà uv^ixy^iz per ogni $i > 1$
  - sostituire il più grande con il più piccolo ci dà uxz
- in tutti i casi la nuova stringa appartiene a L
- rimane da dimostrare:
  - che $|vy| > 0$ (v e y non possono essere entrambi la parola vuota)
  - che $|vxy| \leq k$

- supponiamo che $v = \varepsilon$ e $y = \varepsilon$
- allora se sostituiamo il sottoalbero più grande con il più piccolo otteniamo di nuovo w:

$$w = uvxyz = u\varepsilon x\varepsilon z = uxz$$

- Assurdo: avevamo scelto τ come l'albero sintattico più piccolo!

- l'occorrenza più in alto di R genera vxy
- avevamo scelto le occorrenze tra i $|V| + 1$ nodi più in basso
- il sottoalbero che genera vxy è alto al massimo $|V| + 1$
- quindi può generare una stringa di lunghezza al più $b^{|V|+1} = k$

FINE!

4 Decidibilità e macchine di Turing

Finora si sono visti sia automi a stati finiti (DFA, NFA, ε-NFA), sia automi a pila (PDA). Questi modelli hanno però una memoria limitata, o accessibile per un numero finito di volte. In questa sezione, vedremo un nuovo modello di automi, chiamati Macchine di Turing.

4.1 Macchine di Turing

Le Macchine di Turing (dall'inglese Turing Machines, abbreviato TM) altro non sono che un modello di calcolo proposto da (indovinate?) Alan Turing, nel 1936.

Tali macchine hanno una memoria illimitata e senza restrizioni, fondamentalmente rappresenta ogni calcolo che un computer reale può fare. Da ciò se ne deduce che se una TM non può risolvere un determinato problema, tale problema non può essere risolto da un computer poichè ne supera le capacità.

Si è detto che una TM ha memoria infinita e con accesso illimitato. Infatti, la memoria è rappresentata da un nastro infinito, con inizialmente l'input scritto su esso, dove una testina ci legge e ci scrive i simboli. Tale testina può muoversi in qualunque direzione, e legge stati speciali sia per l'accettazione che per il rifiuto di una parola, che hanno effetto immediato. Formalmente, si definisce in

questo modo una TM:

M=(Q,Σ,Γ,δ,q0,qAccept,qReject), dove Q è l'insieme finito di stati, Σ è l'alfabeto di input e non contiene $\sqcup$, Γ è l'alfabeto del nastro, contiene sia il simbolo di blanck che i simboli presenti in Σ, δ è la funzione di transizione Q x Γ $\mapsto$ Q x Γ x {L,R}, q0 è lo stato iniziale, qAccept è lo stato accettante e qReject è lo stato di rifiuto, diverso da qAccept (ovviamente q0,qAccept,qReject $\in$ Q).

Inoltre, una TM viene descritta da una configurazione, ovvero una tripla uqv, dove q è lo stato corrente, u è il contenuto del nastro prima della testina e v è il contenuto del nastro dalla testina in poi. La testina è posizionata sul primo simbolo di v. Una configurazione C1 produce C2 se la TM in questione, in un passo, passa da C1 a C2.
Esempio Prendiamo in esame a,b,c $\in$ Γ e u,v $\in$ Γ^*, con qi,qj stati. Allora:
Se δ(qi,b)=(qj,c,L), la configurazione uaq$_i$bv produce uq$_j$acv,
se δ(qi,b)=(qj,c,R), la configurazione uaq$_i$bv produce uacq$_j$v.ì

Per quanto riguarda le fasi iniziali e finali, q0w è la configurazione iniziale (con w l'input), una configurazione con stato qAccept è una configurazione di accettazione e una configurazione con stato qReject è una configurazione di rifiuto. Le configurazioni di accettazione e di rifiuto sono configurazioni di arresto.

Introduciamo ora i concetti di linguaggi Turing-riconoscibili e linguaggi Turing-decidibili. Un linguaggio è **Turing-riconoscibile** se $\exists$ una TM che lo riconosce, ovvero $\forall$ i,j=1,...,k C1 è la configurazione iniziale, ogni Ci produce Ci+1 e la configurazione di accettazione è Ck.
Un linguaggio è invece **Turing-decidibile** se $\exists$ una TM che lo decide, ovvero se dando un input M alla TM, essa termina sempre la computazione.

Esempio Introduciamo un esempio di TM. Supponiamo di avere il seguente linguaggio:

$$A=\{0^{2^n} : n \geq 0\}$$

Tale linguaggio ha una TM che decide il linguaggio A. In particolare, scorre il nastro da sinistra a destra, cancellando ogni secondo 0. Se il nastro conteneva un solo 0, accetta. Se invece il nastro conteneva un numero dispari di 0, rifiuta. Tale operazioni sono ripetute nel caso non si verifichino i due casi precedenti.

4.1.1 Varianti

Oltre alla definizione che abbiamo appena osservato, esistono delle definizioni differenti di TM, che chiameremo varianti.
La prima è la **TM a nastro semi-infinito**, dove la testina è posizionata all'inizio del nastro e la testina è infinita solo verso destra. Ovviamente la testina può muoversi esattamente come in una TM classica, con l'unica eccezione che se richiede uno spostamento a sinistra a inizio nastro, la testina non si muove.

Segue il teorema:

Teorema: $\forall$ TM a nastro semi-infinito $\exists$ una TM a nastro infinito equivalente (e viceversa).

Seguono ora le **TM multinastro**, dove una TM ha k nastri semi-infiniti. Logicamente, utilizza anche k testine per la lettura e la scrittura dei nastri. Il funzionamento differisce leggermente, poichè l'input per la TM si trova sul primo nastro, e ad ogni passo scrive e si muove simultaneamente su tutti i nastri. Prima di introdurre il teorema, dobbiamo parlare della sua funzione di transizione:

δ: Q x $\Gamma^k \rightarrow$ Q x Γ^k x {L,R}^{2}

Se lo stato è qi e le testine leggono a1,...,ak allora scriviamo b1,...,bk sui k nastri, e poi muovendo ogni testina a sinistra o destra come specificato.

Teorema: $\forall$ TM multinastro $\exists$ una TM a singolo nastro equivalente.
L'idea è di creare una TM dove il nastro è formato dai k nastri, separati dal carattere #, e indicando con un punto la posizione della testina.

Possiamo anche definire un **Corollario**: Un linguaggio è Turing-riconoscibile $\Leftrightarrow \exists$ una TM multinastro che lo riconosce.
Tale corollario è valido poichè abbiamo definito una TM multinastro come una variante di una TM regolare.

Passiamo ora alle **TM non deterministiche**, dove sono possibili più strade durante la computazione. Per semplicità considereremo delle TM a nastro semi-infinito. Per queste TM, la computazione è un albero decisionale, e la TM accetta se $\exists$ un ramo che porta allo stato accettante. Anche in questo caso abbiamo bisogno di una nuova funzione di transizione:

δ: Q x $\Gamma \rightarrow 2^{(Qx\Gamma x\{L,R\})}$

Quindi questa computazione richiede l'esaminazione di ogni ramo, fino a quello di accettazione (se esiste). Segue il teorema:

Teorema: $\forall$ TM non deterministica $\exists$ una TM deterministica equivalente.
In queste TM, il terzo nastro (in una TM a 3 nastri) tiene traccia delle scelte non deterministiche.

Anche con queste TM possiamo definire un **Corollario**: Un linguaggio è Turing-riconoscibile $\Leftrightarrow \exists$ una TM non deterministica che lo riconosce. Anche in questo caso, una TM non determinista può essere trasformata in una TM deterministica, rendendo valido il corollario.

Vediamo ora gli **Enumeratori**, ovvero TM che usufruiscono anche di una stampante. Un linguaggio è detto enumerato se ne ha stampato tutte le stringhe, sono ammesse ripetizioni.
Ne segue subito il **Corollario**: Un linguaggio è Turing-riconoscibile $\Leftrightarrow \exists$ un enumeratore che lo enumera.

Concludiamo con le **TM monodirezionali**, dove la macchina non può spostare la sua testina a sinistra, ma può tenerla ferma o spostarla a destra. Funzione di transizione: δ: Q x $\Gamma \rightarrow$ Q x $\Gamma\{$S,R$\}$.

Esistono poi altri modelli, che non affrontiamo. Quel che è rilevante sono gli elementi in comune, ovvero l'accesso senza restrinzioni ad una memoria illimitata, e l'equivalenza tra i vari modelli.

4.1.2 Algoritmi

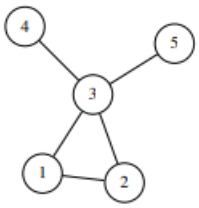
Parliamo ora degli algoritmi per le TM. O meglio, di come possiamo descrivere la procedura con cui una TM riconosce, o decide, un determinato linguaggio.
Essenzialmente esistono 3 modi: la descrizione formale, la descrizione implementativa e la descrizione di alto livello.
Nella **descrizione formale** ogni singolo elemento è descritto in maniera dettagliata, e tutto è dichiarato esplicitamente. Questo metodo è sconsigliato poichè alle volte non è possibile descrive tutto dettagliatamente, ed è da evitare il più possibile.
Nella **descrizione implementativa** non vengono espressi dettagli sugli stati, e vengono espressi a parole il movimento della testina e la scrittura sul nastro. Questa descrizione è più tollerata rispetto alla precedente, ma non è la descrizione ottimale.
Nella **descrizione ad alto livello**, l'algoritmo viene descritto a parole e non viene fornito alcun dettaglio implementativo. Questa descrizione è quella da preferire sempre, tranne quando specificato di usarne una diversa.
Segue un esempio:

I grafi e le loro applicazioni

- I grafi sono strutture dati che vengono usate estensivamente in informatica
- Ci sono migliaia di problemi computazionali che sono importanti per le applicazioni e che si possono modellare con i grafi.
- In questa lezione vedremo che cos'è un grafo, e studieremo alcuni problemi sui grafi che sono interessanti per la loro classe di complessità.

Definizioni di base

Un grafo è definito da un'insieme di nodi (o vertici) e da un'insieme di archi che collegano i nodi.



Definition (Grafo non orientato)

Un grafo non orientato (detto anche indiretto) G è una coppia (V, E) dove:

- $V = \{v_1, v_2, \ldots, v_n\}$ è un insieme finito e non vuoto di vertici;
- $E \subseteq \{\{u, v\} \mid u, v \in V\}$ è un insieme di coppie non ordinate, ognuna delle quali corrisponde ad un arco non orientato del grafo.

Un problema di connessione

- Un grafo è connesso se ogni nodo può essere raggiunto da ogni altro nodo tramite gli archi del grafo

Problema

Il linguaggio $A = \{\langle G \rangle \mid G \text{ è un grafo connesso}\}$ è decidibile?

Una TM che decide A

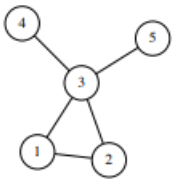
Descrizione di alto livello:

M = "Su input $\langle G \rangle$, la codifica di un grafo G:

1. Seleziona il primo nodo di G e lo marca.
2. Ripeti la fase seguente fino a quando non vengono marcati nuovi nodi:
3. per ogni nodo in G, marcalo se è connesso con un arco ad un nodo già marcato.
4. Esamina tutti i nodi di G: se sono tutti marcati, accetta, altrimenti rifiuta."

Codifica del grafo

- Codifica di G: lista dei nodi + lista degli archi

$G =$ 

$\langle G \rangle$ = `(1,2,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))`

- M verifica che l'input sia sia una codifica di un grafo:
  - Se l'input non è nella forma corretta, rifiuta
  - Se l'input codifica un grafo, prosegue con la fase 1

14 of 14

4.2 Linguaggi decidibili

In questa sezione, si tratteranno quali sono i linguaggi decidibili, ovvero se i linguaggi sono risolvibili o meno mediante un algoritmo.
Iniziamo vedendo i **problemi sui linguaggi regolari**. Una TM converta un DFA, un NFA o una RE in un'altra codifica. L'importante è mostrare che il linguaggio è decidibile mostrando che il problema computazione è decidibile. Di seguito alcuni esempi:

Il problema dell'accettazione

- Problema dell'accettazione: testare se un DFA accetta una stringa

$$A_{DFA} = \{\langle B, w \rangle \mid B \text{ è un DFA che accetta la stringa } w\}$$

- B accetta w se e solo se $\langle B, w \rangle$ appartiene ad A_{DFA}
- Mostrare che il linguaggio è decidibile equivale a mostrare che il problema computazionale è decidibile

4 of 22

Teorema: A_{DFA} è decidibile

Idea: definire una TM che decide A_{DFA}

$M =$ "Su input $\langle B, w \rangle$, dove B è un DFA e w una stringa:

1. Simula B su input w
2. Se la simulazione termina in uno stato finale, accetta. Se termina in uno stato non finale, rifiuta."

