Elementi di Calcolo e Complessità

Federico Matteoni

A.A. 2019/20

Indice

Introduzione

Prof. Pierpaolo Degano pierpaolo.degano@unipi.it
Con Giulio Masetti giulio.masetti@isti.snr.it
Esame: compitini/scritto + orale

Capitolo 1

Calcolabilità

1.1 Teoria della Calcolabilità

Illustra **cosa può essere calcolato da un computer** senza limitazioni di risorse come spazio, tempo ed energia. Vale a dire:

- Quali sono i **problemi *solubili*** mediante una **procedura effettiva** (qualunque linguaggio su qualunque macchina)?
- Esistono **problemi *insolubili***? Sono interessanti, realistici, oppure puramente artificiali?
- Possiamo raggruppare i problemi in **classi**?
- Quali sono le **proprietà** delle classi dei problemi solubili?
- Quali sono le relazioni tra le classe dei problemi insolubili?

Astrazione Utilizzeremo **termini astratti per descrivere la possibilità di eseguire un programma ed avere un risultato**. Questa astrazione è un **modello** che non tiene conto di dettagli al momento irrilevanti.
Un po' come l'equazione per dire quanto ci mette il gesso a cadere che non tiene conto delle forze di attrito dell'aria.

Problema della Decisione Un problema è risolto se si conosce una **procedura** che permette di decidere con un numero **finito** di operazioni di decedere se una proposizione logica è vera o falsa.

1.2 Algoritmo

Un algoritmo è un insieme **finito** di istruzioni.

Istruzioni Elementi da un insieme di **cardinalità finita** ed ognuna ha **effetto limitato** (localmente e "*poco*") sui dati (che devono essere **discreti**). Un'istruzione deve richiedere tempo finito per essere elaborata.

Computazione Successione di istruzioni finite in cui ogni passo dipende solo dai precedenti. Verificando una porzione finita dei dati (**deterministico**). Non c'è limite alla memoria necessaria al calcolo (è finita ma illimitata). Neanche il tempo è limitato (necessario al calcolo). Tanto tempo e tanta memoria quante ce ne servono.

Un'eccezione a questa definizione di algoritmo è costituita dalle macchine concorrenti/interattive, dove gli input variano nel tempo. Inoltre vi sono formalismi che tengono conto di algoritmi probabilistici e stocastici. Altre eccezioni sono gli algoritmi non deterministici, ma per ognuno di essi esiste un algoritmo deterministico equivalente (Teorema `3.3.6`)

1.3 Macchina di Turing

Introdotta da **Alan Turing** nel 1936, confuta la speranza "*non ignorabimus*" di poter risolvere qualsiasi cosa con un programma.
Turing originariamente la presenta supponendo di aver un impiegato precisissimo ma stupido, con una pila di fogli di carta ed una penna, ed un foglio di carta con le istruzioni che esegue con estrema diligenza. Non capisce quello che fa, e si chiama "**computer**".

Struttura matematica Una Macchina di Turing (MdT) è una quadrupla:

$$\mathrm{M} = (\mathrm{Q}, \sum, \delta, \mathrm{q}_0)$$

Q = $\{q_i\}$ è l'insieme finito degli **stati** in cui si può trovare la macchina.
Indicheremo con lo stato speciale h la fine corretta della computazione, h non $\in$ Q.

$\sum = \{\sigma, \sigma' \dots\}$ è l'insieme finito di **simboli**. Ci sono elementi che devono per forza esistere:

\# carattere **bianco**, vuoto

$\triangleright$ carattere di inizio della memoria, chiamato **respingente**, che funziona come un inizio file

$\delta \subseteq (\mathrm{Q} \times \sum) \to (\mathrm{Q}' \bigcup \{h\}) \times \sum' \times \{\mathrm{L}, \mathrm{R}, -\}$ è **funzione di transizione**.
Mantiene determinismo perché funzione, ad un elemento associa un solo elemento (la transizione è univoca).
Transizioni finite perché prodotto cartesiano di insiemi finiti.
$\delta(\mathrm{q}, \triangleright) = (\mathrm{q}', \triangleright, \mathrm{R})$, cioè se sono a inizio file possono solo andare a destra.
Può essere vista come una relazione di transizione, $\delta \subseteq (Q \times \sum) \times (Q \bigcup \{h\}) \times \sum \times \{L, R, -\}$

$\mathrm{q}_0 \in \mathrm{Q}$ lo **stato iniziale**

Mappatura a coda di rondine, bigezione tra (m, n) $\to$ k, cioè $N^2 \to N$.
Costruire un modello per il calcolo dopo aver posto delle condizioni affinché qualcosa si possa chiamare algoritmo.

