LFC (Linguaggi Formali e Compilatori) - Note del Corso

Edoardo Lenzi

November 15, 2017

Contents

1 Automi a stati finiti **3**
- 1.1 Thompson construction . . . 3
- 1.2 Simulare un NFA . . . 3
- 1.3 DFA . . . 5
- 1.4 Subset Construction . . . 5
- 1.5 Partition Refinement . . . 7

2 14/11/17 **8**
- 2.1 LRm(1) . . . 8
- 2.2 Algoritmo . . . 10
- 2.3 Costruzione automa simbolico . . . 10

0.0.1 Esempi Linguaggi Liberi

Essendo un linguaggio libero chiuso rispetto alla concatenazione, dati:
$L_1 = \{a^n b^n c^j \ / \ n, j \geq 0\}$ Libero
$L_2 = \{a^n b^n c^n \ / \ n, j \geq 0\}$ Libero perché concatenazione di $\{a^n b^n \ / \ n \geq 0\}$ e $\{c^j \ / \ j \geq 0\}$, entrambi liberi
$L_3 = \{a^n b^n c^n \ / \ n \geq 0\}$ Non é libero:
Suppongo L_3 libero, sia $p \in \mathbb{N}^+$, $z = a^p b^p c^p$ Allora $z \in L_3, \ |z| = 3p > p$
Spacco z in $A = a...a, \ B = b...b, C = c...c$
Siano $z = uvwxy \ \wedge \ |vwx| \leq p \ \wedge \ |vx| > 0$:

- vwx é composto da sole a in A
- vwx é composto da a in A e b in B
- vwx é composto da sole b in B
- vwx é composto da b in B e c in C
- vwx é composto da sole c in C

Considero la parola $z' = uv^0 wx^0 y$

1. $z' = a^k b^p c^p, \ k < p, \ z' \notin L_3$
3. $z' = a^p b^k c^p, \ k < p, \ z' \notin L_3$
5. $z' = a^p b^p c^k, \ k < p, \ z' \notin L_3$
2. $z' = a^k b^j c^p, \ k < p \ \vee \ j < p, \ z' \notin L_3$
4. $z' = a^p b^k c^j, \ k < p \ \vee \ j < p, \ z' \notin L_3$

Quindi visto che la parola non appartiene mai ad L_3 il linguaggio non é libero. $\square$

Quindi la classe di linguaggi liberi **non é chiusa rispetto all'intersezione**

$L_4 = \{a^n b^m c^{n+m} \ / \ n, m > 0\}$ Libero
$S \rightarrow aSc|aBc$
$B \rightarrow bBc|bc$

$L_5 = \{a^n b^m c^n d^m | n, m > 0\}$ Non libero
$L_6 = \{wcw^R \ / \ w \in \{a, b\}^+\}$ Libero
$S \rightarrow aSa|bSb|aca|bcb$

Chapter 1

Automi a stati finiti

Un NFA accetta/riconosce un certo linguaggio.

Sia N un NFA, allora il linguaggio riconosciuto/accettato da N é il set delle parole per le quali esiste almeno un cammino dallo stato iniziale di N ad uno stato finale di N.

notare che $\forall\ a \in A,\ a\epsilon = \epsilon a = a$.