Dimostrazione:

- la codifica di B è una lista dell componenti Q, Σ, δ, q_0 e F
- fare la simulazione è facile

Teorema: A_{NFA} è decidibile

$$A_{NFA} = \{\langle B, w \rangle \mid B \text{ è un } \varepsilon\text{-NFA che accetta la stringa } w\}$$

Idea: usiamo la TM M che decide A_{DFA} come subroutine

Dimostrazione:

$N =$ "Su input $\langle B, w \rangle$, dove B è un ε-NFA e w una stringa:

1. Trasforma B in un DFA equivalente C usando la costruzione per sottoinsiemi
2. Esegui M con input $\langle C, w \rangle$
3. Se M accetta, accetta; altrimenti, rifiuta."

N è un decisore per A_{NFA}, quindi A_{NFA} è decidibile

Teorema: A_{REX} è decidibile

$A_{REX} = \{\langle R, w\rangle \mid R \text{ è una espressione regolare che genera la stringa } w\}$

Idea: usiamo la TM N che decide A_{NFA} come subroutine

Dimostrazione:
$P =$ "Su input $\langle R, w\rangle$, dove R è una espressione regolare e w una stringa:

1. Trasforma R in un ε-NFA equivalente C usando la procedura di conversione
2. Esegui N con input $\langle C, w\rangle$
3. Se N accetta, accetta; altrimenti, rifiuta."

P è un decisore per A_{REX}, quindi A_{REX} è decidibile

Test del vuoto

- Negli esempi precedenti dovevamo decidere se una stringa appartenesse o no ad un linguaggio
- Ora vogliamo determinare se un automa finito accetta una qualche stringa

$$E_{DFA} = \{\langle A \rangle \mid A \text{ è un DFA e } L(A) = \emptyset\}$$

- Puoi descrivere un algoritmo per eseguire questo test?

E_{DFA} è decidibile

Dimostrazione: verifica se c'è uno stato finale che può essere raggiunto a partire dallo stato iniziale.

T = "Su input $\langle A \rangle$, la codifica di un DFA A:

1. Marca lo stato iniziale di A.
2. Ripeti la fase seguente fino a quando non vengono marcati nuovi stati:
3. marca ogni stato di A che ha una transizione proveniente da uno stato già marcato.
4. Se nessuno degli stati finali è marcato, accetta; altrimenti rifiuta."

Test di equivalenza

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ e } B \text{ sono DFA e } L(A) = L(B)\}$$

Idea:

- costruiamo un DFA C che accetta solo le stringhe che sono accettate da A o da B, ma non da entrambi
- se $L(A) = L(B)$ allora C non accetterà nulla
- il linguaggio di C è la differenza simmetrica di A e B

EQ_{DFA} è decidibile

Dimostrazione:

- la differenza simmetrica di A e B è:

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

- i linguaggi regolari sono chiusi per unione, intersezione e complementazione
- F = "Su input $\langle A, B\rangle$, dove A e B sono DFA:
  1. Costruisci il DFA C per differenza simmetrica
  2. Esegui T, la TM che decide E_{DFA} con input $\langle C\rangle$
  3. Se T accetta, accetta; altrimenti, rifiuta."

Vediamo ora i **problemi sui linguaggi Context-free**, mostrando direttamente gli esempi:

Problema dell'accettazione

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ è una CFG che genera la stringa } w\}$$

Idea: costruiamo una TM che provi tutte le derivazioni di G per trovarne una che genera w

Perché questa strategia non funziona?

A_{CFG} è decidibile

- Se la CFG è in forma normale di Chomsky, allora ogni derivazione di w è lunga esattamente $(2|w| - 1)$ passi
- Le TM possono convertire le grammatiche nella forma normale di Chomsky!

Dimostrazione:

$S =$ "Su input $\langle G, w\rangle$, dove G è una CFG e w una stringa:

1. Converti G in forma normale di Chomsky
2. Elenca tutte le derivazioni di $2|w| - 1$ passi. Se $|w| = 0$, elenca tutte le derivazioni di lunghezza 1
3. Se una delle derivazioni genera w, accetta; altrimenti rifiuta."

Test del vuoto

$$E_{CFG} = \{\langle G \rangle \mid A \text{ è una CFG ed } L(G) = \emptyset\}$$

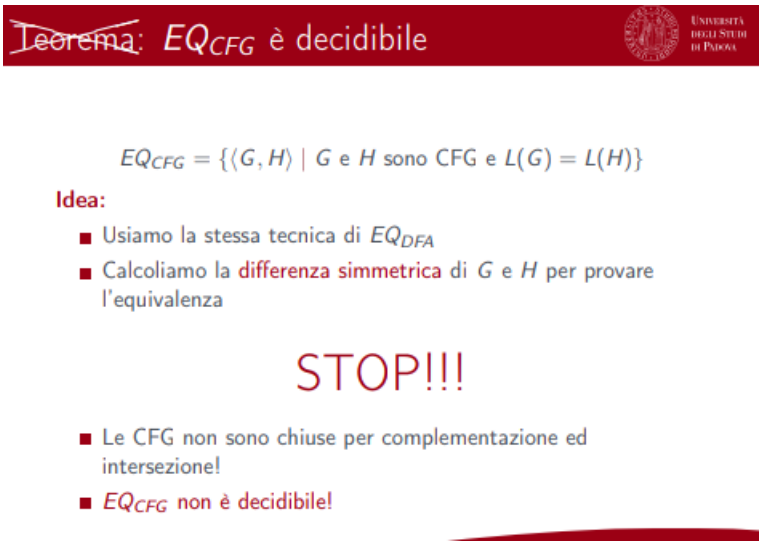
- Problema: non possiamo usare S del teorema precedente. *Perché no?*
- Bisogna procedere in modo diverso!

Teorema: E_{CFG} è decidibile

Idea: stabilisci per ogni variabile se è in grado di generare una stringa di terminali

R = "Su input $\langle G \rangle$, la codifica di una CFG G:

1. Marca tutti i simboli terminali di G.
2. Ripeti la fase seguente fino a quando non vengono marcate nuove variabili:
3. marca ogni variabile A tale che esiste una regola $A \to U_1 \dots U_k$ dove ogni simbolo $U_1 \dots U_k$ è già stato marcato.
4. Se la variabile iniziale non è marcata, accetta; altrimenti rifiuta."



Concludiamo dando la relazione tra le classi di linguaggi:

Regolari $\subseteq$ Context-free $\subseteq$ Decidibili $\subseteq$ Turing-riconoscibili

Queste non sono solo classi di linguaggi, ma anche classi di capacità computazionale.

4.3 Indecidibilità e riducibilità

Ora passiamo alla indecidibilità e riducibilità di un algoritmo, ovvero come dimostrare se un algoritmo è indecidibile, ovvero non può essere risolto algoritmicamente, e un metodo per la dimostrazione, ovvero la riducibilità.

4.3.1 Indecidibilità

Essendo quasi tutto legato ad esempi, si riportano le immagini.

Teorema: A_{TM} è indecidibile

$A_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che accetta la stringa } w\}$

- Chiarimento: A_{TM} è Turing-riconoscibile
- Conseguenza: i riconoscitori sono più potenti dei decisori
- U = "Su input $\langle M, w\rangle$, dove M è una TM e w una stringa:
  1. Simula M su input w
  2. Se la simulazione raggiunge lo stato di accettazione, accetta; se raggiunge lo stato di rifiuto, rifiuta."
- U è un riconoscitore. Perché non è un decisore?

Il metodo della diagonalizzazione

- Dimostrare che A_{TM} è indecidibile usa la diagonalizzazione
- Metodo scoperto da Cantor nel 1873
- Serve per confrontare le dimensioni di insiemi infiniti

Idea: due insiemi finiti hanno la stessa dimensione se gli elementi di un insieme possono essere accoppiati agli elementi dell'altro insieme.

Corrispondenze

- Abbiamo due insiemi A e B e una funzione $f : A \mapsto B$
- f è iniettiva se non mappa mai elementi diversi nello stesso punto: $f(a) \neq f(b)$ ogniqualvolta che $a \neq b$
- f è suriettiva se tocca ogni elemento di B: per ogni $b \in B$ esiste $a \in A$ tale che $f(a) = b$
- Una funzione iniettiva e suriettiva è chiamata biettiva: è un modo per accoppiare elementi di A con elementi di B

Definition

A e B hanno la stessa cardinalità se esiste una funzione biettiva $f : A \mapsto B$

Corollario

- L'insieme di tutte le macchine di Turing è numerabile
- L'insieme di tutti i linguaggi è non numerabile
- Devono esistere linguaggi non riconoscibili da una macchina di Turing

$A_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che accetta la stringa } w\}$

Dimostrazione:

- per contraddizione. Assumiamo A_{TM} decidibile per poi trovare una contraddizione
- Supponiamo H decisore per A_{TM}
- Cosa fa H con input $\langle M, w\rangle$?

$$H(\langle M, w\rangle) = \begin{cases} \text{accetta} & \text{se } M \text{ accetta } w \\ \text{rifiuta} & \text{se } M \text{ non accetta } w \end{cases}$$

Teorema: A_{TM} è indecidibile

- Definiamo una TM D che usa H come subroutine
- D = "Su input $\langle M\rangle$, dove M è una TM:
  1. Esegue H su input $\langle M, \langle M\rangle\rangle$
  2. Dà in output l'opposto dell'output di H. Se H accetta, rifiuta; se H rifiuta, accetta."
- Cosa fa H con input $\langle D\rangle$?

$$D(\langle D\rangle) = \begin{cases} \text{accetta} & \text{se } D \text{ non accetta } \langle D\rangle \\ \text{rifiuta} & \text{se } D \text{ accetta } \langle D\rangle \end{cases}$$

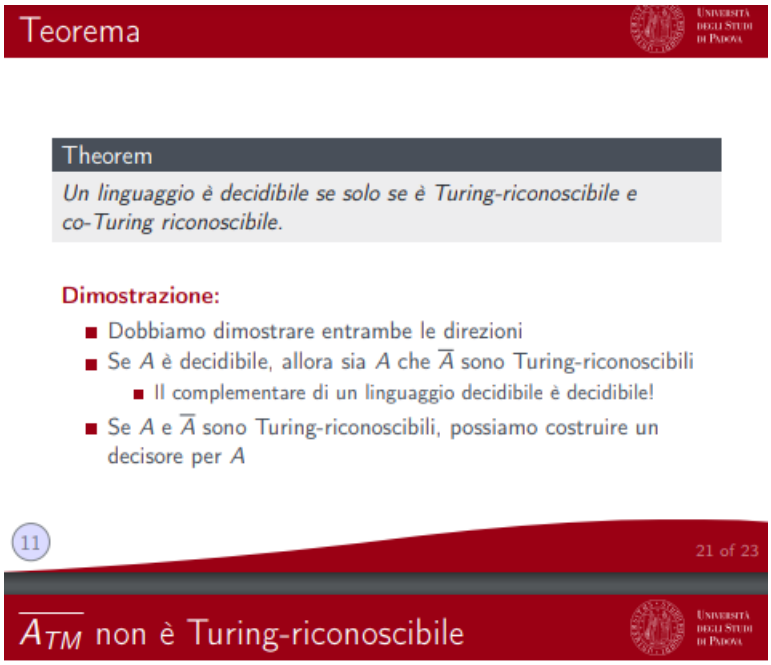
- Contraddizione!

