Lezione di Informatica Teorica: Il Problema dell'Arresto e Introduzione agli Automi

Appunti da Trascrizione Automatica

30 giugno 2025

Indice

1 Introduzione e Riepilogo

Questa lezione mira a consolidare i concetti di base della Teoria della Computazione e introdurre gli strumenti formali per la loro analisi. In particolare, approfondiremo il concetto di problema, dimostreremo l'indecidibilità del Problema dell'Arresto e introdurremo gli Automi a Stati Finiti Determinici (DFA) come modello per lo studio dei linguaggi.

1.1 Richiamo: Definizione di Problema

Per noi, un problema è una relazione tra un input e un output. Più precisamente:

Definizione 1 (Problema)**.** *Un problema è un sottoinsieme di tutte le possibili coppie formate da una stringa di input e una stringa di output. Formalmente, un problema è una relazione $R \subseteq \Sigma^* \times \Gamma^*$, dove Σ è l'alfabeto degli input e Γ è l'alfabeto degli output.*

Quando si analizza un problema, è fondamentale chiarire tre elementi:

1. **Input**: Cosa è l'input? Come viene rappresentato? Qual è il suo significato?
2. **Output**: Cosa è l'output? Come viene rappresentato? Qual è il suo significato?
3. **Relazione**: Qual è la relazione che lega l'output all'input? Data una specifica istanza di input, quale output ci si aspetta?

Questo processo di chiarificazione è cruciale per evitare ambiguità e per una corretta formulazione del problema.

2 Il Problema dell'Arresto (Halting Problem)

2.1 Definizione Informale

Il Problema dell'Arresto chiede: "Dato un programma e un input per quel programma, il programma si arresterà (terminerà) o continuerà a girare indefinitamente su quell'input?"

2.2 Definizione Formale

Applichiamo la struttura di analisi del problema:

Definizione 2 (Problema dell'Arresto (HALT))**.** ***Input****: Una coppia $\langle P, I \rangle$, dove:*

- *P è una stringa che codifica un programma (e.g., codice Python).*
- *I è una stringa che rappresenta l'input per il programma P.*

Output*: Una risposta booleana: "sì" o "no".* ***Relazione****: L'output è "sì" se il programma P, eseguito con l'input I, si arresta. L'output è "no" se il programma P, eseguito con l'input I, non si arresta (cioè, entra in un loop infinito o non termina in tempo finito).*

Questo problema ha un'enorme utilità pratica per i programmatori, in quanto un compilatore o un ambiente di sviluppo che potesse prevedere la terminazione di un programma sarebbe uno strumento diagnostico potentissimo.

2.3 Dimostrazione dell'Indecidibilità

Dimostreremo che il Problema dell'Arresto è **indecidibile**, ovvero non esiste un algoritmo universale che possa risolvere correttamente il problema per *ogni* possibile coppia $\langle P, I \rangle$. La dimostrazione è per assurdo.

Teorema 1 (Indecidibilità del Problema dell'Arresto)**.** *Non esiste un algoritmo che, per ogni data coppia $\langle P, I \rangle$ (programma P e input I), sia in grado di determinare correttamente se P si arresta su I.*

Dimostrazione per Assurdo. Assumiamo per assurdo che esista una procedura (un algoritmo) perfetta per risolvere il Problema dell'Arresto. Chiamiamo questa procedura `halt_checker`. Questa procedura `halt_checker` prende in input due stringhe: una che rappresenta un programma P e un'altra che rappresenta un input I per P. Restituisce un valore booleano (`True` se P si arresta su I, `False` altrimenti).

```python
# Assunzione: questa funzione "magica" esiste
def halt_checker(P_code: str, I_input: str) -> bool:
    """
    Restituisce True se il programma P_code si arresta con I_input,
    False altrimenti.
    """
    pass # La sua implementazione è sconosciuta, ma si assume che funzioni perfettamente
```

Ora, useremo questa `halt_checker` per costruire un nuovo programma, che chiameremo `reverse`. Il programma `reverse` prende in input una singola stringa, che rappresenta il codice di un programma P.

```python
def reverse(P_code: str):
    """
    Questo programma prende in input il codice di un altro programma P.
    """
    # Chiama halt_checker passando P_code sia come programma che come input per sé stesso
    halts = halt_checker(P_code, P_code)

    if halts:
        # Se halt_checker ha detto che P_code si arresta su P_code,
        # allora reverse entra in un loop infinito.
        while True:
            pass # Loop infinito
    else:
        # Se halt_checker ha detto che P_code non si arresta su P_code,
        # allora reverse si arresta immediatamente.
        pass # Termina
```

Questo programma `reverse` è un programma Python legittimo. Può essere compilato ed eseguito. Sia `code_reverse` la stringa che rappresenta il codice del programma `reverse` stesso.