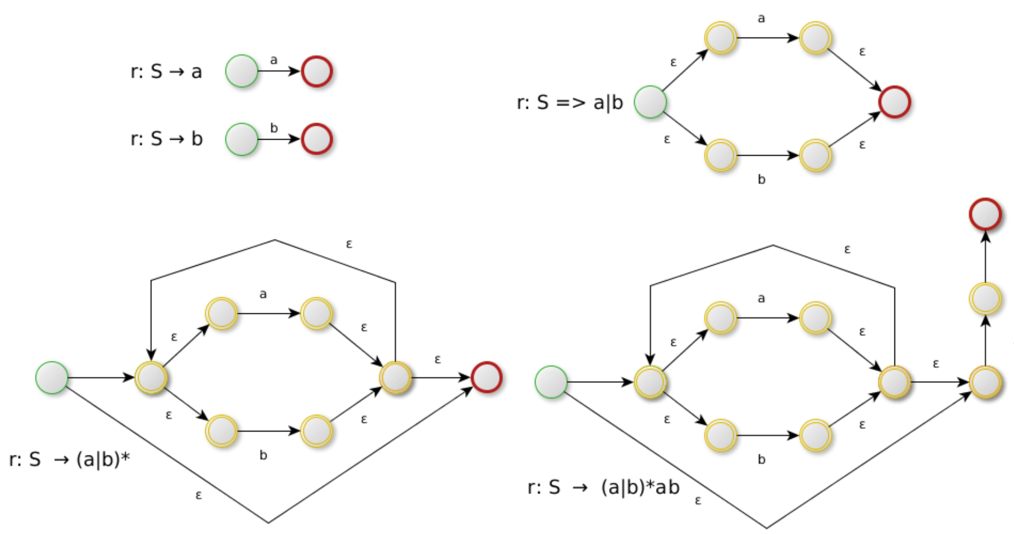
1.1 Thompson construction

input	regular expression r
output	NFA N $/\ L(N) = L(r)$

Gli NFA usati nei passi della costruzione hanno:

- un solo stato finale
- non hanno archi entranti sul nodo iniziale
- non hanno archi uscenti dal nodo finale

Lemma: Lo NFA ottenuto dalle costruzini di Thompson ha al massimo 2k stati e 4k archi, con k lunghezza della re. r. **Osservazione**: Ogni passo della costruzione introduce al massimo 2 nodi e 4 archi.



Algoritmo a complessitá $O(|r|)$

1.2 Simulare un NFA

Il backtracking consiste nel seguire un percorso e se non va bene tornare in dietro e provarne un altro finché alla fine li provo tutti mal che vada.

$N = (S, A, move_n, S_0, F)$, S insieme stati, A degli archi, S_0 stato iniziale, F set stati finali, $move_n$ funzione di transizione
$t \in S, T \subset S$
$\epsilon - closure(\{t\})$ il set degli stati S raggiungibili tramite zero o piú $\epsilon - transizioni$ da t (in pratica il nodo stesso e tutti i nodi raggiungibili con una $\epsilon - transition$).

Nota che $\forall t \in S,\ t \in \epsilon - closure(t)$
$\epsilon - closure(T) = \cup_{t \in T} \epsilon - closure(t)$

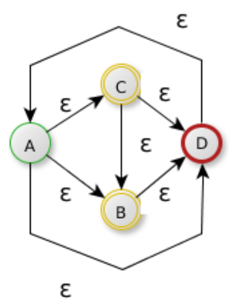
Questo algoritmo é piú performante del backtracking.

1.2.1 Algoritmo per la computazione

Strutture dati:

- pila
- bool[] alreadyOn, dimensione $|S|$
- array[][] $move_n$

```
for(int i = 0; i < |S|; i++){
    alreadyOn[i] = false;
}
closure(t, stack){
    push t onto stack;
    alreadyOn[t] = true; //posso sempre arrivare a me stesso con una epsilon-transition
    foreach(i in move_n(t, epsilon)){
        if(!alreadyOn[i]){
            closure(i, stack);
        }
    }
}
```




```
alredyOn[F F F F];
closure(A, pila vuota)
    [A] [T F F F]
        //B non e' ancora nella pila
        closure(B, [A])
            [A, B] [T T F T]
            closure(D, [A, B])
                [A, B, D] [T T F T]
        closure(C, [A, B, D])
            [A, B, C, D] [T T T T]
```

1.2.2 Algoritmo per la simulazione di un NFA

input	NFA N, w$
output	yes se $w \in L(N)$, no altrimenti

```
N = (S, A, move_n, S_0, F)
states = epsilon-closure({S_0})
symbol = nextchar()
while(symbol != $){
    states = epsilon-closure(Unione_{t in states} di move_n(t, symbol));
    symbol = newxtchar();
}
if(states intersecato F != emptyset){
    return yes;
}
return no;
```

Algoritmo a complessitá $O(|w|(n+m))$

1.3 DFA

Automa a stati finiti, deterministico; una sottoclasse degli NFA che rispettano:

$$\text{DFA} \triangleq (S, A, move_d, s_0, F)$$
$$move_d \triangleq (S \otimes A) \to S$$

- non hanno $\epsilon - transizioni$

- $\forall a \in A, s \in S,\ move_n(s, a)$ é un unico stato se **funzione di transizione totale** (al piú uno stato se **funzione di transizione parziale**)

> Sink è il nodo pozzo dove confluiscono tutte le transizioni non segnate; viene aggiunto per rendere la funzione di transizione una funzione di transizione totale

1.3.1 Linguaggio riconosciuto dal DFA

Dato il DFA D, L(D) é il linguaggio riconosciuto da D.
$L(D) = \{w = a_1, ..., a_k\ /\ \exists$ cammino in D dallo stato iniziale al finale$\}$. $\epsilon \in L(D) \iff s_0 \in F$.