Un linguaggio non Turing-riconoscibile

- Abbiamo visto che A_{TM} è Turing-riconoscibile
- Sappiamo che l'insieme di tutte le TM è numerabile
- Sappiamo che l'insieme di tutti i linguaggi è non numerabile
- Di conseguenza deve esistere un linguaggio non Turing-riconiscibile

Un linguaggio non Turing-riconoscibile

- C'è ancora una cosa che dobbiamo fare prima di poter mostrare un linguaggio non Turing-riconoscibile.
- Mostreremo che se un linguaggio e il suo complementare sono Turing-riconoscibili, allora il linguaggio è decidibile.
- Un linguaggio è co-Turing riconoscibile se è il complementare di un linguaggio Turing-riconoscibile



4.3.2 Riducibilità

La riduzione permette la trasformazione di un problema di un altro problema, di cui si può ricavare la soluzione, in modo da risolvere il problema originale.
In tal modo, prendendo A e B due problemi distinti, se A è riducibile a B e B è decidibile, allora anche A è decidibile. Mentre, se A è riducibile a B e A è indecidibile, allora pure B è indecidibile.
Tale riduzione può avvenire anche mediante una funzione. Seguono rappresentazione grafica e relativi esercizi/teoremi.

Il problema del vuoto

$$E_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide E_{TM}
- Useremo R per costruire la TM S che decide A_{TM}

Stabilire se un linguaggio è regolare

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) \text{ è regolare}\}$$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide $REGULAR_{TM}$
- Useremo R per costruire la TM S che decide A_{TM}
- Capire come possiamo usare R per implementare S è meno ovvio di prima

Il problema dell'equivalenza

$EQ_{TM} = \{\langle M_1, M_2 \mid M_1, M_2 \text{ TM tali che } L(M_1) = L(M_2)\}$

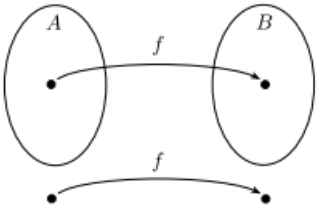
- La dimostrazione è per contraddizione e riduzione di E_{TM} (problema del vuoto)
- Chiamiamo R la TM che decide EQ_{TM}
- Useremo R per costruire la TM S che decide E_{TM}

Definizione

Definition

Un linguaggio A è riducibile mediante funzione al linguaggio B ($A \leq_m B$), se esiste una funzione calcolabile $f : \Sigma^* \mapsto \Sigma^*$ tale che

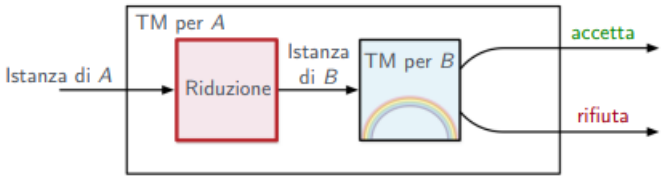
per ogni w : $w \in A$ se e solo se $f(w) \in B$



f è la riduzione da A a B

Riducibilità mediante funzione

Se esiste una riduzione da A a B, possiamo risolvere A usando una soluzione per B:



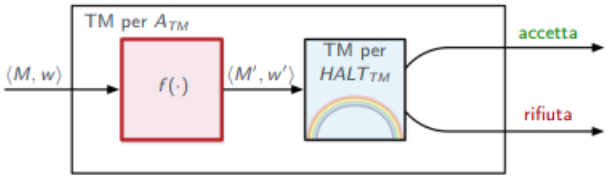
Il problema della fermata (2)

$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che si ferma su input } w\}$

- Possiamo dimostrare che $A_{TM} \leq_m HALT_{TM}$?
- Qual è l'input della funzione di riduzione?
- Qual è l'output?
- Quali proprietà devono rispettare?

2-4

$A_{TM} \leq_m HALT_{TM}$?



M accetta w se e solo se M' si ferma su w'

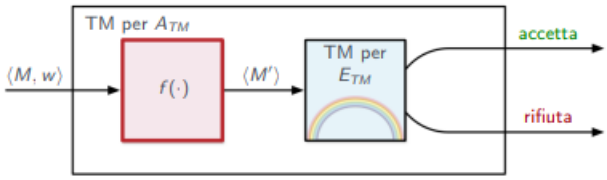
Il problema del vuoto (2)

$E_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$

- Possiamo dimostrare che $A_{TM} \leq_m E_{TM}$?
- Qual è l'input della funzione di riduzione?
- Qual è l'output?
- Quali proprietà devono rispettare?

5-7 18 of 22

$A_{TM} \leq_m E_{TM}$?



M accetta w se e solo se $L(M') = \emptyset$

5 Teoria della complessità

In questa sezione vediamo come classificare la qualità di un algoritmo, e vedremo determinati insiemi di problemi.

5.1 Complessità di tempo

Quando si analizza un algoritmo, è fondamentale conoscere il suo tempo di esecuzione. Esso ci permette di dire se tale algoritmo è buono o meno, e per tale valutazione si cerca sempre di guardare il tempo al caso peggiore, quindi il tetto di massima in cui l'algoritmo gira.

In particolare, si utilizza la notazione O-grande:

Definizione Date f,g due funzioni, se $\exists$ due interi positivi c,n0 : $\forall$ n $\geq$ n0: f(n) $\leq$ c g(n) allora f(n)=O(g(n)).
Si dice che g(n) è limite superiore asintotico per f(n).

Quindi, diamo sotto forma di teoremi/definizioni, delle valutazioni sulle varie TM viste sinora:

Teorema: Sia t(n) una funzione : t(n) $\leq$ n. Ogni TM multinastro di tempo t(n) ammette una TM equivalente a nastro singolo di tempo $O(t^2(n))$.

Definizione: Sia N una TM non deterministica che è anche un decisore. Il tempo di esecuzione di N è la funzione f : N $\rightarrow$ N tale che f(n) è il massimo numero di passi che N usa per ognuno dei rami di computazione, su input di lunghezza n.

Teorema: Sia t(n) una funzione : t(n) $\leq$ n. Ogni TM non deterministica di tempo t(n) ammette una TM equivalente a nastro singolo di tempo $2^{O}(t(n))$.

Inoltre, la tesi di Church-Turing implica che tutti i modelli di calcolo "ragionevoli" sono equivalenti tra di loro.

5.2 Tempo Polinomiale

Ora possiamo parlare della classe P.
Definizione P è la classe di linguaggi che sono decidibili in tempo polinomiale da una TM a singolo nastro.
Per polinomiale s'intende un tempo $O(n^k)$, con k$\geq$1. Questa classe corrisponde ai problemi che sono realisticamente risolvibili da un computer.
Inoltre, possiamo introdurre questo teorema:
Teorema Ogni linguaggio context-free è un elemento di P.
Ciò si dimostra partendo da un linguaggio CF, essendo decidibile, arrivando ad una soluzione con complessità $O(n^3)$.

5.3 La classe NP

Sappiamo che un problema è classificabile nella classe P se è decidibile in tempo polinomiale, rendendolo "facile". Ma se un problema è "difficile", ovvero risolverlo richiede un tempo più che polinomiale?
Perchè certi problemi richiedono tempi estremamente lunghi, come il gioco della Torre di Hanoi (circa $O(2^n)$), o non sono affatto risolvibili tramite un algoritmo, come l'Halting Problem di Turing. Tali linguaggi sono racchiusi nella classe NP.
La classe NP contiene linguaggi che non possono essere decisi in tempo polinomiale da una TM deterministica, ma tali linguaggi ammettono un verificatore che impiega tempo polinomiale. In pratica, in NP un algoritmo può verificare

la correttezza di un risultato in un tempo accettabile, ma non può decidere un linguaggio (e se lo fa, non impiega un tempo polinomiale). Definiamo il "verificatore":
Definizione Un verificatore per un linguaggio A è un algoritmo V : A={w : V accetta (w,c) per qualche stringa c}.
L'informazione necessaria al verificatore è il certificato c, con il quale stabilisce se w appartiene al linguaggio. Seguono due esempi di problemi nelle relative classi:

Raggiungibilità in un grafo

$PATH = \{\langle G, s, t\rangle \mid G$ grafo che contiene un cammino da s a $t\}$

Numeri relativamente primi

$RELPRIME = \{\langle x, y\rangle \mid 1$ è il massimo comun divisore di x e $y\}$

Problema del circuito Hamiltoniano

$HAMILTON = \{\langle G\rangle \mid G$ è un grafo con un circuito Hamiltoniano$\}$

Numeri composti

$COMPOSITES = \{\langle x\rangle \mid x = pq$, per gli interi $p, q > 1\}$

5.4 Problemi NP completi

Tra la classe NP, che abbiamo appena visto, possiamo specificarne due tipi. Prima però è necessario introdurre il teorema di Cook e Levin:
Teorema L'esistenza di una TM polinomiale per risolvere CircuitSAT implica che P=NP.

CircuitSAT è un problema in NP, e può essere usato per risolvere tutti i problemi NP in tempo polinomiale.
Possiamo affermare ciò perchè CircuitSAT è NP-completo, ovvero è un problema che appartiene alla classe NP ed è NP-Hard. Un problema è detto NP-Hard se l'esistenza di un algoritmo che impiega tempo polinomiale implica l'esistenza di un algoritmo che risolve ogni problema in NP in tempo polinomiale.
Trattare problemi NP-Completi permette di risolverlo identificando un caso particolare e cercando una soluzione approssimata, poichè risulta molto complesso trovare un algoritmo efficiente per il caso generale che potrebbe non esistere.

Concludiamo con alcune immagini:

Tesi di Church computazionale

Tutti i formalismi di calcolo ragionevoli sono equivalenti a meno di fattori polinomiali nei tempi di calcolo.

Esempi:

- Macchine di Turing Deterministiche
- Linguaggi di programmazione concreti: Java, C++, Python, ...

Eccezioni:

- Computer quantistici
- DNA Computing, Bio Computing

Verificatori

Definition

Un verificatore per un linguaggio A è un algoritmo V tale che

$$A = \{w \mid V \text{ accetta } \langle w, c \rangle \text{ per qualche stringa } c\}$$

- il verificatore usa ulteriori informazioni per stabilire se w appartiene al linguaggio
- questa informazione è il certificato c
- un verificatore polinomiale opera in tempo $O(|w|)^k$ per qualche costante k

Due problemi in P . . .

Raggiungibilità in un grafo

$PATH = \{\langle G, s, t\rangle \mid G \text{ grafo che contiene un cammino da } s \text{ a } t\}$

Numeri relativamente primi

$RELPRIME = \{\langle x, y\rangle \mid 1 \text{ è il massimo comun divisore di } x \text{ e } y\}$

. . . e due problemi in NP

Problema del circuito Hamiltoniano

$HAMILTON = \{\langle G\rangle \mid G \text{ è un grafo con un circuito Hamiltoniano}\}$

Numeri composti

$COMPOSITES = \{\langle x\rangle \mid x = pq, \text{ per gli interi } p, q > 1\}$

Un verificatore per *COMPOSITES*

La seguente TM è un verificatore per *COMPOSITES*. Il certificato è uno dei due divisori di x

$V =$ "su input $\langle x, p \rangle$, con x, p numeri interi:

1. se $p \leq 1$ o $p = x$, rifiuta
2. se p è un divisore di x, accetta, altrimenti rifiuta."

Problemi P ed NP

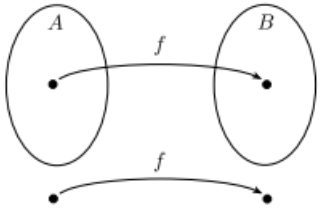
- **P** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing deterministica in tempo $O(|x|^k)$
- **NP** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere verificata da un verificatore in tempo $O(|x|^k)$.
- **Equivalente:** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing nondeterministica in tempo $O(|x|^k)$.
- **coNP** è la classe dei linguaggi tali che il loro complementare è in *NP*

Riducibilità in tempo polinomiale

Definition

Un linguaggio A è riducibile in tempo polinomiale al linguaggio B ($A \leq_P B$), se esiste una funzione calcolabile in tempo polinomiale $f : \Sigma^* \mapsto \Sigma^*$ tale che

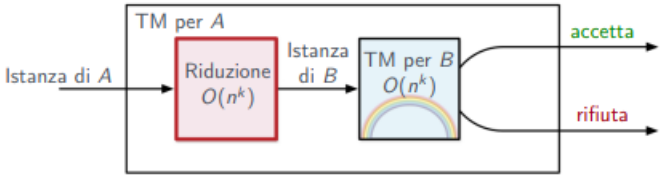
per ogni $w \in \Sigma^*$: $w \in A$ se e solo se $f(w) \in B$



f è la riduzione polinomiale da A a B

Riduzioni polinomiali

Se $A \leq_P B$, e $B \in P$, allora $A \in P$:



Dimostrare che un problema è NP-completo

Ogni dimostrazione di NP-completezza di compone di due parti:

1. dimostrare che il problema appartiene alla classe NP;
2. dimostrare che il problema è NP-hard.