Consideriamo ora l'esecuzione del programma `reverse` avendo come input il proprio codice, cioè `reverse(code_reverse)`. Ci sono due possibilità per l'esecuzione di `reverse(code_reverse)`:

1. **Caso 1:** `reverse(code_reverse)` **si arresta.**
    - Se `reverse(code_reverse)` si arresta, significa che l'esecuzione è giunta alla linea `pass` (termina).
    - Questo accade se e solo se la condizione `if halts:` era falsa, cioè `halts` era `False`.
    - Il valore di `halts` è determinato dalla chiamata `halt_checker(code_reverse, code_reverse)`.
    - Quindi, `halt_checker(code_reverse, code_reverse)` deve aver restituito `False`.
    - Ma per definizione di `halt_checker`, se restituisce `False`, significa che il programma `code_reverse` (cioè `reverse`) **non si arresta** quando esegue con input `code_reverse`.
    - Abbiamo una contraddizione: abbiamo assunto che `reverse(code_reverse)` si arresta, ma la logica interna del programma e l'assunzione di `halt_checker` implicano che **non si arresta**.
2. **Caso 2:** `reverse(code_reverse)` **non si arresta (entra in loop).**
    - Se `reverse(code_reverse)` non si arresta, significa che l'esecuzione è giunta al `while True: pass` (loop infinito).
    - Questo accade se e solo se la condizione `if halts:` era vera, cioè `halts` era `True`.
    - Il valore di `halts` è determinato dalla chiamata `halt_checker(code_reverse, code_reverse)`.
    - Quindi, `halt_checker(code_reverse, code_reverse)` deve aver restituito `True`.
    - Ma per definizione di `halt_checker`, se restituisce `True`, significa che il programma `code_reverse` (cioè `reverse`) **si arresta** quando esegue con input `code_reverse`.
    - Abbiamo una contraddizione: abbiamo assunto che `reverse(code_reverse)` non si arresta, ma la logica interna del programma e l'assunzione di `halt_checker` implicano che **si arresta**.

In entrambi i casi possibili (si arresta o non si arresta), giungiamo a una contraddizione logica. Poiché la costruzione del programma `reverse` e la sua esecuzione sono perfettamente lecite secondo le regole della computazione, l'unica conclusione possibile è che l'assunzione iniziale fosse falsa.

Pertanto, la procedura `halt_checker` **non può esistere**. □

2.3.1 Implicazioni della Dimostrazione

- **Indipendenza dalla Tecnologia**: La dimostrazione non fa alcuna assunzione specifica sul linguaggio di programmazione utilizzato (Python è solo un esempio) o sull'architettura hardware su cui il programma viene eseguito. L'unico requisito è che il programma P possa essere codificato come una stringa e che possa prendere un input, e che la procedura `halt_checker` possa essere invocata. Questo rende il risultato universale: il Problema dell'Arresto è indecidibile in qualsiasi modello di computazione "abbastanza potente" da poter implementare un `halt_checker` e un `reverse` (come la Macchina di Turing, che vedremo in seguito).
- **Non Esistenza di Soluzioni Perfette**: Ciò non significa che non si possano scrivere programmi che, in alcuni casi specifici o per determinate classi di input, riescano a determinare la terminazione. Significa che non esiste un algoritmo *generale* e *perfetto* che per *tutti* gli input dia sempre la risposta corretta in un tempo finito. Possiamo avere soluzioni approssimate o euristiche, ma non una soluzione infallibile per ogni caso.

3 Classificazione dei Problemi: Decisione vs. Ricerca

Abbiamo visto che i problemi possono essere di diversa natura. È utile categorizzarli per facilitarne lo studio.