1.3.2 Simulazione di un DFA con $move_d$ totale

input	w\$, DFA $D = (S, A, move_d, F)$
output	yes se $w \in L(D)$, no altrimenti

```
state = s_0;
while(symbol != $ && state != bottom){
    //move_d(s, a) = bottom <=> move_d non e' definita su (s,a)
    state = move_d(state, symbol);
    symbol = newxtchar();
}
if(state \in F)
    return yes;
return true;
```

Simulazione NFA costa $O(|w|(n+m))$ Simulazione DFA costa $O(|w|)$

1.4 Subset Construction

input	$NFA(S^n, A, move_n, S_0^n, F^n)$
output	$DFA(S^d, A, move_d, S_0^d, F^d)$

```
S_0^d = epsilon-closure({S_0^n});
 //raggruppo stati della epsilon closure in un unico stato S_0^d del DFA
states = {S_0^d};
tag S_0^d come non marcato;
```

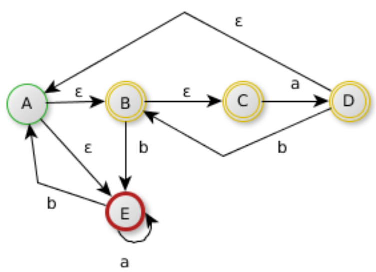
```
while(exist T in states non marcato){
   marco T;
   foreach(a in A){ //guardo ogni arco
      T_1 = epsilon-closure(U_{t in T} di move_n(t,a));
         //tutti gli stati raggiungibili con una a-transition da uno stato in T
         //poi la loro epsilon closure
      if(T_1 != emptySet){
         move_d(T, a) = T_1;
         if(T_1 !in states){
            aggiungi T_1 a states come non marcato;
         }
      }
   }
}

foreach(T in states){
   if( (T intersecato F^n) != 0){
      metti T_1 in F^d;
   }
}
```

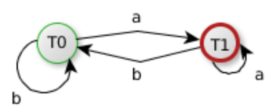
Lo stato iniziale del DFA sará la $\epsilon - closure$ dallo stato iniziale del NFA (quindi un set di stati). Considero lo stato iniziale del NFA e lo marco in grassetto poi espando T_0 con la $\epsilon - closure$ dello stato iniziale.

Dallo stato T_0 guardo per ogni arco gli stati in cui arrivo e li marco in grassetto (T_1, T_2, ...); poi espando quelli in grassetto guardando le rispettive $\epsilon - closure$.

Alla fine guardo i set degli stati se due set coincidono mergio gli stati.

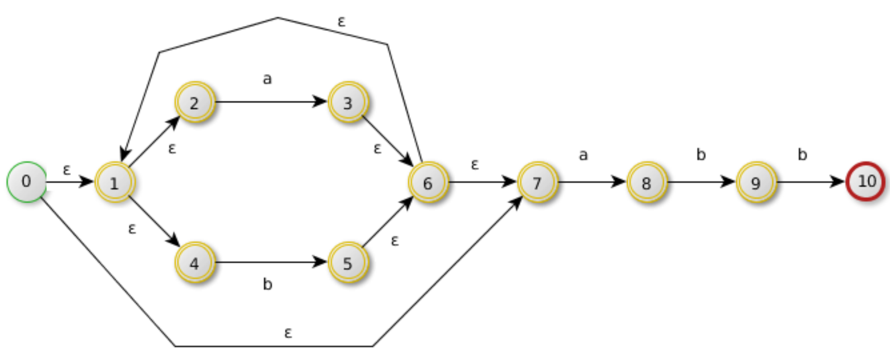
1.4.1 Esercizio



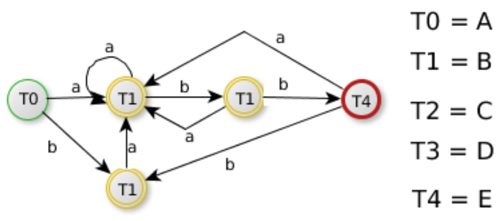


States	**a**	**b**
T0 = { **A** B C E }	T1	T2 = T0
T1 = { A B C **D** **E** }	T1	T0
T2 = { **A** B C **E** } = T0 (quindi T0 va in T0 tramite b)	come T0	come T0