1.3.1 $\sum$

$\sum^0 = \{\epsilon\}$, con ϵ = parola vuota, che non contiene caratteri

$\sum^{i+1} = \sum \cdot \sum^i = \{\sigma \cdot u \mid \sigma \in \sum \wedge u \in \sum^i\}$

$\sum^* = \bigcup_{i \in N} \sum^i$, insieme di tutte le possibili combinazioni di simboli

$\sum^f = \sum^* \cdot (\sum - \{\#\} \bigcup \{\epsilon\})$, cioè linsieme di tutte le stringhe che terminano con un carattere non bianco ma può terminare con la stringa vuota

Esempio $\sum_B = \{0, 1\} \longrightarrow \sum_B^* = \{\epsilon, 0, 1, 01, 10, 010, 110010, \dots\}$ tutti i numeri binari

1.3.2 Transizioni

La **situazione corrente** di una macchina di Turing può essere scritto come (q, u, σ, v) dove:

q è lo **stato attuale**, $q \in Q$

u è la **stringa a sinistra** del carattere corrente, $u \in \sum^*$

σ è il **carattere corrente**, $\sigma \in \sum$

v è il **resto della stringa** che termina con un carattere non nullo, $v \in \sum^f$

Può anche essere più comodamente espressa come $(q, u \underline{\sigma} v)$

1.3.3 Computazione

Una computazione è una transizione $(q, x) \longrightarrow (q', \omega)$. Una macchina di Turing parte **sempre** da $(q_0, \trianglerighteq x)$.
Ogni computazione può esprimere il numero di passi necessari, ad esempio $\gamma \longrightarrow^n \gamma'$.
$\forall$ *computazione* $\gamma \Rightarrow \gamma \longrightarrow^0 \gamma$. Inoltre se $\gamma \longrightarrow \gamma' \wedge \gamma' \longrightarrow^n \gamma''$ allora $\gamma \longrightarrow^{n+1} \gamma''$

Esempio Macchina di Turing che esegue la semplice somma di due semplici numeri romani.

q	σ	$\delta(q,\sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	I	(q_0, I, R)
q_0	$+$	(q_1, I, R)
q_1	I	(q_1, I, R)
q_1	$\#$	$(q_2, \#, L)$
q_2	I	$(h, \#, -)$

$(q_0, \underline{\triangleright} II + III) \longrightarrow (q_0, \triangleright \underline{I} I + III) \longrightarrow (q_0, \triangleright I \underline{I} + III) \longrightarrow$
$(q_0, \triangleright II \ \underline{+} \ III) \longrightarrow (q_1, \triangleright III \underline{I} III) \longrightarrow (q_1, \triangleright IIII \underline{I} II) \longrightarrow$
$(q_1, \triangleright IIIII \underline{I}) \longrightarrow (q_1, \triangleright IIIIII \underline{\#}) \longrightarrow (q_2, \triangleright IIIIII \underline{I}) \longrightarrow$
$(h, \triangleright IIIII)$

Esempio Macchina di Turing che verifica se una stringa di lettere a, b è palindroma o no.

q	σ	$\delta(q,\sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	a	$(q_A, \triangleright, R)$
q_0	b	$(q_B, \triangleright, R)$
q_0	$\#$	$(h, \#, -)$
q_A	a/b	$(q_A, a/b, R)$
q_A	$\#$	$(q_{A'}, \#, L)$
$q_{A'}$	a	$(q_R, \#, L)$
q_B	a/b	$(q_B, a/b, R)$
q_B	$\#$	$(q_{B'}, \#, L)$
$q_{B'}$	a	$(q_R, \#, L)$
q_R	a/b	$(q_R, a/b, R)$
q_R	$\triangleright$	$(q_0, \triangleright, R)$

