Informatica Teorica
Appunti

Giovanni "Qua' Qua' dancer" Palma
e
Alex "Morbidelli WhatsApp" Basta

Contents

Chapter 1

Introduzione a problemi e indecidibilita

Studieremo due arie:

- **Calcolabilita**: ci chiediamo se per quel problema esistera mai un algoritmo in grado di risolverlo (decidibilita' del problema). Non li incontriamo spesso irl dato che siamo piu' spinti a fare robe che sappiamo gestire. Bisogna studiare la struttura del problema per decidere se esiste o meno un algoritmo che lo risolve.
- **Complessita**: vogliamo determinare se un problema decidibile e' "facile" o "difficile", ovvero qual'e' la sua complessita' (diverso dalla complessita' degli algoritmi)

Note:

Il professore usa in modo informale i termini "facile" e "difficile" perche' si tratta solo di un'introduzione. Piu' avanti definiremo meglio questi concetti in base a se la complessita' e' *polinomiale* o meno.

Definition 1.0.1: Problema

Relazione fra stringhe.

Example 1.0.1 (Somma)

Dati x e y calcola $x + y \to z$. E' una relazione binaria fra stringhe:

- $< (2, 3), 5 >$
- $< (4, 3), 7 >$

  $\vdots$

Dove la prima stringa e' l'input e la seconda e' l'output. Dato che elencare tutte le tuple e' impossibile, le descriviamo a parole ma stando attenti ad essere precisi. Ci servono tre elementi:

- What is input?
- What is output?
- Che relazione c'e' fra out e in?

1.1 Halting Problem

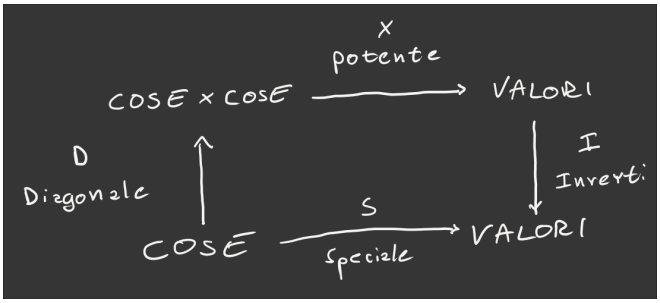
Definiamo prima per bene il problema:

- Input: Una stringa P che rapresenta il codice sorgente di un programma
- Output: Un booleano
- Relazione: Se P termina ritorna T, altrimenti F

Sembra un problema semplice, ma vedremo che in realta' e' indecidibile. Per confermare cio' dobbiamo formulare una dimostrazione formale, usando quel che viene chiamato "metodo diagonale".
Il prof non va cosi' in dettaglio per il metodo usato nella dimostrazione (dato che e' solo un esempio introduttivo), ma secondo me e' interessante e fornisce un bell'esempio per l'utilita' della Teoria delle Categorie.

1.1.1 Il Metodo Diagonale

Il *metodo diagonale* e' uno schema utile per la dimostrazione di certi teoremi. E' stato ideato per la prima volta da Cantor per dimostrare, per esempio, che $\mathbb{R}$ non e' numerabile, dove viene usata la *diagonale* di una tabella per dimostrare un assurdo. Questo metodo e' stato successivamente usato per dimostrare altre proposizioni molto importanti, come il teorema di Cantor (quello sugli insiemi potenza), il paradosso di Russel e i teoremi dell'incompletezza di Godel.
La struttura della dimostrazione generale e' la seguente:



Lo schema e' molto astratto, e' formato solo da *punti* (i vertici) e *frecce* che possono essere composte per formare altre frecce. I primi possono essere insiemi e le frecce funzioni, come vedremo. Le quattro frecce hanno delle funzioni particolari:

- La funzione "potente" X: nella Teoria delle Categorie viene definita *morfismo punto-suriettivo*(?). E' la freccia che assumiamo esista per poi dimostrare l'assurdo.
- La funzione diagonale D: dato un elemento x ritorna la coppia (x, x). Raccoglie l'essenza del metodo diagonale in quanto e' la componente che causa l'*auto-referenzialita'* che sta al cuore della dimostrazione.
- La funzione che inverte I: dato l'output di X lo *inverte* in modo da causare l'assurdita'.
- La funzione "speciale" S: composizione delle tre frecce sopra, insieme all'assunzione iniziale permette di dimostrare l'assurdo.

Note:

Lowkey non l'ho spiegato benissimo dato che non ho studiato abbastanza le sue applicazioni e so solo le basi della Teoria delle Categorie quindi non e' rigoroso per niente. Forse nel futuro ci ritornero', boh.

Finita la costruzione, i prossimi passi sono:

1. Codifica S con una *COSA*, k.
2. Cosa succede se passo k a S?

Note:

In verita' non sono sicurissimo se questi passi fanno parte del metodo generale o solo dell'halting

1.1.2 Dimostrazione Halting Problem

Andiamo ora a vedere come puo' essere dimostrato che l'halting problem e' indecidibile.
Per prima cosa dobbiamo definire la controparte concreta delle frecce e punti dello schema:

- I punti *COSE* sono insiemi di tutti i programmi o dati possibili. Sfruttando la **Numerazione di Godel** possiamo attribuire a ognuno di questi un numero naturale. Quindi *COSE* diventa $\mathbb{N}$.