1.4.2 Esercizio



States	**a**	**b**
S_0^d = { **0** 1 2 4 7 }	T1	T2
T1 = { 1 2 **3** 4 6 7 **8** }	T1	T3
T2 = { 1 2 4 **5** 6 7 }	T1	T2
T3 = { 1 2 4 **5** 6 7 9 }	T1	T4
T4 = { 1 2 4 **5** 6 7 **10** }	T1	T2



1.5 Partition Refinement

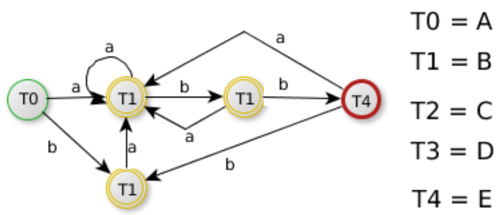
Guado gli archi, se tutta partizione punta ad un nodo dell'altra transizione con lo stesso non terminale allora va bene; altrimenti spacco la partizione.

1.5.1 Algoritmo di Partition Refinement

Input	DFA D = $\{A, A, move_d, s_0, F\}$
Output	partizione di S in blocchi equidistanti

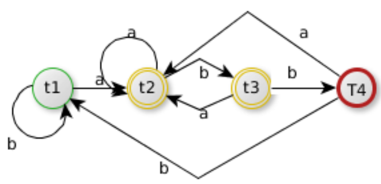
```
B_1 = F;
B_2 = S \ F;
P = {B_1, B_2};
while(exists B_i, B_j in P, exists a in A, B_i e'' partizionabile rispetto a (B_j, a)){
   sostituire B_i in P con split(B_i, (B_j, a));
}
```

1.5.2 Esempio



{ A B C D } { E }	Considero le partizioni dei terminali e non terminali
{ A B C D } { E }	Con a-transizione non esco dal primo set
{ A B C } { D } { E }	Con b-transizione vado da D in E (e A B C non vanno in E con b-transizioni)
{ A B C } { D } { E }	Con a-transizione non esco
{ A C } { B } { D } { E }	Con b-transizione vado da B in D e gli altri no quindi splitto
{ A C } { B } { D } { E }	vanno bene

Rinomino { A C } { B } { D } { E } in $t_1,\ t_2,\ t_3,\ t_4$



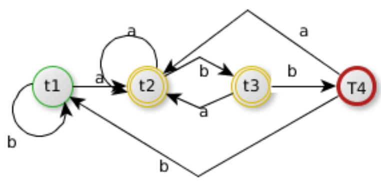
1.6 Algoritmo di minimizzazione di DFA

Input	DFA D $= \{A, A, move_d, s_0, F\}$ con $move_d$ totale
Output	minimo DFA che riconosce lo stesso linguaggio del primo

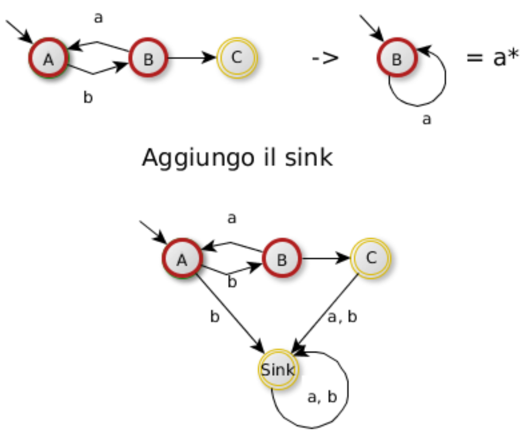
```
P = PartitionRefinement(DFA D);
// P = (B_1, ..., B_k);
foreach(B_i in P){
    if(s_o in B_i){
        t_i e'' iniziale per min(D);
    }
}

foreach(B_i in P, B_i in F){
    t_i e'' lo stato finale di min(D);
}

foreach( (B_i, a, B_j) tale che esiste s_i in B_i tali che move_d(s_i, a) = s_j){
    setto una transizione temporanea in min(D) da t_i a t_j secondo il simbolo a;
}

foreach(dead state t_i){
    rimuovere t_i e tutte le transizioni da/verso t_i;
}
tutti i temporanei residui (sia stati che transizioni) sono gli stati e le transizioni
     di min(D);
```

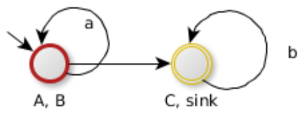
Complessitá $O(nlgn)$.

1.6.1 Esempio



Arrivato qua: rinominati { A C } { B } { D } { E } in t_1, t_2, t_3, t_4, applico la minimizzazione del DFA.





Visto {C, sinc} un sinc per il grafo, posso eliminarlo