Esempio 1 (Problemi di Grafo)**.** 1. ***Problema del Percorso (PATH)****:*

- ***Input****: Un grafo G, un nodo sorgente S, un nodo destinazione T.*
- ***Output****: Un percorso da S a T in G.*

2. ***Problema del Ciclo Hamiltoniano (HAMILTONIAN CYCLE)****:*

- ***Input****: Un grafo G.*
- ***Output****: "Sì" se esiste un ciclo Hamiltoniano (un ciclo che visita ogni nodo esattamente una volta) in G, "No" altrimenti.*

Notiamo una differenza fondamentale nell'output:

Definizione 3 (Problema di Ricerca)**.** *Un problema di ricerca è un problema il cui output può essere una stringa arbitraria, come un percorso, un numero, una derivata, una matrice risultante, ecc. L'obiettivo è trovare o calcolare qualcosa.*

Definizione 4 (Problema di Decisione)**.** *Un problema di decisione è un problema il cui output è sempre una risposta booleana: "sì" o "no". L'obiettivo è decidere se una certa proprietà è vera o falsa per l'input dato.*

3.0.1 Relazione tra Problemi di Ricerca e Problemi di Decisione

Queste due classi non sono disgiunte. Per ogni problema di ricerca, possiamo definire un problema di decisione correlato.

Esempio 2 (PATH come Problema di Decisione)**.** *Consideriamo il problema del percorso come problema di decisione:*

- ***Input****: Un grafo G, un nodo sorgente S, un nodo destinazione T.*
- ***Output****: "Sì" se esiste un percorso da S a T in G, "No" altrimenti.*

È evidente che se siamo in grado di risolvere il problema di ricerca (trovare un percorso), siamo automaticamente in grado di risolvere il problema di decisione (sapere se un percorso esiste). Spesso, risolvere il problema di decisione è concettualmente più semplice, poiché richiede solo una risposta binaria. Per questa ragione, nella teoria della computazione, ci si concentra principalmente sui problemi di decisione.

4 Problemi di Decisione e Linguaggi Formali

Sebbene i problemi di decisione siano più semplici dei problemi di ricerca, possono comunque avere input di forme molto diverse (grafi, matrici, numeri, ecc.). Per standardizzare lo studio e semplificare l'analisi degli algoritmi, si ricorre al concetto di **linguaggio formale**.

4.1 Definizioni Fondamentali

Definizione 5 (Alfabeto (Σ))**.** *Un alfabeto Σ è un insieme finito e non vuoto di simboli.*

Esempio 3. $\Sigma = \{a, b, c\}$
$\Sigma = \{0, 1\}$
$\Sigma = \{quadratino, cerchietto\}$

Definizione 6 (Parola o Stringa)**.** *Una parola (o stringa) su un alfabeto Σ è una sequenza finita di zero o più simboli tratti da Σ.*

Esempio 4. *Se $\Sigma = \{a, b\}$, allora ababa, a, b, ϵ (stringa vuota) sono parole su Σ.*

Definizione 7 (Σ^*)**.** *Σ^* denota l'insieme di tutte le possibili parole (di qualsiasi lunghezza, inclusa la stringa vuota ϵ) che possono essere formate usando i simboli dell'alfabeto Σ.*

Definizione 8 (Linguaggio (L))**.** *Un linguaggio L su un alfabeto Σ è un qualsiasi sottoinsieme di Σ^*. Formalmente, $L \subseteq \Sigma^*$.*

Esempio 5. *Se $\Sigma = \{0, 1\}$, il linguaggio di tutte le stringhe binarie che iniziano con '0' potrebbe essere $L = \{0, 00, 01, 000, 001, \ldots\}$.*

4.2 Decidere un Linguaggio

Definizione 9 (Problema di Decisione di un Linguaggio)**.** *Decidere un linguaggio L significa risolvere il seguente problema: **Input**: Una stringa $w \in \Sigma^*$. **Output**: "Sì" o "No". **Relazione**: L'output è "Sì" se $w \in L$. L'output è "No" se $w \notin L$.*

È importante notare che il linguaggio L fa parte della definizione del problema, non dell'input. L'input è solo la stringa w.

4.3 Conversione di Problemi di Decisione in Problemi di Linguaggio

Ogni problema di decisione può essere ricondotto (o codificato) come un problema di decisione di un linguaggio. Questo si fa codificando le istanze del problema come stringhe.

Esempio 6 (Codifica del Problema PATH come Linguaggio)**.** *Riprendiamo il problema di decisione PATH: "Esiste un percorso da S a T in G?". Possiamo definire un linguaggio L_{PATH} su un opportuno alfabeto Σ (ad esempio, simboli per nodi, archi, parentesi, virgole, ecc.) tale che:*

$$L_{PATH} = \{\langle G, S, T\rangle \mid G \text{ è un grafo, } S \text{ e } T \text{ sono nodi in } G, \text{ ed esiste un percorso da } S \text{ a } T \text{ in } G\}.$$

Dove $\langle G, S, T\rangle$ è una stringa che codifica la tripla (G, S, T). Decidere se una stringa w appartiene a L_{PATH} è equivalente a risolvere il problema di decisione PATH originale.