- Dimostrare che un problema è in NP vuol dire dimostrare l'esistenza di un verificatore polinomiale.
- Le tecniche che si usano per dimostrare che un problema è NP-hard sono fondamentalmente diverse.

- Per dimostrare che un certo problema è NP-hard si procede tipicamente con una dimostrazione per riduzione polinomiale

Per dimostrare che B è NP-hard dobbiamo ridurre un problema NP-hard a B.

- Abbiamo bisogno di un problema NP-hard da cui partire: CircuitSAT

Schema di riduzione polinomiale

Per dimostrare che un problema B è NP-hard:

1. Scegli un problema A che sai essere NP-hard.
2. Descrivi una riduzione polinomiale da A a B:
   - data un'istanza di A, trasformala in un'istanza di B,
   - con una funzione che opera in tempo polinomiale.
3. Dimostra che la riduzione è corretta:
   - Dimostra che la funzione trasforma istanze "buone" di A in istanze "buone" di B.
   - Dimostra che la funzione trasforma istanze "cattive" di A in istanze "cattive" di B.
     Equivalente: se la tua funzione produce un'istanza "buona" di B, allora era partita da un'istanza "buona" di A.
4. Mostra che la funzione impiega tempo polinomiale.

Il problema della soddisfacibilità booleana

- una formula Booleana come

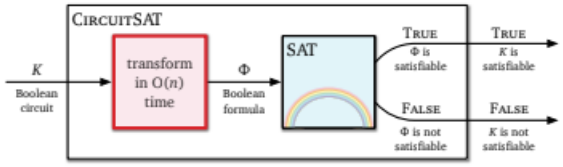
$$(a \vee b \vee c \vee \overline{d}) \leftrightarrow ((b \wedge \overline{c}) \vee \overline{(\overline{a} \rightarrow d)} \vee (c \neq a \wedge b)),$$

- è soddisfacibile se è possibile assegnare dei valori booleani (Vero/Falso) alle variabili $a, b, c, \ldots$, in modo che il valore di verità della formula sia Vero

SAT $= \{\langle \varphi \rangle \mid \varphi$ è una formula booleana soddisfacibile$\}$

SAT è NP-completo

- **SAT** è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- **SAT** è NP-hard:
  dimostrazione per riduzione di **CircuitSAT** a **SAT**



Correttezza della riduzione

Ora dobbiamo mostrare che il circuito originale K è soddisfacibile se e solo se la formula risultante Φ è soddisfacibile.
Dimostriamo questa affermazione in due passaggi:

- $\Rightarrow$ Dato un insieme di input che rende vero il circuito K, possiamo ottenere i valori di verità per le variabili nella formula Φ calcolando l'output di ogni porta logica di K.
- $\Leftarrow$ Dati i valori di verità delle variabili nella formula Φ, possiamo ottenere gli input del circuito semplicemente ignorando le variabili delle porte logiche interne y_i e la variabile di uscita z.

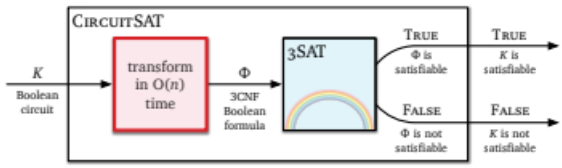
L'intera trasformazione da circuito a formula può essere eseguita in tempo lineare. Inoltre, la dimensione della formula risultante cresce di un fattore costante rispetto a qualsiasi ragionevole rappresentazione del circuito.

Una versione ristretta di soddisfacibilità

- Intendiamo dimostrare l'NP-completezza di un'ampia gamma di problemi
- Dovremmo procedere per riduzione polinomiale da **SAT** al problema in esame
- Esiste però un importante problema "intermedio", detto **3SAT**, molto più facile da ridurre ai problemi tipici rispetto a **SAT**:
  - anche **3SAT** è un problema di soddisfacibilità di formule booleane
  - **3SAT** però richiede che le formule siano di una forma ben precisa, formate cioè da congiunzione logica di clausole ognuna delle quali è disgiunzione logica di tre variabili (anche negate)

3SAT $= \{\langle\varphi\rangle \mid \varphi$ è una formula booleana in 3-CNF soddisfacibile$\}$

- **3SAT** è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- **3SAT** è NP-hard:
  dimostrazione per riduzione di **CircuitSAT** a **3SAT**



30 of 52

Riduzione di CircuitSAT a 3SAT

1. Fai in modo che ogni porta logica abbia al massimo due input
2. Trasforma il circuito in una formula booleana come fatto per SAT
3. Trasforma la formula in CNF usando le regole seguenti:

$$a = b \wedge c \mapsto (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$$
$$a = b \vee c \mapsto (\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c})$$
$$a = \overline{b} \mapsto (a \vee b) \wedge (\overline{a} \vee \overline{b})$$

4. Trasforma la formula in 3CNF aggiungendo variabili alle clausole con meno di tre letterali:

$$a \vee b \mapsto (a \vee b \vee x) \wedge (a \vee b \vee \overline{x})$$
$$a \mapsto (a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y})$$

6 Esercizi svolti

Di seguito sono elencati alcuni esercizi (risolti) tipicamente presenti all'esame.

Automi e Linguaggi Formali – 17/6/2021
Prima parte – Linguaggi Regolari e Context Free

1. *Dimostra che se L ed M sono linguaggi regolari sull'alfabeto $\{0,1\}$, allora anche il seguente linguaggio è regolare:*

$$L \sqcap M = \{x \sqcap y \mid x \in L, y \in M \text{ e } |x| = |y|\},$$

*dove $x \sqcap y$ rappresenta l'*and bit a bit *di x e y. Per esempio, $0011 \sqcap 0101 = 0001$.*

Poiché L e M sono regolari, sappiamo che esiste un DFA $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ che riconosce L e un DFA $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$ che riconosce M.

Costruiamo un NFA A che riconosce il linguaggio $L \sqcap M$:

- L'insieme degli stati è $Q = Q_L \times Q_M$, che contiene tutte le coppie composte da uno stato di A_L e uno stato di A_M.
- L'alfabeto è lo stesso di A_L e di A_M, $\Sigma = \{0,1\}$.
- La funzione di transizione δ è definita come segue:

  $$\delta((r_L, r_M), 0) = \{(\delta_L(r_L, 0), \delta_M(r_M, 0)), (\delta_L(r_L, 1), \delta_M(r_M, 0)), (\delta_L(r_L, 0), \delta_M(r_M, 1))\}$$
  $$\delta((r_L, r_M), 1) = \{(\delta_L(r_L, 1), \delta_M(r_M, 1))\}$$

  La funzione di transizione implementa le regole dell'and tra due bit: l'and di due 1 è 1, mentre è 0 se entrambi i bit sono 0 o se un bit è 0 e l'altro è 1.
- Lo stato iniziale è (q_L, q_M).
- Gli stati finali sono $F = F_L \times F_M$, ossia tutte le coppie di stati finali dei due automi.

2. *Considera il linguaggio*

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ contiene lo stesso numero di } 00 \text{ e di } 11\}.$$

Dimostra che L_2 non è regolare.

Usiamo il Pumping Lemma per dimostrare che il linguaggio non è regolare.

Supponiamo per assurdo che L_2 sia regolare:

- sia k la lunghezza data dal Pumping Lemma;
- consideriamo la parola $w = 0^k 1^k$, che appartiene ad L_2 ed è di lunghezza maggiore di k;
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$;
- poiché $|xy| \leq k$, allora x e y sono entrambe contenute nella sequenza di 0. Inoltre, siccome $y \neq \emptyset$, abbiamo che $x = 0^q$ e $y = 0^p$ per qualche $q \geq 0$ e $p > 0$. z contiene la parte rimanente della stringa: $z = 0^{k-q-p}1^k$. Consideriamo l'esponente $i = 0$: la parola xy^0z ha la forma

  $$xy^0z = xz = 0^q 0^{k-q-p} 1^k = 0^{k-p} 1^k$$

  e contiene un numero di occorrenze di 00 minore delle occorrenze di 11. Di conseguenza, la parola non appartiene al linguaggio L_2, in contraddizione con l'enunciato del Pumping Lemma.

3. *Dimostra che se L è un linguaggio context-free, allora anche L^R è un linguaggio context-free.*

Se L è un linguaggio context free allora esiste una grammatica G che lo genera. Possiamo assumere che G sia in forma normale di Chomksy. Di conseguenza le regole di G sono solamente di due tipi: $A \to BC$, con A, B, C simboli non terminali, oppure $A \to b$ con b simbolo nonterminale.

Costruiamo la grammatica G^R che genera L^R in questo modo:

- ogni regola $A \to BC$ viene sostituita dalla regola $A \to CB$;
- le regole $A \to b$ rimangono invariate.

Tutorato 01

Giulio Umbrella

14 March 2022

Nota questa versione delle soluzioni e' una bozza e puo' essere soggetta a modifiche