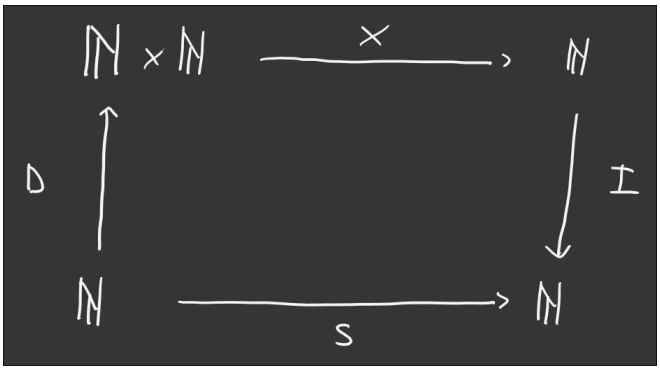
- I punti *VALORI* sono tutti i possibili output di un programma, quindi dei dati. Quindi *VALORI* diventa $\mathbb{N}$.
- La freccia X diventa il programma *HALT*, ovvero la funzione totale che prende un *Programma* P e un *Dato* D (entrambe $\in \mathbb{N}$) tale che:

$$X(P, D) = \begin{cases} \text{"termina"} & P \text{ termina su } D \\ \text{"diverge"} & P \text{ non termina su } D \end{cases}$$

  Per ora assumiamo l'esistenza di questo programma.
- La freccia D e' il programma che implementa la funzione totale $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$

$$D(x) = (x, x)$$

  che e' facile da costruire e quindi esiste.



Non e' chiaro come dobbiamo definire I in modo da dimostrare l'assurdo, quindi lavoriamo top-down.
Dato che una composizione di programmi e' un programma, anche S e' sicuramente un programma, quindi una funzione parziale che puo' terminare o divergere dato un input.
Essendo S un programma, e' possibile rappresentarlo con un numero $k \in \mathbb{N}$. Analizziamo $S(k)$:

- $S(k)$ termina: in questo caso $X(k, k) =$ "termina", quindi $S(k) = I($"termina"$)$. Per rimanere coerente I deve terminare sull'input "termina".
- $S(k)$ non termina: in questo caso $X(k, k) =$ "diverge", quindi $S(k) = I($"$diverge$"$)$. Per rimanere coerente I deve divergere sull'input "diverge".

Ma noi non vogliamo la coerenza, vogliamo costruire S in modo da dimostrare l'assurdo. Quindi definiamo I come la funzione parziale che:

- Diverge sull'input "termina"
- Termina sull'input "diverge"

Abbiamo quindi che:

- Se $X(k, k) =$ "termina", allora $S(k)$ diverge
- Se $X(k, k) =$ "$diverge$", allora $S(k)$ termina

Queste due implicazioni sono in opposizione alla definizione del programma *HALT*, che quindi non puo' esistere.

1.2 Tipi di problemi

Ci sono due categorie:

- **Ricerca**: vogliamo calcolare un risultato arbitrario, ovvero *ricercano* un output vario.
- **Decisione**: l'output e' *booleano*.

Queste due tipologie non sono completamente sganciate, esiste una relazione.
Dato un problema di ricerca, possiamo ricavare la versione "decisionale" che e' piu' facile da calcolare.

Example 1.2.1 (Somma)

Ricerca: dati a, b calcolare la somma $c = a + b$.
Decisione: dati a, b, c determinare se $c = a + b$ (vero o falso).

Per questo motivo parleremo quasi totalmente solo di problemi di decisione.

Note:

E' da qui che deriva il termine "decidibile".

1.2.1 Problemi di decisione

Theorem 1.2.1

Tutti i problemi di decisione possono essere ridotti al riconoscimento di uno specifico linguaggio.

Example 1.2.2

PATH $< G, s, t >$: vogliamo stabilire se esiste un percorso da s a t.
Mostriamo di poter trasformare questo problema decisionale in un problema che chiede se una "parola" appartiene a uno specifico linguaggio.

$$L_{\text{PATH}} = \{< G, s, t > | G \text{ e' un Grafo } \wedge s, t \text{ sono nodi del grafo } G \wedge \exists \text{ un percorso da } s \rightarrow t\}$$

Ci riduciamo quindi a decidere se una data tripla appartiene a L_{PATH} o meno.

Quindi, mentre studiamo decidibilita' dei problemi guarderemo linguaggi e i loro *automi*.

1.3 Linguaggi

1.3.1 Definizioni

Definition 1.3.1: Alfabeto

$\Sigma = \{a, b, c, ...\}$ e' l'*alfabeto*
E' un insieme di simboli.

Definition 1.3.2: Parola

w e' una parola o striga su Σ e' un concatenamento di 0 o piu' simboli di Σ.

Definition 1.3.3: Kleene

def

Definition 1.3.4: Linguaggio

Sottoinsieme delle parole che si possono costruire su un determinato alfabeto Σ:

$$L \subseteq \Sigma^*$$

Definition 1.3.5

Dato un linguaggio L **fissato** per una stringa w, decidere se:

$$w \in L$$

Il problema di appartenenza a un linguaggio L (**fissato!**) e' definito come:
dato il linguaggio L, prendere in input una parola w e decidere se $w \in L$.
input: stringa output: booleano
Perche' vogliamo usare automi? : e' un formalismo piu' basico e piu' semplice da studiare rispetto a un arbitrario linguaggio di prorammazione.