Questa riduzione ci permette di studiare tutti i problemi di decisione concentrandoci unicamente sulla classe dei problemi di riconoscere linguaggi. Questo è il passo che ci permette di definire modelli di calcolo astratti (gli automi) che non dipendono da specifici linguaggi di programmazione o architetture hardware.

5 Introduzione agli Automi

5.1 Concetto Intuitivo di Automa

Un automa è un dispositivo astratto (o una macchina) che ha un numero finito di "stati" o "modalità di funzionamento". L'automa riceve segnali o input e, in base al suo stato corrente e al segnale ricevuto, cambia stato e/o produce un output.

5.2 Esempio: Automa di un Lettore CD

Consideriamo il comportamento di un lettore CD come un automa per capire il concetto.

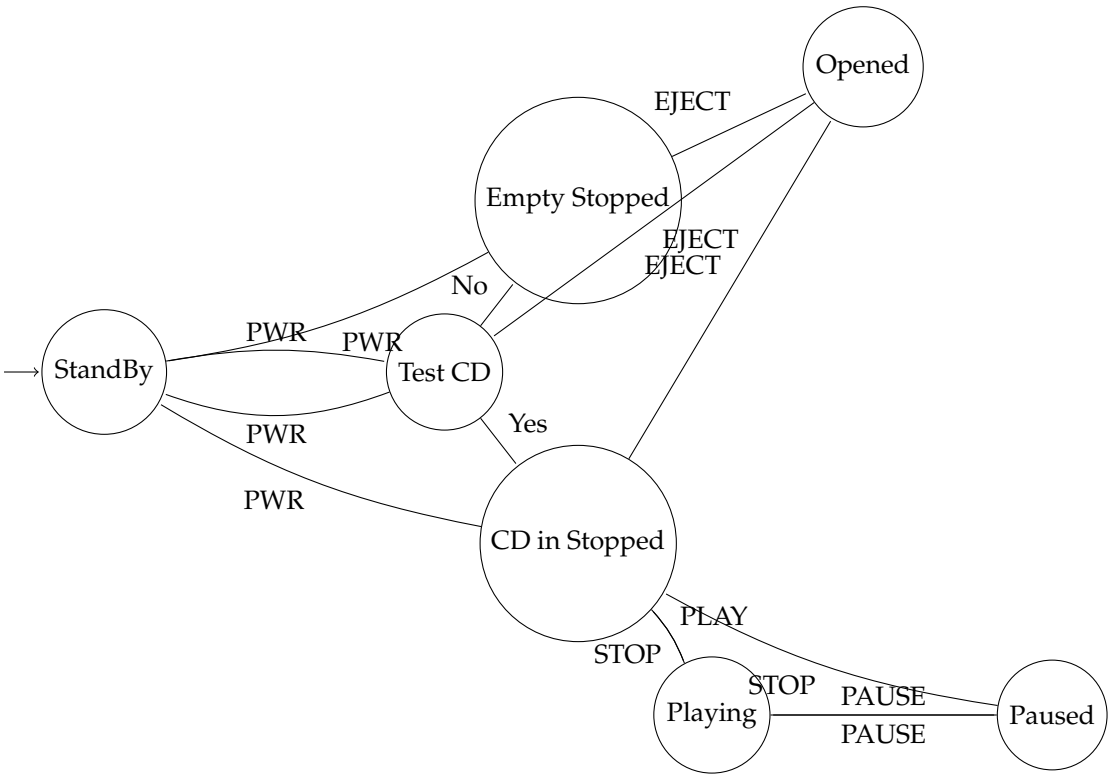

Figura 1: Automa di un Lettore CD

5.2.1 Spiegazione dell'Automa del Lettore CD

- **Stati (nodi)**: Rappresentano le diverse modalità operative del lettore (e.g., `StandBy`, `Playing`, `Paused`). Il pallino pieno indica lo stato iniziale (`StandBy`).
- **Transizioni (frecce)**: Rappresentano il passaggio da uno stato all'altro in risposta a un input (e.g., `PWR`, `EJECT`, `PLAY`). Le etichette sulle frecce indicano l'input che provoca la transizione.
- **Input/Segnali**: Azioni dell'utente (pressione di tasti) o sensori (rilevamento CD).

Questo esempio illustra come un automa modellizzi un sistema: lo stato in cui si trova il sistema influenzerà come reagirà al prossimo input.