1 DFA

1.1 Ex 1

$w \in \sum ||w| = 2$

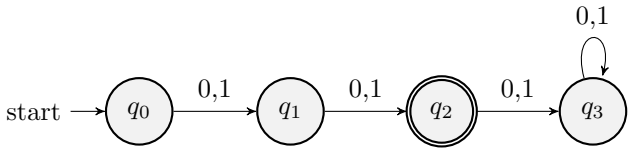


Figure 1: Esercizio 01

1.2 Ex 2

$w \in \sum ||w| <= 2$

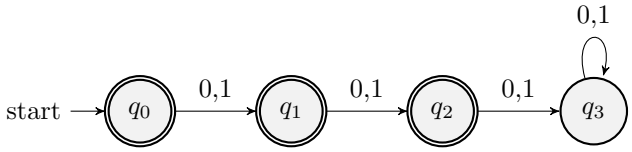


Figure 2: Esercizio 02

1.3 Ex 3

$w \in \sum ||w| mod 2 = 0$

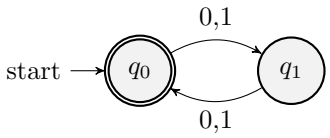


Figure 3: Esercizio 03

1.4 Ex 4

$w \in \sum | ogni 0 e' seguito da 11$

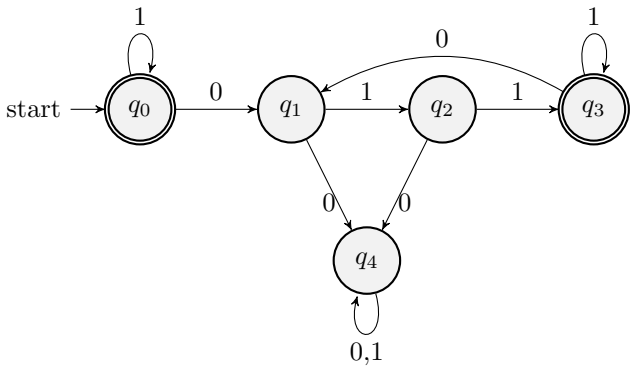


Figure 4: Esercizio 04

1.5 Ex 5

$w \in \sum | contiene 000 come sottostringa$

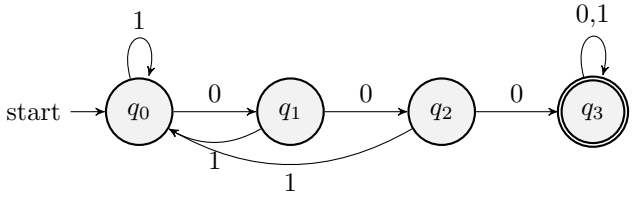


Figure 5: Esercizio 05

1.6 Ex 6

$w \in \sum | |w| mod 3 = 0$

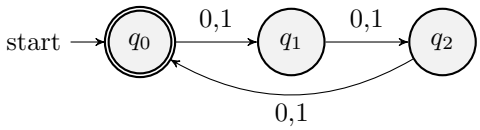


Figure 6: Esercizio 06

1.7 Ex 7

$w \in \sum ||w| mod 3 = 1$

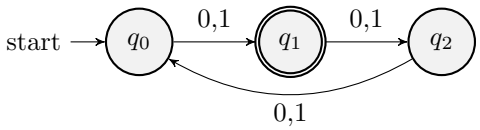


Figure 7: Esercizio 07

2 NFA ϵ -NFA

2.1 Ex 1

$w \in a, b, c|$ non compaiono tutti i simboli

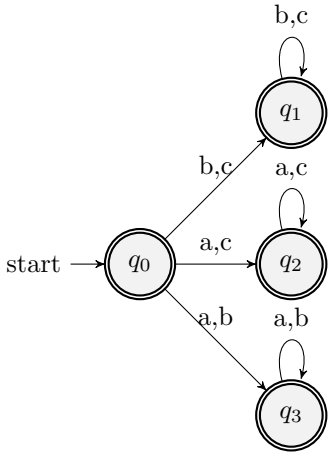


Figure 8: Esercizio 02

2.2 Ex 2

$w \in 0,1_*|$ contiene tre zero consecutivi

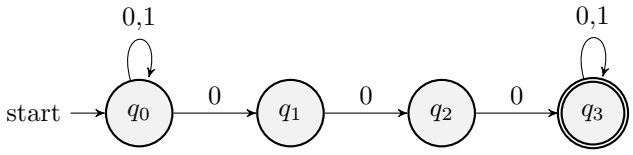


Figure 9: Esercizio 02

2.3 Ex 3

$w \in 0,1_*|$ contiene al suo interno la stringa 11 oppure 101

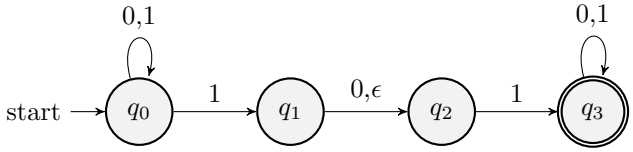


Figure 10: Esercizio 03

3 Conversione NFA $\rightarrow$ DFA

3.1 Ex 1

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow *\{q0, q1, q3\}$	{q1,q3}	{q2,q4}
{q1,q3}	{q1,q3}	{q2,q4}
$*\{q2, q4\}$	{q2,q5}	$\emptyset$
$*\{q2, q5\}$	{q2}	{q4}
$*\{q2\}$	{q2}	$\emptyset$
$*\{q4\}$	{q5}	$\emptyset$
{q5}	$\emptyset$	{q4}

Tutorato 02 Soluzioni Esercizi

Giulio Umbrella

03 Aprile 2022

1 Espressioni regolari

1.1 Ex1

Scrivere l'espressione regolare per i seguenti linguaggi.

1. 1(0+1)*1
2. ((0+1)(0+1))*
3. ((0+1)(0+1))*(0+1)
4. (0+1)*00
5. (0+1)(0+1)(0+1)0(0+1)
6. (1 + 01 + 0011)*(0+e)
7. (0+1)*00 + 0
8. (0+1)(0+1)
9. 01 + 10 + 11

1.2 Ex2

Dire se le seguenti affermazioni sono vere o false.

1. Falso
2. Vero
3. Devo controllare l'ordine di applicazione delle parentesi, aggiornero' il prima possibile.

2 Conversione da RE a FA

Convertire le seguenti espressioni regolari in FA
Per la soluzione applicare quanto riportato nelle slide

3 Conversion da FA a RE

3.1 Ex1

Fornire un FA per i seguenti linguaggi e convertire l'automa in una espressione regolare.

3.1.1 Ex1

Stringhe binarie di lunghezza pari.

Per prima cosa, costruiamo l'automa che riconosce le stringhe binarie di lunghezza pari, Figura 1

Prima di procedere, dobbiamo verificare se dobbiamo modifiare l'automa. Le etichette che contengono la virgola $0, 1$ vengono modificate in somme $0 + 1$ come vediamo in 2. L'automa ha un solo stato accettante, quindi possiamo applicare l'algoritmo.

Abbiamo un solo stato da rimuovere q_1, quindi procediamo come indicato dall'algoritmo. Il percorso da eliminare e' $q_0 \to q_0$, quindi facciamo la concatenazione delle espressioni regolari lungo il percorso - senza includere self loop perche' asssenti in q_1. Notare che il punto di partenza e di arrivo conicidono ma la tecnica da applicare e' la stessa. L'espressione regolare ottenuta viene aggiunta come self loop su q_1, Figura 3

In Figura 3 vediamo che lo stato iniziale e finale coincidono, quindi possiamo ricavare l'espressione regolare come $((0 + 1)(0 + 1))^\star$.

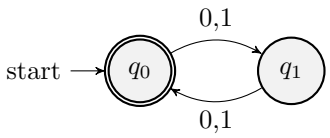


Figure 1: Automa di partenza

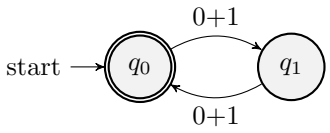


Figure 2: Sostituzione con espressoni regolari

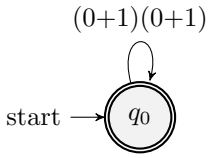


Figure 3: Rimozione stato q1

3.1.2 Ex2

Strighe binarie in cui il simbolo 1 e' ripetuto al piu' una volta sola

Anche in questo esercizio, il primo passaggio e' rappresentare l'automa che riconosce il linguaggio (Figura 4).

Rispetto all'esercizio precedente, l'automa ha bisogno di una modifica in piu'. Oltre a trasformare le etichette in espressioni regolari, dobbiamo ricavare un automa con un solo stato accettante. Per farlo e' sufficiente:

1. Aggiungere un nuovo stato accettante
2. Aggiungere una epsilon transizione da ciascuno stato accettante al nuovo stato accettante
3. Gli stati accettanti nell'automa di partenza diventano stati ordinari.

Il risultato e' in Figura 5

Ora possiamo procedere ed eliminiamo lo stato q_1. Osservando le frecce in ingresso ed in uscita, vediamo che lo stato ha un precedessore e due successori, quindi dobbiamo modificare due percorsi $q_0 \rightarrow q_3$ e $q_0 \rightarrow q_2$.

Per il percorso $q_0 \rightarrow q_2$ concateniamo le espressioni regolari lungo il percorso e inseriamo in mezzo il self-loop su q_1 e otteniamo l'espressione regolare $10^\star 1$. Notiamo che non ci sono percorsi diretti fra $_0$ e q_2, quindi non dobbiamo aggiunggere altri percorsi. Il risultato e' in 6.

Per il percorso $q_0 \rightarrow q_3$ dobbiamo fare piu' attenzione. Abbiamo un percorso rappresentato da $10^\star\epsilon$ e un percorso diretto rappresentato da ϵ, quindi sappiamo che dobbiamo unire le due espressioni regolare tramite il $+$. Qui e' bene fermarsi un attimo e ragionare su cosa stiamo facendo; mentre nella concatenazione il simbolo ϵ puo' essere rimosso, nell'unione dobbiamo convervarlo, come si vede in Figura 6.

Il motivo e' che nella concatenazione stiamo aggiungendo il simbolo vuoto all'espressione regolare $10^\star\epsilon$ e quindi lasciamo il risultato immutanto. Nell'unione

$10^\star\epsilon+\epsilon$ invece stiamo aggiungendo il simbolo vuoto all'espressione regolare, dandoli la possibilita' di accettare la parola vuota. In altri termini, scrivere $10^\star$ e' errato perche' stiamo lasciando fuori una parola, quella vuota.

La rimozione di q_1 da come risultato l'automa di Figura 6. Per prima cosa osserviamo che lo stato q_2 non ha alcuna transione verso q_3. Infatti, il percorso da q_0 a q_2 contiene tutte le stringhe non accettate dal linguaggio, ossia quelle in cui il simbolo 1 e' ripetuto piu' di una volta. Lo stato q_2 rappresenta una sorta di "cestino" o vicolo cieco; una volta entrati non e' piu' possibili arrivare in uno stato accettante. Per proseguire possiamo semplicemente rimuoverlo, ottenendo l'automa in Figura 7.

Abbiamo due stati, quindi terminiamo come suggerito dall'algoritmo e otteniamo l'espressione regolare $0^\star10^\star + \epsilon$.

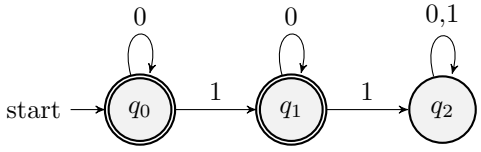


Figure 4: Automa di partenza Step 1

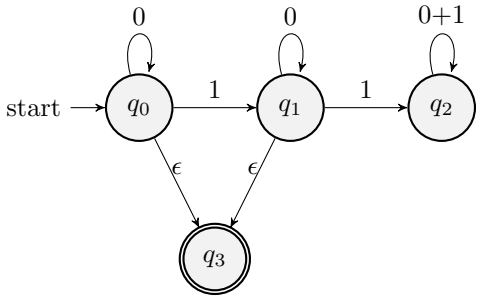


Figure 5: Aggiunta nuovo stato finale q3

3.1.3 Ex3

Stringhe binarie che non comprendono la stringa 101.

Questo linguaggio presenta una complicazione, infatti richiede di ottenere tutte le stringhe che non rispettano una condizione. Anziche' ragionare su questo automa, possiamo ragionare in due passaggi:

1. Costruiamo il DFA che accetta il linguaggio complementare.

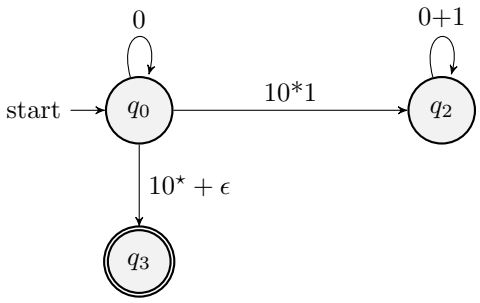


Figure 6: Rimozione stato q1

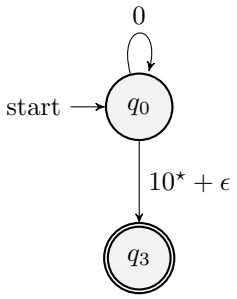


Figure 7: Rimozione stato q2

2. Invertiamo il DFA per ottenere il linguaggio di interesse.

L'operazione complemento infatti e' chiusa per la classe dei linguaggi regolari; se applichiamo il complemento ad un linguaggio regolare, otteniamo un linguaggio regolare.

Per poter ottenere l'inverso, scegliamo di costruire un DFA che accetta il linguaggio complementare (Figura 8). Infatti, sappiamo che DFA e NFA sono equivalenti, ma come avete visto in classe, possiamo ottenere il linguaggio complementare da un DFA in modo piu' agevole, invertendo gli stati (Figura 9).

Una volta ottenuto l'automa, come primo step lo modifichiamo per ottenere un automa su cui applicare l'algoritmo di conversione, il risultato e' in Figura 10. Anche per questo esercizio, abbiamo uno stato che porta a parole non accettate dal linguaggio, lo stato q_3. Possiamo quindi rimuovere questo stato e procedere come l'algoritmo.

3.2 Ex2

Convertire in RE il seguente automa (Figura) 11).