$(q_0, \underline{\triangleright} abba) \longrightarrow (q_0, \triangleright \underline{a} bba) \longrightarrow (q_A, \triangleright \triangleright \underline{b} ba) \longrightarrow$
$(q_A, \triangleright \triangleright b \underline{b} a) \longrightarrow (q_A, \triangleright \triangleright bb \underline{a}) \longrightarrow (q_A, \triangleright \triangleright bba \underline{\#}) \longrightarrow$
$(q_{A'}, \triangleright \triangleright bb \underline{a}) \longrightarrow (q_R, \triangleright \triangleright b \underline{b}) \longrightarrow (q_R, \triangleright \triangleright \underline{b} b) \longrightarrow (q_R, \triangleright \underline{\triangleright} bb)$
$\longrightarrow (q_0, \triangleright \triangleright \underline{b} b) \longrightarrow (q_B, \triangleright \triangleright \triangleright \underline{b}) \longrightarrow (q_B, \triangleright \triangleright \triangleright b \underline{\#}) \longrightarrow$
$(q_{B'}, \triangleright \triangleright \triangleright \underline{b}) \longrightarrow (q_R, \triangleright \triangleright \underline{\triangleright}) \longrightarrow (h, \triangleright \triangleright \triangleright)$

1.4 Linguaggi di Programmazione

Un primo formalismo di algoritmo, come abbiamo visto, è la **macchina di Turing**: attenendosi alle richieste di tempo e spazio arbitrariamente grandi ma finiti, risolve un **problema**.
Un secondo formalismo sono i **linguaggi di programmazione**.

1.4.1 Sintassi

Sintassi astratta Definiamo la **sintassi** dello scheletro di un semplice linguaggio di programmazione imperativo.
Una **sintassi astratta** è una sintassi non concreta, cioè che non tiene conto di alcune cose come la procedenza tra gli operatori.
Sintassi

Expr → E ::= x | n | E + E | E · E | E - E

Bexpr → B ::= tt | ff | E < E | ¬B | B ∨ B

Comm → C ::= skip | x = E | C;C | if B then C else C | for i = E to E do C | while B do C

Abbiamo una serie di insiemi da definire ulteriormente

x ∈ Var, l'insieme delle **variabili**

n ∈ N, **numeri naturali**.

Abbiamo anche la **memoria** per poter **assegnare ad una variabile il suo significato**

σ : Var $\rightarrow_{fin} N$
Si dice "a dominio finito", indicata dal fin sotto la freccia, per indicare che il dominio Var ha cardinalità finita.
Var dominio è quindi un sottoinsieme di Var insieme delle variabili che sarebbe infinito.
La memoria si può aggiornare, diventando $\sigma' = \sigma[\mathrm{x} \mapsto \mathrm{n}]$.
Ad esempio, $\sigma'(y) = \mathrm{n}$ se y = x, altrimenti $\sigma'(y) = \sigma(y)$

1.4.2 Funzioni di Valutazione

Inoltre, per valutare le espressioni generate dalla grammatica, servono delle **funzioni di valutazione**. Esse **trovano il significato di ogni espressione**

Funzione di valutazione delle espressioni
$\mathscr{E} : \text{Expr} \times (\text{Var} \to N) \to N$
La sua **semantica denotazionale** è la seguente

$$\mathscr{E}[\text{x}]_\sigma = \sigma(\text{x}) \qquad \mathscr{E}[\text{E}_1 \pm \text{E}_2]_\sigma = \mathscr{E}[\text{E}_1]_\sigma \pm \mathscr{E}[\text{E}_2]_\sigma$$
$$\mathscr{E}[\text{n}]_\sigma = \text{n}$$

Importante notare come gli operatori $+$, $-$, $\cdot$ *dentro* le espressioni siano dei **semplici token denotazionali**, mentre sono gli operatori *valutati* ad eseguire il vero e proprio calcolo. Per chiarire questo aspetto, facciamo un esempio. Valutiamo con la nostra funzione $\mathscr{E}[\text{E}_1 + \text{E}_2]_\sigma = \mathscr{E}[\text{E}_1]_\sigma$ *più* $\mathscr{E}[\text{E}_2]_\sigma$. Se non definiamo l'operatore "*più*", allora se poniamo $\sigma(\text{x}) = 25$ la valutazione

$$\mathscr{E}[3 + \text{x}]_\sigma = \mathscr{E}[3]_\sigma \textit{ più } \mathscr{E}[\text{x}]_\sigma = 3 \textit{ più } 25 = 42$$