1.3.2 Automi

Vabbe' introduce gli automi, il Guerriero ci ha gia preparato adeguatamente.
Alright whatsapp.

1.3.3 Macchia di Turing

Automa piu' potente che Turing si invento per studiare i problemi importanti da chiudere, fra cui l'automatizzazione dei teoremi.
E' nato per formalizzare il calcolo automatico in un tempo in cui non esistevano i computer, era la versione "automatizzata" di lui che scriveva su carta.
Iniziamo con una spiegazione informale:
E' una macchina caratterizzata da un nastro con delle cellette
In ognuna cella c'e' un simbolo, ed e' infinito in entrambe le direzioni
Ha una testina posizionata su una cella da cui puo' leggere il simbolo
La testina puo' spostarsi a dx o sx, e a differenza degli automi visti puo' benissio tornare indietro e cambiare la decisione presa prima (backtracking)
La testina puo' anche scrivere su una cella, perdendo cio' che c'era prima (una seconda differenza)
Per noi sta roba e' un algoritmo, dato che la macchina di Turing, in quanto automa, ha vari stati di funzionamento

- Legge simbolo cella
- In base allo stato, sovrascrive qualcosa
- Sposto la testina avanti o indietro
- Cambio lo stato

Example 1.3.1

Progetta una MdT che riconosca il linguaggio

$$L = \{a^m b^m | m \geqslant 0\}$$

La stringa da decidere si trova sul nastro da sx a dx e la testina si trova sopra il primo simbolo.
Un algoritmo e' un filmato di quello che succede nella nostra testa per risolvere un problema.

- Leggi cella, se e' vuota fine (ok)
- altrimenti se e' b fine (no)
- se e' a, cancella
- spostati a dx finche' la cella e' bianca
- vai a sx di uno e leggi
- se e' a, fine (no)
- se e' b, cancella e vai a sx finche e' bianca
- muoviti di uno a dx

TODO: disegna automa

Definizioni formali

Definition 1.3.6: Macchina di Turing

E' la settupla

$$M = \{Q, q_0, F, \Sigma, \Gamma, b/, \delta\}$$

Dove:

- Q e' l'insieme di stati
- q_0 e' lo stato iniziale
- F e' l'insieme degli stati finali
- Σ e' l'insime dei simboli in input
- Γ e' l'insieme dei simboli che la macchina puo' scrivere $\Sigma \subset \Gamma$
- $b/$ e' il simbolo di cella vuota ("blank") $b/ \in \Gamma$
- δ e' la funzione

$$Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$$

La macchina di turing ha stati finiti (come gli altri automi) e il suo programma e' fissato (non cambia). Quindi come fa a eseguire qualunque computazione? Vedremo.
δ codifica le funzioni di transizioni (gli archi dell'automa)

Definition 1.3.7: Configurazione

Sia $M = \{Q, q_0, F, \Sigma, \Gamma, \not b, \delta\}$ una MdT. Si definisce *configurazione* una stringa $\alpha q \beta$ dove

- $q \in Q$ è lo stato corrente della macchina.
- $\alpha, \beta \in \Gamma^*$ sono le stringhe di simboli che si trovano a sinistra e a destra della testina.

Note:

Si può esprimere informalmente come una "fotografia" dello stato corrente di computazione della macchina:

- Stato macchina
- Posizione testina
- Cosa c'e' sul nastro

Example 1.3.2

$$aq_1abb$$

dove q_1 e' lo stato della macchina e la testina legge il carattere a dx dello stato.

$$aabq_3b/$$

sta leggendo la prima cella bianca a dx
Nota: i caratteri bianchi ci permettono di precisare la posizione della testina, ma non ne dobbiamo mettere infiniti obv

Partendo dalla configurazione, usando la funzione di transizione siamo in grado di calcolare il passo che eseguira'

Definition 1.3.8: Configurazione Accettante

Quando al suo interno si trova uno stato accettante

Definition 1.3.9: Legal Successor

Data una MdT M e sia c_1 una sua configurazione, c_2 si dice *legal successor* di c_1 rispetto a M se c_2 si ottiene da c_1 in esattamente un passo di computazione mediante l'applicazione di una transizione definita nella funzione di transizione δ.
E lo si denota con:

$$c_1 \vdash_M c_2$$

Example 1.3.3 (Legal successor)

Sia M una MdT e sia $c_1 = aq_1b$ una sua configurazione. Se $\delta(q_1, b) = (q_2, a, \rightarrow)$, allora $c_2 = aaq_2b$ e' legal successor di c_1 rispetto a M

Definition 1.3.10: Configurazione iniziale

si definisce configurazione iniziale di una MdT M e di una stringa $w \in \Sigma^*$ la configurazione $c_1 = q_0w$

Definition 1.3.11: Configurazione Finale

Configurazione che non ammette legal successor secondo M

Definition 1.3.12: Computazione Parziale

Data una MdT M, una *computazione parziale* per M e' una sequenza:

$$c_1, c_2, \ldots, c_n$$

dove

$$c_1 \vdash_M c_2 \vdash_M \cdots \vdash_M c_n$$

Note:

Non è detto che c_1 sia il punto di partenza e c_n sia l'ultima configurazione della computazione, per questo è parziale.