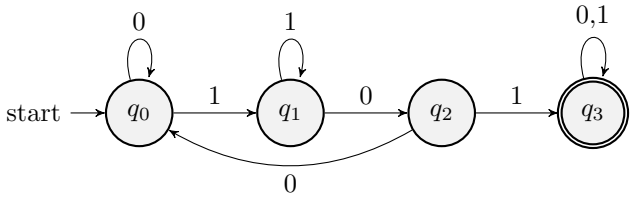


Figure 8: DFA linguaggio complementare

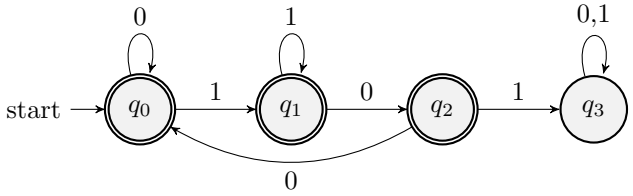


Figure 9: DFA inverso per lingugaggio di partenza

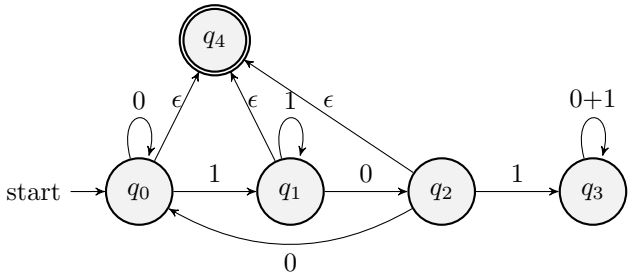


Figure 10: NFA

Per procedere rimuoviamo lo stato q_1 (Figura 12 e ricaviamo l'espressione regolare $1 + 01^\star 0(0 + 1)^\star$

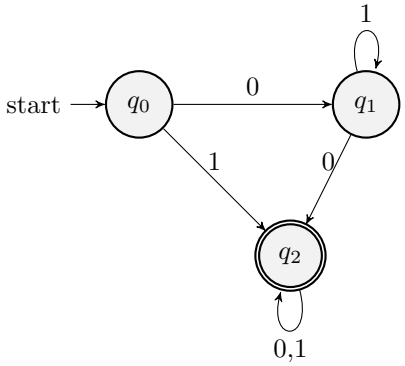


Figure 11: Automa di partenza

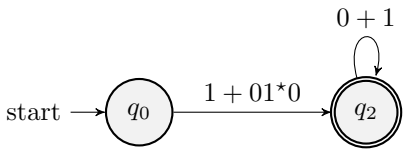


Figure 12: Rimozione stato q_1

Tutorato 03

Giulio Umbrella 03 Aprile 2022

Ex 1 Linguaggi regolari

Ex 1.1

Per una stringa w = $w_1 w_2 ... w_n$, l'inversa e' la stringa $w^R = w_n, \cdots, w_2, w_1$. Per ogni linguaggio regolare A, sia $A^R = \{w^R | w \in A\}$. Mostrare che se A e' regolare, allora lo e' anche A^R.

Questo esercizio e' diverso da quelli visti in precedenza; dato un linguaggio regolare, ci viene chiesto di dimostare che con una modifica possiamo ottenere un altro linguaggio regolare. Dobbiamo percio' ragionare in modo piu' generico, senza produrre un automa specifico. Una soluzione informale e' la seguente

1. Dato il linguaggio A, produco un NFA_1 che lo riconosce A.
2. Per produrre l'automa che riconoscere il linguaggio A^R, posso modificare NFA_1 in NFA_2
    1. Inverto la direzione delle transizioni
    2. Lo stato iniziale di NFA^1 diventa accettante
    3. Lo stato accettante di NFA^2 diventa iniziale
3. Se NFA^1 ha piu' di uno stato accettante, aggiungo uno nuovo stato iniziale e una epsilon transizione pverso tutti gli ex-stati accettanti.

Per questo esercizio e' necessario partire da un NFA in modo da avere un solo percorso da stato iniziale ad accettante e poter seguire l'automa in senso inverso.

Ex 1.2

Sia A/b = $\{w | wb \in A\}$. Mostrare che se A e' un linguaggio regolare e $b \in \sigma$, allora A/b e' regolare.

Come in Ex1.1 partiamo da un automa che riconosce il linguaggio regolare A e produciamo l'automa che riconosce A/b. Modifichiamo l'automa spostando lo stato accettante allo stato precendente a quello in cui viene riconosciuta la parola che termina per b (ed eliminiamo la transione associata).

2 Pumping Lemma

Dimostrare che i seguenti linguaggi non sono regolari

Ex 2.1

$\{0^n 1^m 0^n | m, n \geq 0\}$

Per dimostrare che il linguaggio non e' regolare procediamo per contraddizione, ossia ipotiziamo che il linguaggio sia regolare e dimostriamo che questo porta ad una contraddizione, l'obbiettivo e' trovare una parola che rispetti le condizioni del pumping lemma che una volta pompata produca una parola non appartenente al linguaggio.

Il pumping lemma infatti deve valere per tutte le parole che ne rispettano i requisiti; basta trovare una singola parola che viola la condizione per falsificarlo

Passo 1 Scelta dimostrazione: la tecnica che usiamo e' la contraddizione - assumiamo che il linguaggio sia regolare e che rispetti il pumping lemma. Esiste quindi una costante k che rappresenta la lunghezza minima della parola.

NB: stiamo ipotizzando solo che k esista, non stiamo effettivamente scegliendo un valore.

Passo 2 Scelta parola: sappiamo che il pumping lemma deve valere per tutte le parole di lunghezza almeno k, quindi scegliamo una parola che rispecchi tale condizione. Una possibile scelta e' $w = 0^k 1^k 0k$, dato che soddisfa automaticamente la condizione di lunghezza minima.

Passo 3 Suddivisione parola: ora dobbiamo spezzare la parole in tre componenti x, y e z. Il teorema ci dice che $|xy| \leq 0$ e $|y| > 0$. La scelta della parola $w = 0^k 1^k 0k$ ci garantisce in modo automatico la struttura di xy e y, dato che i suoi primi k caratteri sono tutti zero.

Possiamo quindi scrivere che

- $y = 0^p$, con $p > 0$
- $xy = 0^{k-p}$
- z e' uguale al resto della parola

Passo 4 Scelta i: i passi da 1 a 3 sono funzionali al passo quattro; ora dobbiamo usare quanto fatto prima per dimostrare che la parola pompata non appartine al linguaggio. Quindi con $w' = xy^iz$ se scegliamo i = 0otteniamo $xy^0z = 0^{k-p}1^k0k$.

La parola w' non appartiene al linguaggio, e quindi le condizioni del pumping lemma sono violate. L'ipotesi iniziale mi porta ad una contraddizione e quindi il linguaggio non puo' essere regolare.

Ex 2.2

$\{w \in \{0,1\} \mid we'palindroma\}$

Per dimostrare che il linguaggio non e' regolare, procediamo come nel precedente esercizio.

1. Assumiamo che il linguaggio sia regolare e che esista un costante k
2. Scegliamo una parola w che appartiene al linguaggio con $|w| > k$, ad esempio $w = 0^k0^k$
3. Suddividiamo la parola in xyz

- $y = 0^p, p > 0$
- $x = 0^{k-p}$
- z tutto il resto

4. Dimostriamo che esiste un i per cui la parola xy^iz non appartiene al linguaggio. Ad esempio con i = 0 otteniamo che $xy^0z = 0^{k-p}0^k$ ossia una parola non palindroma.

NB: sia per Ex 2.1 che 2.2 abbiamo scelto i=0 per trovare un controesempio, ma non e' l'unico possibile. Qual'e' il solo valore che non puo' essere scelto?

EX 2.3

Per questo esercizio possiamo utilizzare il risultato dell'Ex 2.2. Sappiamo infatti che il complementare di linguaggio regolare e' regolare, e ragionare per assurdo.

1. Assumiamo che il linguaggio delle parole non palindrome sia regolare.
2. Dal punto 1, sappiamo che il linguaggio complementare composto dalle parole palindrome e' regolare
3. Il punto 2 ci porta ad una contraddizione, perche' abbiamo dimostrato che il linguaggio delle parole palindrome non puo' essere regolare.

Ex 2.4

Questo esercizio puo' essere risolto come l'esercizio 2.3 utilizzando la non regolarita' del linguaggio complementare. Dal momento che sia p che q sono strettamente maggiori di zero, il linguaggio e' formato da parole di lunghezza composta. Il linguaggio di parole di lunghezza pari a numeri primi non e' regolare (vedi Slide) e quindi il linguaggio non e' regolare.

Ex 2.5

5. $\{0^n1^m0^{m+n}n, m \in \mathbb{N}\}$

La soluzione di questo esercizio e' simile agli esercizi Ex 1 e Ex 2.

Ex 2.6

$\{0^n0^{2^n}n, m \in \mathbb{N}\}$

Questo esercizio verra' svolto nel prossimo tutorato [se non fosse possibile, aggiornero' le soluzioni].

Ex 3 Lunghezza minima pumping

Per ognuno dei seguenti linguaggi, dare la lunghezza minima del pumping e giustificare la risposta.

L'esercizio chiede di identificare una lunghezza minima in modo da poter applicare il pumping lemma. In altri termini, dobbiamo indentificare la lunghezza per cui non e' possibile in alcun modo applicare il pumping lemma.

NB: le soluzioni degli altri punti verrano caricate prossimamente.

Ex 3.1

$0001^\star$

Per questo linguaggio, possiamo vedere che e' impossibile applicare il pumping lemma per parole piu' corte di tre. Infatti pompando una parola come 000 otterremmo una parola con piu' di tre zeri, che non viene riconosciuta dal linguaggio. La lunghezza minima e' quattro, infatti data la parola 0001 possiamo impostare xy = 0001, y = 1 e z = ϵ e pompare la parola per ogni i.

Ex 3.3

1011

Il linguaggio e' composta da una sola parola. Qualsiasi suddivisone in xyz ci darebbe la possibilita' di pompare la parola aggiungendo o togliendo caratteri. La lunghezza minima e' quindi 5.

NB: se utilizzassimo i=0 otterremo parole appartenti al linguaggi perche' la parola non viene modificata, ma il nostro obbiettivo e' trovare una lunghezza che funzioni per ogni $i \geq 0$

Soluzioni Tutorato 4

Giulio Umbrella

Ex 1 Grammatiche libere da contesto

Definire delle gramatiche libere da contesto per i seguenti linguaggi

Ex 1.1

{ w | la lunghezza di w e' dispari con all'interno il simbolo zero}

Data la parola w, possiamo dividerla nella concatenazione di tre componentti, w = Sx 0 Dx.

Per la somma di numeri pari e dispari sappiamo che valgono i seguenti

- Pari + Pari = Pari
- Dispari + Dispari = Pari
- Pari + Dispari = Dispari

Possiamo verificare se w e' pari o dispari sulla base della lunghezza delle tre componenti.

|Sx|	|0|	|Dx|	Dispari?
Pari	Dispari	Pari	S
Pari	Dispari	Dispari	N
Dispari	Dispari	Pari	N
Dispari	Dispari	Dispari	S

Quindi vogliamo che a destra e a sinistra le stringhe siano entrambe di lunghezza pari o entrambe di lunghezza dispari.

Le produzioni sono le seguenti:

$S \rightarrow$ P0P | D0D| 0
$P \rightarrow 00P|01P|10P|11P|\epsilon$
$D \rightarrow 0P|1P$

Ex 1.2

$\{w \mid w = w^R$,w e' palindroma$\}$

Per ogni produzione, dobbiamo aggiungere lo stesso carattere a destra e a sinistra. Quindi definima una singola variabile che aggiunga lo stesso simbolo ogni volta che viene invocata. Aggiungiamo inoltre due produzioni in piu con terminali 0 e 1, in modo da produrre anche stringhe di lunghezza 1 (sono palindrome) e stringhe palindrome di lunghezza pari. Le regole per la grammatica sono le seguenti.

$R \to 0R0|1R1|1|0|\epsilon$

Ex 1.3

$\{w\#x \mid w^R$ e' una sottostringa di x $\}$

In maniera informale possiamo rappresentare il linguaggio nel seguente modo $\{w\#\{0+1\}^\star w^R\{0+1\}^\star\}$ (NB la precedente rappresentazione **non** vuole essere una espressione regolare, ma un modo schematico di capire la struttura delle stringe)

Possiamo individuare due componenti

1. L'insieme di tutte le stringhe formate da 0,1
2. Le stringhe palindrome - possiamo infatti pensare alle stringhe palindrome come alla contatenzione di una parola w con la sua opposta.

Fatto questo, scriviamo delle regole per i due casi

1. $X \to 0X|1X|\epsilon$
2. $T \to 0T0|1T1$

Il problema adesso e' come combinare le due regole? Per farlo, possiamo pensare che le variabili possono essere chiamate in qualsiasi ordine, non necessariamente da destra a sinistra. Possiamo modificare la prima regola nel seguente modo $T \to 0T0|1T1|\#X$. Tornando al nostro esempio informale, abbiamo una regola che ci permette di produrre stringhe del tipo $\{w\#Xw^R\}$. Adesso possiamo usare la regola su X per aggiungere qualsiasi stringa.