5.3 Automi a Stati Finiti Determinici (DFA)

Gli automi vengono usati in informatica teorica per riconoscere linguaggi formali. Un Automaton a Stati Finiti Determinico (DFA) è il modello più semplice.

Definizione 10 (Automa a Stati Finiti Determinico (DFA))**.** *Un DFA M è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove:*

- *Q è un insieme finito di stati.*
- *Σ è l'alfabeto di input.*
- *$\delta : Q \times \Sigma \to Q$ è la funzione di transizione, che per ogni stato e ogni simbolo di input determina in modo unico il prossimo stato.*
- *$q_0 \in Q$ è lo stato iniziale.*
- *$F \subseteq Q$ è l'insieme degli stati finali (o accettanti).*

Un DFA **accetta** una stringa se, partendo dallo stato iniziale e processando la stringa simbolo per simbolo, l'automa termina in uno stato accettante. Altrimenti, la stringa viene **rifiutata**.

Esempio 7 (Linguaggio dei Numeri Binari Dispari)**.** *Consideriamo il linguaggio $L_{dispari}$ delle stringhe binarie che rappresentano numeri dispari.*

- $\Sigma = \{0, 1\}$
- *Una stringa binaria rappresenta un numero dispari se e solo se il suo ultimo simbolo è '1'. (Es: $1_2 = 1$, $11_2 = 3$, $101_2 = 5$; $0_2 = 0$, $10_2 = 2$, $100_2 = 4$). La stringa vuota ϵ non rappresenta un numero e non deve essere accettata.*

Costruiamo un DFA per $L_{dispari}$:

- $Q = \{Q_0, Q_1\}$
  - *Q_0: Rappresenta lo stato in cui la stringa letta finora indica un numero pari (o la stringa vuota, o un numero che termina con 0).*
  - *Q_1: Rappresenta lo stato in cui la stringa letta finora indica un numero dispari (termina con 1).*
- *$q_0 = Q_0$ (Inizialmente, il numero è considerato pari, o non abbiamo ancora letto nulla).*
- *$F = \{Q_1\}$ (Se la stringa termina mentre siamo in questo stato, è un numero dispari).*
- *Funzione di transizione δ:*
  - *$\delta(Q_0, 0) = Q_0$ (Se eravamo in stato 'pari' e leggiamo '0', rimaniamo 'pari').*
  - *$\delta(Q_0, 1) = Q_1$ (Se eravamo in stato 'pari' e leggiamo '1', diventiamo 'dispari').*
  - *$\delta(Q_1, 0) = Q_0$ (Se eravamo in stato 'dispari' e leggiamo '0', diventiamo 'pari').*
  - *$\delta(Q_1, 1) = Q_1$ (Se eravamo in stato 'dispari' e leggiamo '1', rimaniamo 'dispari').*

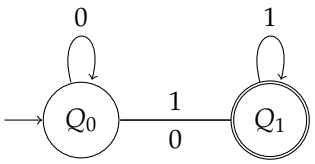

Figura 2: DFA per il Linguaggio dei Numeri Binari Dispari

5.3.1 Esempio di Tracciamento

Stringa di input: `101`

1. *Inizia in* Q_0.
2. *Legge '1':* $\delta(Q_0, 1) = Q_1$. *Automa si sposta in* Q_1.
3. *Legge '0':* $\delta(Q_1, 0) = Q_0$. *Automa si sposta in* Q_0.
4. *Legge '1':* $\delta(Q_0, 1) = Q_1$. *Automa si sposta in* Q_1.
5. *Stringa terminata. L'automa si trova in* Q_1, *che è uno stato accettante.*

Quindi, la stringa `101` *(che rappresenta il numero 5, dispari) è accettata.*
Stringa di input: `110`

1. *Inizia in* Q_0.
2. *Legge '1':* $\delta(Q_0, 1) = Q_1$. *Automa si sposta in* Q_1.
3. *Legge '1':* $\delta(Q_1, 1) = Q_1$. *Automa si sposta in* Q_1.
4. *Legge '0':* $\delta(Q_1, 0) = Q_0$. *Automa si sposta in* Q_0.
5. *Stringa terminata. L'automa si trova in* Q_0, *che non è uno stato accettante.*

Quindi, la stringa `110` *(che rappresenta il numero 6, pari) è rifiutata.*

Questo modello dimostra come un DFA possa "decidere" se una data stringa appartiene a un linguaggio specifico. I DFA sono un modello di calcolo molto semplice ma potente per la loro classe di problemi.