Definition 1.3.13: Computazione

Data una MdT M e una stringa $w \in \Sigma^*$, una *computazione* per M e' una computazione parziale $c_1, c_2, \ldots, c_n$ per M dove c_1 è una configurazione iniziale e c_n è una configurazione finale

Possiamo ora definire quando una MdT accetta un input

Definition 1.3.14: Accettazione

Una MdT accetta w se si arresta su uno stato accettante, e rifuita w se non si arresta su uno stato accettante.

Note:

Da questa definizione, se una MdT non si arresta allora l'input non e' accettato. Quindi per non accettare puo' anche divergere. Questa roba qua per noi e' un po' cringe.

Definition 1.3.15: Linguaggio di una MdT

Linguaggio

$$\mathcal{L}(M) = \{w | M(w) = 1\}$$

Definition 1.3.16: Decide

Sia L un linguaggio, M *decide* L sse $\forall w \in \Sigma^*$

$$w \in L \implies M \text{ accetta } w$$

$$w \notin L \implies M \textit{ si arresta} \text{ e dice di "no" } w$$

Definition 1.3.17: Riconosce

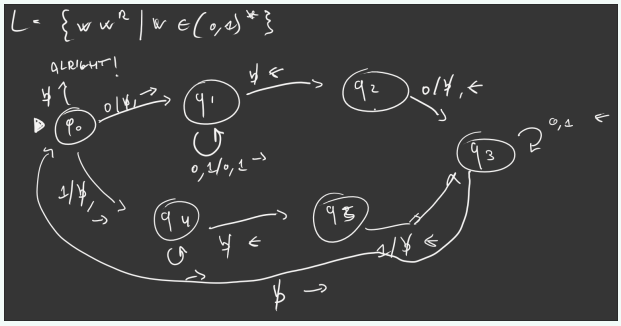
Sia L un linguaggio, M *riconosce* L sse $\forall w \in \Sigma^*$

$$w \in L \implies M \text{ accetta } w$$

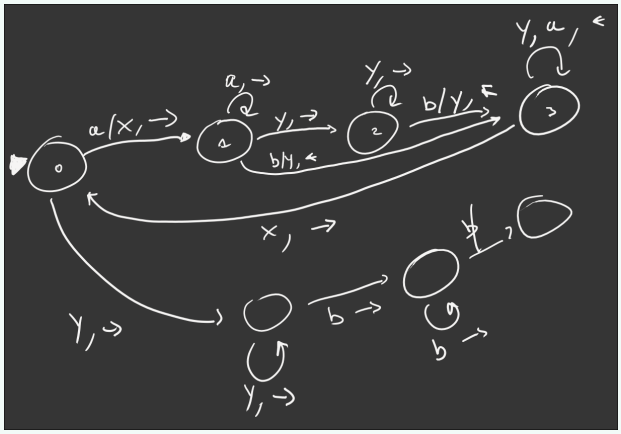
$$w \notin L \implies M \text{ non accetta } w$$

Esercizi

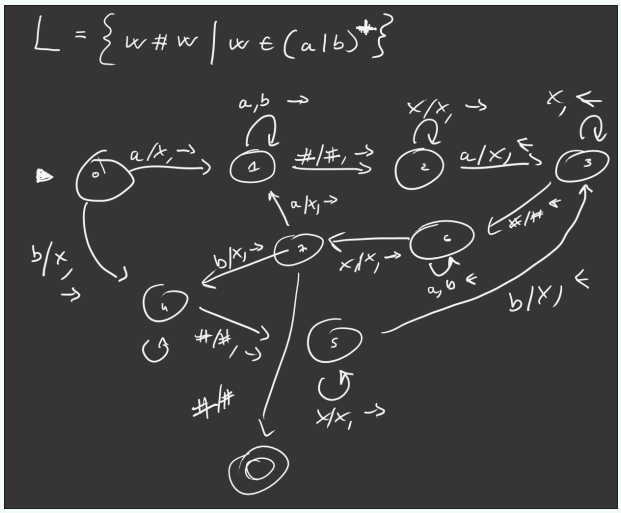
Example 1.3.4



Example 1.3.5



Example 1.3.6



Chapter 2

Macchine di Turing e Linguaggi

2.1 Linguaggi

2.1.1 Definizioni

Definition 2.1.1: Alfabeto

$\Sigma = \{a, b, c, ...\}$ e' l'*alfabeto*
E' un insieme di simboli.

Definition 2.1.2: Parola

w e' una parola o striga su Σ e' un concatenamento di 0 o piu' simboli di Σ.

Definition 2.1.3: Kleene

def

Definition 2.1.4: Linguaggio

Sottoinsieme delle parole che si possono costruire su un determinato alfabeto Σ:

$$L \subseteq \Sigma^*$$

Definition 2.1.5

Dato un linguaggio L **fissato** per una stringa w, decidere se:

$$w \in L$$

Il problema di appartenenza a un linguaggio L (**fissato!**) e' definito come:
dato il linguaggio L, prendere in input una parola w e decidere se $w \in L$.
input: stringa output: booleano
Perche' vogliamo usare automi? : e' un formalismo piu' basico e piu' semplice da studiare rispetto a un arbitrario linguaggio di prorammazione.