è corretta quanto

$$\mathscr{E}[3 + \text{x}]_\sigma = \mathscr{E}[3]_\sigma \textit{ più } \mathscr{E}[\text{x}]_\sigma = 3 \textit{ più } 25 = 28$$

Ovviamente utilizzeremo la valutazione specificata in precedenza e gli operatori aritmetici assumeranno il loro significato standard.
L'unica eccezione è l'operatore $-$, che nel nostro caso sarà il **meno limitato** dal simbolo $\dot{-}$, la cui unica differenza è che non può dare un risultato inferiore a 0. Ad esempio, $5 \dot{-} 7 = 0$

Funzione di valutazione di espressioni booleane
$\mathscr{B} : \text{Bexpr} \times (\text{Var} \to N) \to \{\text{tt}, \text{ff}\}$
La cui **semantica denotazionale** è la seguente

$$\mathscr{B}[\text{tt}]_\sigma = \text{tt} \qquad \mathscr{B}[\neg \text{B}]_\sigma = \neg \mathscr{B}[\text{B}]_\sigma$$
$$\mathscr{B}[\text{ff}]_\sigma = \text{ff}$$
$$\mathscr{B}[\text{E}_1 < \text{E}_2]_\sigma = \mathscr{E}[\text{E}_1]_\sigma < \mathscr{E}[\text{E}_2]_\sigma \qquad \mathscr{B}[\text{B}_1 \vee \text{B}_2]_\sigma = \mathscr{B}[\text{B}_1]_\sigma \vee \mathscr{B}[\text{B}_2]_\sigma$$

Anche qua vale il medesimo discorso sulla definizione sugli effettivi operatori.

1.4.3 Semantica Operazione Strutturale

Structural Operationale Semantics Metodo attraverso il quale viene fornita la semantica dei comandi. Parte da un **insieme di configurazioni** Γ

$$\Gamma = \{(\text{C}, \sigma) \mid \text{FV(C)} \subset \text{dom}(\sigma)\} \cup \{\sigma\}$$

dove FV(C) sono le **variabili del programma** e con $\text{FV(C)} \subset \text{dom}(\sigma)$ si richiede che tutte le variabili del programma abbiano un valore nella memoria fornita. Si fa l'unione con la sola memoria σ perché la situazione finale è $(\ ,\ \sigma)$ che, analogamente allo stato fittizio h nella macchina di Turing, segnala la fine dell'esecuzione. Inoltre si hanno le **transizioni** $\to$

$$\to \subset \Gamma \times \Gamma$$

Definiamo quindi un **insieme di transizioni** $(\Gamma, \to)$ tramite delle **regole di inferenza** del tipo $\frac{premesse}{conclusione}$. In assenza di premesse, $-$, la regola di inferenza si dice **assioma**.

$$\frac{-}{(skip, \sigma) \to \sigma} \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_1, \sigma)} \mathscr{B}[B]_\sigma = tt$$

$$\frac{-}{(x = E, \sigma) \to \sigma[x \mapsto n]} \mathscr{E}[E]_\sigma = n \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_2, \sigma)} \mathscr{B}[B]_\sigma = ff$$

$$\frac{(C_1, \sigma) \to (C_1', \sigma')}{(C_1; C_2, \sigma) \to (C_1'; C_2, \sigma')} \qquad \frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \to \sigma} \mathscr{B}[E_2 < E_1]_\sigma = tt$$

$$\frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \rightarrow (i = n_1;\ C;\ for\ i = n_1 + 1\ to\ n_2\ do\ C, \sigma)} \mathscr{B}[E_2 < E_1]_\sigma = ff \wedge [E_1]_\sigma = n_1 \wedge [E_2]_\sigma = n_2$$

$$\frac{-}{(while\ B\ do\ C, \sigma) \rightarrow (if\ B\ then\ C;\ while\ B\ do\ C, \sigma)}$$