L'ultimo passaggio e' capire come aggiungere la parte finale della stringa. Per farlo aggiungiamo una nuova regola per inserire immediatamente X a destra di T, $S \to TX$

La grammatica che otteniamo e' la seguente:

$S \to TX$
$T \to 0T0|1T1|\#X$
$X \to 0X|1X|\epsilon$

Ex 1.4

$\{x\#y \mid x, y \in \{0,1\}^\star, x \neq y\}$

S $\rightarrow$ A#B | B#A
A $\rightarrow$ TAT|0
B $\rightarrow$ TBT|1
T $\rightarrow$ 0|1

Ex 2 Grammatiche ambigue

Considerate la seguente grammatica

S $\rightarrow$ A | If-then | If-then-else
If-then $\rightarrow$ if cond then S
If-then-else $\rightarrow$ if cond then S else
A $\rightarrow$ a:=1

Ex1 Dimostrare che la grammatica e' ambigua
Ex2 Modificare la grammatica per rimuovere l'ambiguita'

Ex 2.1 Dimostrare che la grammatica e' ambigua

Per dimostrare chw una grammatica e' sufficiente tovar due derivazioni a sinistra che producono la stessa parola. Un possibile esempio e' il seguente:

S $\rightarrow$ If-then $\rightarrow$ If cond then S $\rightarrow$ If cond then It-then-else $\rightarrow$
If cond then If cond then S else S

S $\rightarrow$ If-then-else $\rightarrow$ If cond then S else S $\rightarrow$ If cond then If-then else S $\rightarrow$
If cond then If cond then S else S

Per completare entrambe le derivazioni, sostituiamo S con A e poi inseriamo il terminale a:=1

Per illustrare il problema, possiamoo vedere che le due derivazioni possono essere interpreatate in due modi diversi:

```
if (conditionA) {
if (conditionB) statementA; else statementB;
}
```

oppure

```
if (conditionA) {
if (conditionB) statementA;
}
else statementB;
```

Ex 2.2 Rimuovere ambiguita' da grammatica

Per rimuovere l'ambiguita', dobbiamo modificarne le regole. Introduciamo quindi due diverse variabili che definiamo Open e Closed con le seguenti caratteristiche:

- **Open**: puo' produre un if senza corrispettivo else
- **Closed**: non ha nessun if oppure tutti gli if hanno un corrispettivo else

Quindi modifichiamo le regole come segue:

S $\rightarrow$ | Open | Closed
Open $\rightarrow$ if cond then S | if cond then Closed else Open
Closed $\rightarrow$ A | if cond then Closed else Closed
A $\rightarrow$ a:=1

Possiamo vedere che tra un if e un else, possiamo aggiungere un if solo se aggiungiamo un corrispettivo else.

Possiamo controllare la derivazione ottenuta in precedenza:

S $\rightarrow$ Open $\rightarrow$ If cond then S $\rightarrow$ if cond then Closed $\rightarrow$ If cond then If cond then Closed If cond Closed $\rightarrow$ If cond if Cond A if cond Closed $\rightarrow$ If cond if Cond A if cond A

Ex 3 Forma normali

Convertire in forma normale la seguente grammatica

A $\rightarrow$ BAB|B|ϵ
B $\rightarrow$ 00|ϵ

Passo 1 Aggiungere variabile iniziale La variabile iniziale A e' presente anche a destra, quindi dobbiamo aggiungere una nuova variable iniziale.

S $\rightarrow$ A
A $\rightarrow$ BAB|B|ϵ
B $\rightarrow$ 00|ϵ

Questo passo e' necessario solo se la variable iniziale compare a destra di qualche produzione.

Passo 2 Rimuovere regole ϵ

Rimuovo A $\rightarrow \epsilon$

Aggiungiamo la produzione ϵ alla variabile iniziale S. Possiamo farlo perche' la forma normale di Chomsky ammetta la ϵ per la variabile iniziale.

S $\rightarrow$ A|ϵ
A $\rightarrow$ BAB|B|
B $\rightarrow$ 00|ϵ

Rimuovo B $\rightarrow \epsilon$

Ora rimuoviamo la ϵ per la B. Dato che abbiamo la produzione BAB, dobbiamo considerare tre possibili casi, BA, AB, A.

S $\rightarrow$ A|ϵ
A $\rightarrow$ BAB|B|BA|AB|A
B $\rightarrow$ 00

Attenzione! Abbiamo eliminato la produzione A -> ϵ al passo precedente, quindi **non** dobbiamo aggiungerla.

Passo 3 Rimuovere regole unitarie

Abbiamo tre regole unitarie da eliminiare

Possiamo eliminiare A-> A perche' e' una regola ciclica (non produce nulla) e sostituiamo 00 al posto di B nella produzione di A. Fatto questo, rimuoviamo la regola unitaria da S.

S $\rightarrow$ BAB|00|BA|AB|ϵ
A $\rightarrow$ BAB|00|BA|AB
B $\rightarrow$ 00

Passo 4 Conversione regole finali

Abbiamo una produzione con tre variabili a destra della freccia, introduciamo una nuova produzione per avere solo due variabli a destra.

S $\rightarrow$ CB|00|BA|AB|ϵ
A $\rightarrow$ CB|00|BA|AB
B $\rightarrow$ 00
C $\rightarrow$ BA

Ora modifichiamo le regole che producono terminali introducendo una regola per produrre il terminale 0 per riportarci alla forma X $\rightarrow$ x.

S $\rightarrow$ CB|DD|BA|AB|ϵ
A $\rightarrow$ CB|DD|BA|AB
B $\rightarrow$ DD
C $\rightarrow$ BA D $\rightarrow$ 0

Ex 4 Esercizi aggiuntivi

Convertire in forma normale le seguenti grammatiche

Ex 4.1

S $\rightarrow$ aXbX
X $\rightarrow$ aY|bYϵ
Y $\rightarrow$ X|c

Ex 4.2

$S \rightarrow AbA$
$A \rightarrow Aa|\epsilon$

Soluzioni Tutorato 5

Giulio Umbrella

Ex 1

Dimostrare che il seguente linguaggio A = $\{0^n | n = 2^k, k \in \mathbb{N}\}$ non e' regolare.

Per dimostrare l'affermazione procediamo per assurdo usando il Pumping Lemma; cioe' dimostriamo che esiste una parola che una volta pompata non appartiene al linguaggio.

Il procedimento e' simile a quanto visto in precedenza

1. Assumiamo che il linguaggio sia regolare
2. Scegliamo una parola che rispetti la lunghezza minima
3. Suddividiamo la parola in xyz
4. Troviamo i tale che $xyiz$ non appartiene al linguaggio

Per questo esercizio, la suddivisione della parola e' immediata, dato un qualsiasi k ottengo $w = 0^k$ e quindi ottengo automaticamente che

- $y = 0^b, b > 0$
- $x = 0^a, a + b \leq k$
- z e' formata solo zeri

Applicando il pumping lemma con i = 0 otteniamo $xy^0z = xz = 0^{2^k - b}$

Il problema e' nella scelta del valore da dare ad i. Per alcuni tipi di esercizi, la scelta di i e' molto semplice, ad esempio per il linguaggi delle parole palindrome abbiamo che se $w = 0^k0^k$, basta impostare i = 0 per ridurre la lunghezza della prima meta' e ottenere una parola che non appartiene al linguaggi.

In questo esercizio invece dobbiamo prestare piu' attenzione. Esistono infatti dei valori di i che possono produrre una parola che appartiene al linguaggio. Ad esempio, $2^6 - 32 = 32 = 2^5$ e quindi la parola appartiene al linguaggio.

Quindi non basta ridurre il numero di simboli, dobbiamo dimostrare che la parola che si ottiene non appartiene al linguaggio per un generico K. Nella dimostrazione assumiamo che il linguaggio sia regolare e che esista la costante del pumping lemma k, senza fare ulteriori assunzioni. La dimostrazione e' quindi parametrizzata sul valore di k, e quindi dobbiamo trattarla come un parametro generico.

Una possibile dimostrazione e' la seguente:

- $w = 0^{2^k}$
- $x = \epsilon$
- $y = 0^p, 0 < p < k$
- $z = 0^{2^k - p}$

Possiamo considerare le potenze di due come $2^2, 2^3, \ldots, 2^k, 2^{k+1}, \ldots$, se riusciamo a pompare una parole per produrne una di lunghezza compresa fra 2^ke 2^{k+1}, abbiamo prodotto una parola che non appartiene al linguaggio.

Se prendiamo i = 2, $xy^2z = (0^p)^2 0^{2^k - p} = (0^{2p})0^{2^k - p} = 0^{2^k} 0^p$

Ora dobbiamo dimostrare che la parola non puo' appartenere al linguaggio. Se prendiamo la parola appartenente al linguaggio successiva 02^{k+1} la possiamo scomporre come $0^{2^{k+1}} = 0^{2^k 2} = 0^{2^k + 2^k} = 0^{2^k} 0^{2^k}$

Dato che p e' sicuramente minore di 2^k, la parola non appartiene al linguaggio

Ex 2

Dati due linguaggi A e B definiamo lo shuffle dei due linguaggi come $\{w | w = a_1 b_1, \ldots, a_k b_k, \text{ con }, a_1, \ldots, a_k \in A, b_1, \ldots, b_k \in B \text{ con } a_i, b_i \in \Sigma\}$.

Dimostrare che se A e B sono regolari, lo shuffle di A e B e' un linguaggio regolare.

Dimostrazione informale

Per dimostrare che un linguaggio e' regolare, basta produrre un automa a stati finiti. A differenza di altri esercizi in cui abbiamo una precisa definizione del linguaggio (ad esempio stringhe binarie di lunghezza pari), in questo contesto sappiamo solo che il linguaggi e' derivato da altri linguaggi.

Questo esercizio va percio' affrontato in modo generale; sappiamo infatti solo che A e B sono regolari, ma non abbiamo indicazioni di nessun tipo sulla loro struttura. Dobbiamo costruire una dimostrazione di carattere generale che funziona per ogni linguaggio regolare A e B.

Sappiamo che A e B sono due linguaggi regolari quindi sappiamo che esistono due automi a stati finiti che li rappresentano. Attenzione, sappiamo che i due automi esistono, ma non abbiamo informazioni su cosa contengono.

Dati i due automi D_A e D_B definiamo

- $D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$
- $D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$

Utilizzando questi automi, costruiamo un automa D_S che riconosce le parole composte dallo shuffle di due parole di A e B. Possiamo pensare che D_S ha a

disposizione D_A e D_B, quindi l'automa puo' richiamarne le funzioni di transizione. Sappiamo che nell'input i caratteri sono alternati fra la parola in A e quella in B.

L'ordine delle operazioni e' il seguente:

1. L'automa D_S parte dallo stato iniziale di A e B
2. L'automa D_S legge il primo input e richiama le funzioni di transizioni di D_A e aggiorna lo stato, infatti sappiamo che il primo simbolo appartiene al linguaggio A ; in questa fase D_B rimane allo stato iniziale
3. Il secondo input per costruzione appartiene al linguaggio B; l'automa D_S lascia D_A immutato e aggiorna D_B.
4. La procedura riparte aggiornando D_A

In questo modo, le due parole di A e B vengono processate in parallelo ma in modo asincrono.

Dimostrazione formale

Per la dimostrazione formale dobbiamo define i cinque elementi di un automa a stati finito. L'alfabeto Σ e' lo stesso dei due automi.

Funzione di transizione

La funzione di transizione e' definita nel seguente modo

$\delta((x, y, A), a) = (\delta_A(x, a), y, B)$

Input: la funzione prende input una tupla di tre valori e l'input corrente

- x stato corrente di $D_A, x \in Q_A$
- y stato corrente di $D_B, y \in Q_B$
- A flag per indicare che l'input a deve essere processato usando D_A

Output: la funzione produce in output una tupla di tre elementi - (x,a) nuovo stato di D_A a partire da x con input a - y stato corrente di D_B - B flag per indicare che l'input deve essere processato usando l'automa D_B

Definiamo poi in modo simile per i passaggi da B ad A con $\delta((x, y, B), b) = (x, \delta_B(y, b), A)$.

Possiamo notare i seguenti punti:

- La procedura aggiorna D_A ma lascia immutato $D_B(oviceversa)$.
- Il flag A viene cambiato in B per indicare che il prossimo input deve essere processato usando D_B.
- Il meccanismo della funzione di transizione di D_S e ' sempre lo stesso: la funzione prende in input uno stato e simbolo e produce in output uno stato.
- Gli stati di D_S sono identificati da tuple di 3 elementi

Insieme di stati

Per completare l'automa, dobbiamo definirne gli stati:

$Q_S = Q_A$ x Q_B x {A,B}

NB: X e' il prodotto cartesiano fra insiemi, ossia l'insieme prodotto da tutte le coppie di singoli elementi. Ad esempio {a,b} X {c,d} = {{a,c), (a,d), (b,c), (b,d)}

In questo modo stiamo creando tutte le possibili combinazioni di coppie di stati e flag. Non tutti gli stati saranno necessariamente percorsi, ma in questo modo stiamo creando tutte le possibili combinazioni di stati in cui possono essere A e B.