2.1.2 Automi

Vabbe' introduce gli automi, il Guerriero ci ha gia preparato adeguatamente.
Alright whatsapp.

2.2 Macchia di Turing

Automa piu' potente che Turing si invento per studiare i problemi importanti da chiudere, fra cui l'automatizzazione dei teoremi.
E' nato per formalizzare il calcolo automatico in un tempo in cui non esistevano i computer, era la versione "automatizzata" di lui che scriveva su carta.
Iniziamo con una spiegazione informale:
E' una macchina caratterizzata da un nastro con delle cellette
In ognuna cella c'e' un simbolo, ed e' infinito in entrambe le direzioni
Ha una testina posizionata su una cella da cui puo' leggere il simbolo
La testina puo' spostarsi a dx o sx, e a differenza degli automi visti puo' benissio tornare indietro e cambiare la decisione presa prima (backtracking)
La testina puo' anche scrivere su una cella, perdendo cio' che c'era prima (una seconda differenza)
Per noi sta roba e' un algoritmo, dato che la macchina di Turing, in quanto automa, ha vari stati di funzionamento

- Legge simbolo cella
- In base allo stato, sovrascrive qualcosa
- Sposto la testina avanti o indietro
- Cambio lo stato

Example 2.2.1

Progetta una MdT che riconosca il linguaggio

$$L = \{a^m b^m | m \geqslant 0\}$$

La stringa da decidere si trova sul nastro da sx a dx e la testina si trova sopra il primo simbolo.
Un algoritmo e' un filmato di quello che succede nella nostra testa per risolvere un problema.

- Leggi cella, se e' vuota fine (ok)
- altrimenti se e' b fine (no)
- se e' a, cancella
- spostati a dx finche' la cella e' bianca
- vai a sx di uno e leggi
- se e' a, fine (no)
- se e' b, cancella e vai a sx finche e' bianca
- muoviti di uno a dx

TODO: disegna automa

2.2.1 Definizioni formali

Definition 2.2.1: Macchina di Turing

E' la settupla

$$M = \{Q, q_0, F, \Sigma, \Gamma, b/, \delta\}$$

Dove:

- Q e' l'insieme di stati
- q_0 e' lo stato iniziale
- F e' l'insieme degli stati finali
- Σ e' l'insime dei simboli in input
- Γ e' l'insieme dei simboli che la macchina puo' scrivere $\Sigma \subset \Gamma$
- $b/$ e' il simbolo di cella vuota ("blank") $b/ \in \Gamma$
- δ e' la funzione

$$Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$$

La macchina di turing ha stati finiti (come gli altri automi) e il suo programma e' fissato (non cambia). Quindi come fa a eseguire qualunque computazione? Vedremo.
δ codifica le funzioni di transizioni (gli archi dell'automa)

Definition 2.2.2: Configurazione

E' una "fotografia" dello stato corrente di computazione della macchina:

- Stato macchina
- Posizione testina
- Cosa c'e' sul nastro

Example 2.2.2

$$aq_1abb$$

dove q_1 e' lo stato della macchina e la testina legge il carattere a dx dello stato.

$$aabq_3b/$$

sta leggendo la prima cella bianca a dx
Nota: i caratteri bianchi ci permettono di precisare la posizione della testina, ma non ne dobbiamo mettere infiniti obv

Partendo dalla configurazione, usando la funzione di transizione siamo in grado di calcolare il passo che eseguira'

Definition 2.2.3: Configurazione Accettante

Quando al suo interno si trova uno stato accettante

Definition 2.2.4: Legal Successor

Data una MdT M e sia c_1 una sua configurazione, c_2 si dice *legal successor* di c_1 sse:

$$c_1 \vdash_M c_2$$

Definition 2.2.5: Configurazione Finale

Configurazione che non ha legal successor

Definition 2.2.6: Computazione Parziale

Data una MdT M, una *computazione parziale* per M e' una sequenza:

$$c_1, c_2, ...c_n$$

dove

$$c_1 \vdash_M c_2 \vdash_M ... \vdash_M c_n$$

Note:

Non e' detto che c_1 sia il punto di partenza e c_n sia l'ultima configurazione della configurazione, per questo e' parziale.

Definition 2.2.7: Computazione

Data una MdT M e una stringa $w = w_1w_2...w_l$, una *computazione* per M e' una computazione parziale per M dove

$$c_1 = q_0w_1w_2...w_l$$

e c_n e' una configurazione finale

Possiamo ora definire quando una MdT accetta un input

Definition 2.2.8: Accettazione

Una MdT accetta w se si arresta su uno stato accettante, e rifuita w se non e' vero che si arresti in uno stato accettante.

Note:

Da questa definizione, se una MdT non si arresta allora l'input non e' accettato. Quindi per non accettare puo' anche divergere. Questa roba qua per noi e' un po' cringe.