Stato iniziale

q = (q_A, q_B, A), ossia gli automi vengono processati a partire dai loro rispetti stati iniziali, e il primo automa ad essere processato e' A.

Stato accettante

F = F_A x F_B x {A}, l'automa accetta se entrambi gli stati sono accettanti; il flag A serve ad indicare che il prossimo input e' dal linguaggio A, e quindi l'input letto in precedenza era da B.

Ex 3

Sia L un linguaggio regolare su un alfabeto Σ con $\# \in \Sigma$ e sia dehash(w) la funzione che rimuove il simbolo hash dalla stringa. Ad esempio dehash(1#1) = 11, dehash(0#10#) = 010. Dimostrare che il linguaggio dehash(L) = $\{dehash(w) : w \in L\}$ e' regolare.

Ex 4

Sia L un linguaggio regolare su un alfabeto Σ. Dimostrare che il linguaggio suffixes(L) = $\{y|xy \in L$per qualche stringa x $\in \Sigma^*\}$ e' regolare.

Esercizi aggiuntivi

Ex 5.1

Dimostrare che il linguaggio $\{0^m 1^n|$n/m e' un numero intero$\}$ non e' regolare

Soluzione

Dimostriamo che il linguaggio non e' regolare per assurdo;

- k lunghezza del pumping lemma
- scegliamo la parola $w = 0^{k+1}1^{k+1}$, che soddisfa la lunghezza minima e appartiene al linguaggio.

Ora suddividiamo la parola w in xyz con $|xy| < k$ e $y \neq \epsilon$

Data la lunghezza della parola e i requisti del pumping lemma sappiamo che xy e formata interameente da 0, e quindi abbiamo

- $x = \epsilon$
- $y = 0^p, p > 0$
- $z = 0^{k+1-p}1^{k+1}$

Consideriamo l'esponente i = 2 e otteniamo la parola $xy^2z = 0^{2p}0^{k+1-p}1^{k+1} = 0^{k+1+p}1^{k+1}$.

Otteniamo un parola con n/m = (k+1)/(k+1+p), ossia un numero compreso fra 0 e 1. Il valore non e' intero e quindi la parola non appartiene al linguaggio. Il linguaggio quindi non e' regolare.

Ex 5.2

Siano L e M due linguaggi regolari su alfabeto {0,1}. Dimostrare che il linguaggio L&M = $\{x\&y | x \in L, y \in M, |x| = |y|\}$, dove x&y e' l'and logic bit a bit. Per esempio, 101&001 = 001.

Soluzione

L'obiettivo e' costruire un automa D_S che riconosca le stringhe composte dall'and logico delle stringhe riconosciute da D_A e D_B. Data una stringa in input, vogliamo che sia riconosciuta dal linguaggio se entrambi gli automi la riconoscono.

I linguaggi sono regolari, quindi abbiamo a disposizione due automi:

- $D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$
- $D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$

Per decidere se accettare o meno, l'automa D_S ha a disposizione gli automi dei due linguaggi; per ogni simbolo in output, D_S fornisce il simbolo sia a D_A che D_{rBA}. Se entrambi gli automi accettano la parola, l'automa D_S accetta a sua volta.

Il punto e' capire che tipo di input fornire ai due automi. Per capire meglio cosa fare, usiamo la tabella logica dell'operazione And:

A	B	A & B
1	1	1
1	0	0
0	1	0
0	0	0

Possiamo dedurre il seguente comportamento:

- Quando l'input e' 1 sappiamo che gli automi stanno processando entrambi il valore 1.
- Se il simbolo e' 0, non sappiamo di preciso quale sia il valore processato - infatti a 0 corrispondono tre coppie di valori.

Definiamo x e y come gli stati attuali di D_A e D_B e definiamo la funzione di transizione come segue.

- $\delta((x,y),1) = (\delta(x,1),\delta(y,1))$
- $\delta((x,y),0) = ((\delta(x,1),\delta(y,0)),(\delta(x,0),\delta(y,1)),(\delta(x,0),\delta(y,0))$

Quindi se abbiamo il simbolo 1 facciamo avanzare entrambi gli automi; se invece abbiamo zero, dobbiamo tentare tutte le possibili strade. In questo modo stiamo costruendo un NFA per coprire tutti i possibili percorsi.

Se D_A automa accetta la parola, esistera' un percorso che porta ad uno stato accettante (stessa cosa per D_B).

Per completare l'automa, dobbiamo definirne le altre componenti:

- $\Sigma = \{0,1\}$
- $Q_S = Q_A$ x Q_B
- q = (q_A, q_B)
- F = F_A x F_B

Ex 5.3

Siano L e M due linguaggi regolari su alfabeto {0,1}. Dimostare che il linguaggio $LXORM = \{xXORy | x \in L, y \in M, |x| = |y|\}$, dove $xXORy$ e' l'and logic bit a bit. Per esempio,$101XOR001 = 100$.

Soluzione

Per questo esercizio seguire la dimostrazione dell'esercizio 5.2

Automi e Linguaggi Formali – 17/6/2021
Seconda parte – Decidibilità e complessità – Soluzione

1. *Una* macchina di Turing bidimensionale *utilizza una griglia bidimensionale infinita di celle come nastro. Ad ogni transizione, la testina può spostarsi dalla cella corrente ad una qualsiasi delle quattro celle adiacenti. La funzione di transizione di tale macchina ha la forma*

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\uparrow, \downarrow, \rightarrow, \leftarrow\},$$

dove le frecce indicano in quale direzione si muove la testina dopo aver scritto il simbolo sulla cella corrente.

Dimostra che ogni macchina di Turing bidimensionale può essere simulata da una macchina di Turing deterministica a nastro singolo.

Mostriamo come simulare una TM bidimensionale B con una TM deterministica a nastro singolo S. S memorizza il contenuto della griglia bidimensionale sul nastro come una sequenza di stringhe separate da #, ognuna delle quali rappresenta una riga della griglia. Due cancelletti consecutivi ## segnano l'inizio e la fine della rappresentazione della griglia. La posizione della testina di B viene indicata marcando la cella con $\widehat{\ }$. Nelle altre righe, un pallino $\overset{\bullet}{\ }$ indica che la testina si trova su quella colonna della griglia, ma in una riga diversa. La TM a nastro singolo S funziona come segue:

$S =$ "su input w:

1. Sostituisce w con la configurazione iniziale $\#\#w\#\#$ e marca con $\widehat{\ }$ il primo simbolo di w.
2. Scorre il nastro finché non trova la cella marcata con $\widehat{\ }$.
3. Aggiorna il nastro in accordo con la funzione di transizione di B:
   - Se $\delta(r, a) = (s, b, \rightarrow)$, scrive b non marcato sulla cella corrente, sposta $\widehat{\ }$ sulla cella immediatamente a destra. Poi sposta di una cella a destra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## finale, di una cella più a destra.
   - Se $\delta(r, a) = (s, b, \leftarrow)$, scrive b non marcato sulla cella corrente, sposta $\widehat{\ }$ sulla cella immediatamente a sinistra. Poi sposta di una cella a sinistra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## iniziale, di una cella più a sinistra.
   - Se $\delta(r, a) = (s, b, \uparrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\widehat{\ }$ sulla prima cella marcata con un pallino posta a sinistra della cella corrente. Se questa cella marcata non esiste, aggiunge una nuova riga composta solo da blank all'inizio della configurazione.
   - Se $\delta(r, a) = (s, b, \downarrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\widehat{\ }$ sulla prima cella marcata con un pallino posta a destra della cella corrente. Se questa cella non esiste, aggiunge una nuova riga composta solo da blank alla fine della configurazione.
4. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di B, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di B allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

2. *Dimostra che il seguente linguaggio è indecidibile:*

$$A_{1010} = \{\langle M \rangle \mid M \text{ è una TM tale che } 1010 \in L(M)\}.$$

Dimostriamo che A_{1010} è un linguaggio indecidibile mostrando che A_{TM} è riducibile ad A_{1010}. La funzione di riduzione f è calcolata dalla seguente macchina di Turing:

$F =$ "su input $\langle M, w \rangle$, dove M è una TM e w una stringa:

1. Costruisci la seguente macchina M_w:
   $M_w =$ "su input x:
   1. Se $x \neq 1010$, rifiuta.
   2. Se $x = 1010$, esegue M su input w.
   3. Se M accetta, *accetta*.
   4. Se M rifiuta, *rifiuta*."
2. Restituisci $\langle M_w \rangle$."

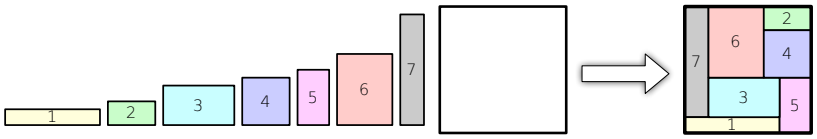
Dimostriamo che f è una funzione di riduzione da A_{TM} ad A_{1010}.

- Se $\langle M, w \rangle \in A_{TM}$ allora la TM M accetta w. Di conseguenza la macchina M_w costruita dalla funzione accetta la parola 1010. Quindi $f(\langle M, w \rangle) = \langle M_w \rangle \in A_{1010}$.
- Viceversa, se $\langle M, w \rangle \notin A_{TM}$ allora la computazione di M su w non termina o termina con rifiuto. Di conseguenza la macchina M_w rifiuta 1010 e $f(\langle M, w \rangle) = \langle M_w \rangle \notin A_{1010}$.

Per concludere, siccome abbiamo dimostrato che $A_{TM} \leq_m A_{1010}$ e sappiamo che A_{TM} è indecidibile, allora possiamo concludere che A_{1010} è indecidibile.

3. *Il problema* SetPartitioning *chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-hard.*

Il problema RectangleTiling *è definito come segue: dato un rettangolo grande e diversi rettangoli più piccoli, determinare se i rettangoli più piccoli possono essere posizionati all'interno del rettangolo grande senza sovrapposizioni e senza lasciare spazi vuoti.*



Un'istanza positiva di RectangleTiling.

Dimostra che RectangleTiling *è NP-hard, usando* SetPartitioning *come problema di riferimento.*

Dimostriamo che RectangleTiling è NP-Hard per riduzione polinomiale da SetPartitioning. La funzione di riduzione polinomiale prende in input un insieme di interi positivi $S = \{s_1, \ldots, s_n\}$ e costruisce un'istanza di RectangleTiling come segue:

- i rettangoli piccoli hanno altezza 1 e base uguale ai numeri in S moltiplicati per 3: $(3s_1, 1), \ldots, (3s_n, 1)$;
- il rettangolo grande ha altezza 2 e base $\frac{3}{2}N$, dove $N = \sum_{i=1}^{n} s_i$ è la somma dei numeri in S.

Dimostriamo che esiste un modo per suddividere S in due insiemi S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 se e solo se esiste un tiling corretto:

- Supponiamo esista un modo per suddividere S nei due insiemi S_1 e S_2. Posizioniamo i rettangoli che corrispondono ai numeri in S_1 in una fila orizzontale, ed i rettangoli che corrispondono ad S_2 in un'altra fila orizzontale. Le due file hanno altezza 1 e base $\frac{3}{2}N$, quindi formano un tiling corretto.
- Supponiamo che esista un modo per disporre i rettangoli piccoli all'interno del rettangolo grande senza sovrapposizioni né spazi vuoti. Moltiplicare le base dei rettangoli per 3 serve ad impedire che un rettangolo piccolo possa essere disposto in verticale all'interno del rettangolo grande. Quindi il tiling valido è composto da due file di rettangoli disposti in orizzontale. Mettiamo i numeri corrispondenti ai rettangoli in una fila in S_1 e quelli corrispondenti all'altra fila in S_2. La somma dei numeri in S_1 ed S_2 è pari ad $N/2$, e quindi rappresenta una soluzione per SetPartitioning.

Per costruire l'istanza di RectangleTiling basta scorrere una volta l'insieme S, con un costo polinomiale.