Definition 2.2.9: Linguaggio di una MdT

Linguaggio

$$\mathcal{L}(M) = \{w | M(w) = 1\}$$

Definition 2.2.10: Decide

Sia L un linguaggio, M *decide* L se $\forall w \in \Sigma^*$

$$w \in L \implies M \text{ accetta } w$$

$$w \notin L \implies M \textit{ si arresta} \text{ e dice di "no" } w$$

Definition 2.2.11: Riconosce

Sia L un linguaggio, M *riconosce* L se $\forall w \in \Sigma^*$

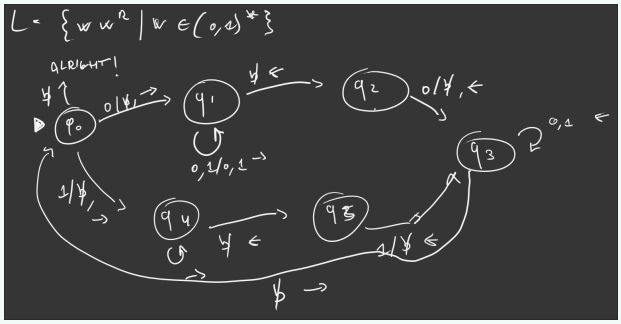
$$w \in L \implies M \text{ accetta } w$$

$$w \notin L \implies M \text{ non accetta } w$$

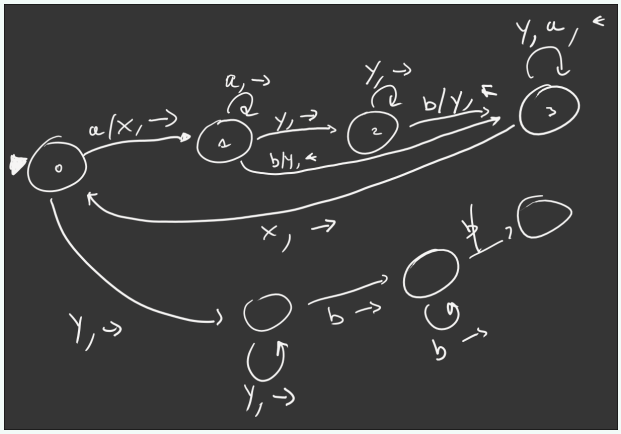
2.2.2 Esercizi

TODO: sarebbe figo fare gli automi in latex, se ho sbatti ci provo. Ah e alcune soluzioni sono imprecise le dovrei fixare.

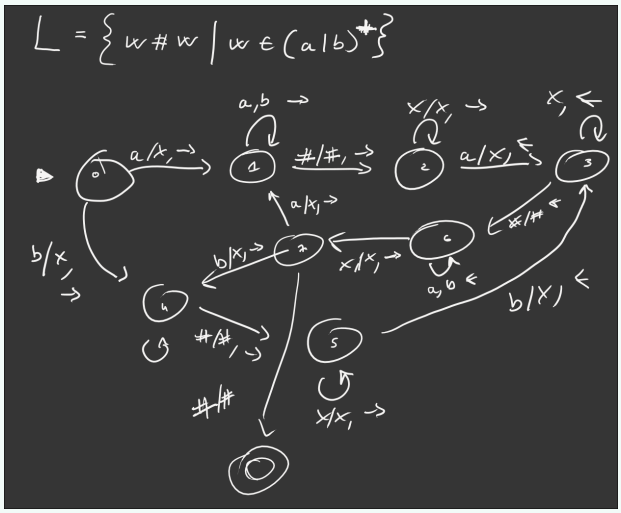
Example 2.2.3



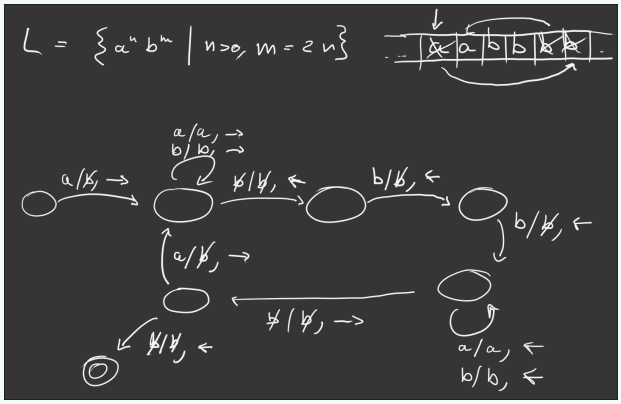
Example 2.2.4



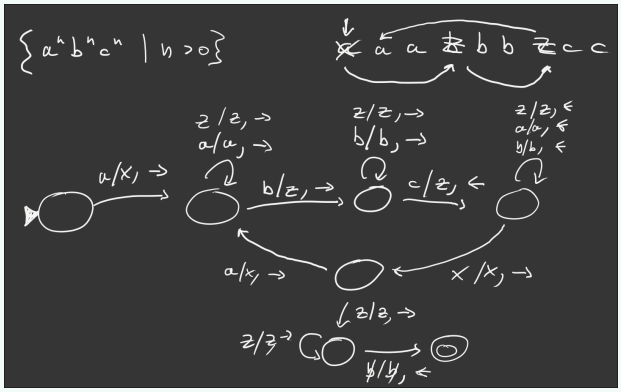
Example 2.2.5



Example 2.2.6



Example 2.2.7



2.3

Le macchine che abbiamo visto finora sono molto tediose.
Per esempio, per riconoscere il linguaggio $L = \{ww^R | w \in (0|1)^*\}$ e' stato necessario costruire due rami (uno per lo 0 e uno per l'1). Sarebbe comodo poter usare un solo ramo che in qualche modo codifica entrambe i rami,

vediamo come possiamo fare:

Definition 2.3.1: Memoria nello stato

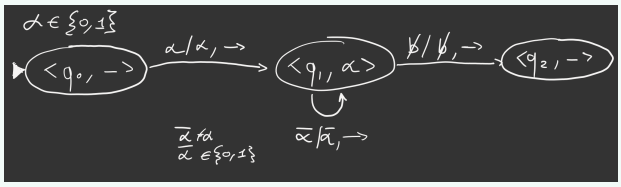
Data una MdT, si dice che questa ha **memoria** (nello stato) se ad ogni stato e' associata una memoria di uno o piu' simboli:

$$\langle q_1, -, ..., - \rangle$$

Il numero di simboli in memoria e' fissato, ovvero non puo' cambiare durante l'esecuzione.

Example 2.3.1

Vogliamo riconoscere $L = 10^*|01^*$



Notiamo che la macchina con memoria sopra non e' altro che una rappresentazione piu' compatta di una normale MdT con due rami al posto di uno.

Proposition 2.3.1 Equipotenza macchine con memoria

Data una MdT con memoria nello stato, e' sempre possibile costruire una MdT senza memoria che riconosce lo stesso linguaggio.

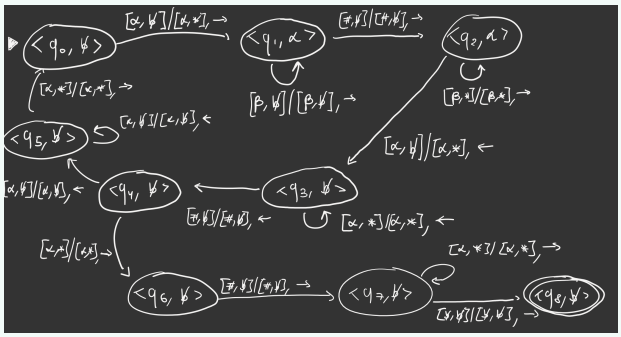
Dimostrazione: Boh in verita' non dimostra sto risultato, dice solo che alla fine le macchine con memoria sono un prodotto cartesiano di stati con simboli.

Definition 2.3.2: Macchine Multitraccia

Multiple tracce lette in sincrono, la testina si muove tutta insieme ma possiamo leggere e scrivere singolarmente sulle tracce.

Example 2.3.2

Vogliamo riconoscere $L = \{w\#w | w \in (a|b)^+\}$:



Proposition 2.3.2 Equipotenza macchine multitraccia

Data una macchina multitraccia, e' sempre possibile costruire una MdT classica che riconosce lo stesso linguaggio.

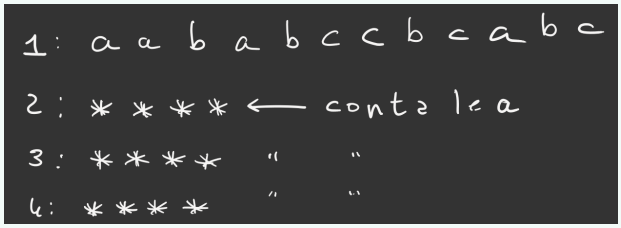
Dimostrazione: Data una macchina multitraccia di n tracce con alfabeti Γ_i per ogni traccia i, possiamo costruire una MdT con $\Gamma = \Gamma_1 \times ... \times \Gamma_n$ che si comporta esattamente come la macchina multitraccia.

Definition 2.3.3: Macchine Multinastro

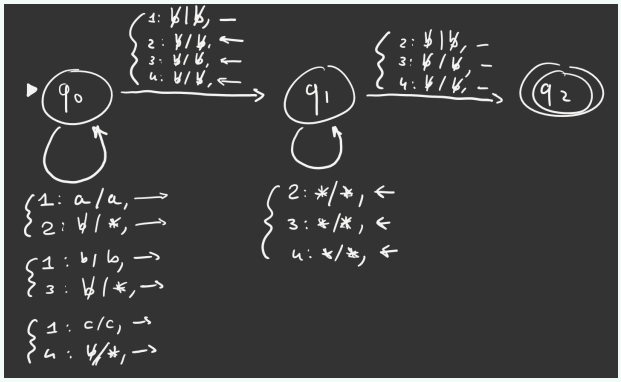
MdT che ha piu' nastri con testine indipendenti. Il numero dei nastri e' fissato a costruzione. E' possibile che una testina rimanga ferma.

Example 2.3.3

Costruiamo la macchina multinastro che riconosce $L = \{w | w \in (a|b|c)^* \wedge$ il numero delle a, b e c sono uguali$\}$. Vediamo l'idea:



Ora costruiamo l'automa:



Proposition 2.3.3 Equipotenza di macchine multinastro

Data una macchina multinastro, e' sempre possibile costruire una MdT classica che riconosce lo stesso linguaggio.

Dimostrazione: Data una macchina multinastro e' possibile costruire una multitraccia equivalente con $2n$ tracce (dove n e' il numero dei nastri) semplicemente usando le tracce dispari come "puntatori" che indicano l'ipotetica posizione della testina, come se fosse multinastro.
Ad ogni passo, collochiamo inizialmente la testina sul primo puntatore a sx per poi scansionare tutto fino all'ultimo puntatore. I simboli sotto ogni puntatore vengono salvati nella memoria, e la testina ritorna indietro cambiando in modo adeguato i simboli puntati e la posizione dei puntatori.

E' il modello di calcolo piu' flessibile che abbiamo a disposizione. Ma abbiamo visto che per simulare il suo comportamento con una macchina multitraccia impieghiamo piu' transizioni, ma quante di piu'?

Dimostrazione: Analizziamo il caso peggiore dopo m passi. Questo avviene quando una testina della macchina si sposta a sx ad ogni passo e una si sposta a dx ad ogni passo, in modo che distino $2m$ celle.
Quindi una macchina multinastro avra' bisogno di percorrere due volte la distanza (passaggio in avanti e indietro) e deve poi aggiornare ogni puntatore (uno per ogni nastro), che nel caso peggiore comporta altre due transizioni. Alla fine abbiamo che un solo passo di una macchina multinastro (k nastri) richiede nel caso peggiore $4m + 2k$ passi di una macchina multitraccia.

Possiamo quindi dire che una computazione che richiede n passi ad una macchina multinastro richiedera'

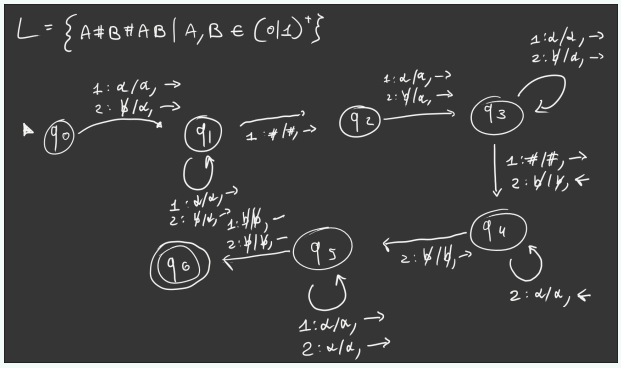
$$\sum_{i=1}^{n} 4i + 2k \leqslant \sum_{i=1}^{n} 4n + 2k \leqslant n(4n + 2k)$$

Quindi e' $O(n^2)$.

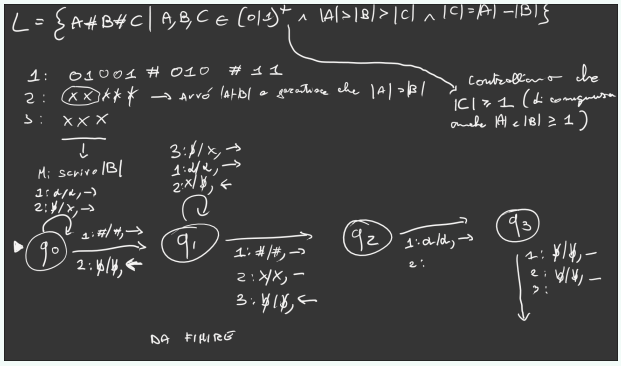
2.4 Macchine di Turing Multinastro

2.4.1 Esercizi

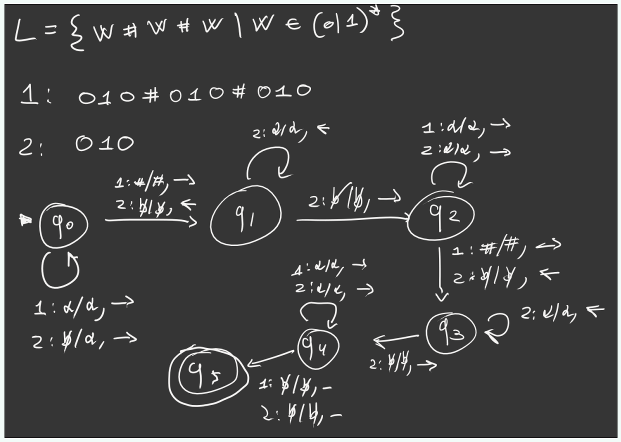
Example 2.4.1



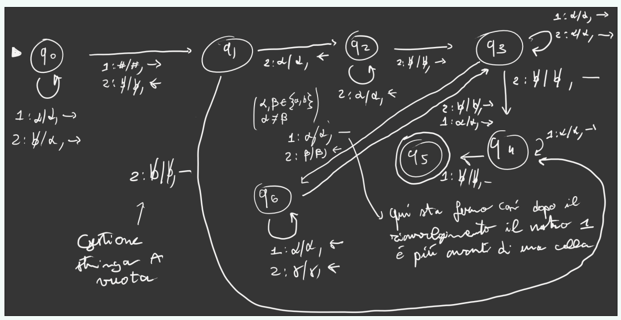
Example 2.4.2



Example 2.4.3


L = { w # w # w | w ∈ (0|1)* }
1: 010#010#010
2: 010

Example 2.4.4


qui sta fermo così dopo il riavvolgimento il nastro 1 è più avanti di una cella

Chapter 3

Riduzioni

Chapter 4

Teorema di Rice

Chapter 5

Complessita Strutturale

Chapter 6

Complessita Spaziale

Chapter 7

Oracoli e Classi Funzionali