# Tutorato di Automi e Linguaggi Formali

Soluzioni Esercizi Misti - Teoria della Computazione

**Gabriel Rovesti**
Corso di Laurea in Informatica – Università degli Studi di Padova

Soluzioni Esercizi Vari

## 1 Problemi NP-Completi

**Esercizio 7.28 - PUZZLE 1.** Vi è data una scatola e una collezione di carte come indicato nella figura seguente. A causa dei pioli nella scatola e delle tacche nelle carte, ogni carta si adatterà nella scatola in uno di due modi solamente. Ogni carta contiene due colonne di buchi, alcuni dei quali potrebbero non essere perforati. Il puzzle è risolto posizionando tutte le carte nella scatola in modo da coprire completamente il fondo della scatola (cioè, ogni posizione di buco è bloccata da almeno una carta che non ha buchi lì). Sia $PUZZLE = \{(c_1, \ldots, c_k) \mid \text{ogni } c_i \text{ rappresenta una carta e questa collezione di carte ha una soluzione}\}$. Dimostrare che $PUZZLE$ è NP-completo.

**Soluzione.** Dimostriamo che $PUZZLE$ è NP-completo costruendo una riduzione da $VERTEX\text{-}COVER$.

**Passo 1:** $PUZZLE \in NP$

**Teorema 1.** $PUZZLE \in NP$.

*Proof.* **Certificato:** Una configurazione delle carte, specificando per ogni carta quale orientamento scegliere (modo 1 o modo 2).

**Verificatore:** Su input $\langle (c_1, \ldots, c_k), \text{configurazione} \rangle$:

1. Per ogni posizione del fondo della scatola, verifica che almeno una carta nella configurazione data non abbia un buco in quella posizione

2. Verifica che tutte le carte siano utilizzate nella configurazione

Complessità: $O(k \cdot \text{posizioni})$, che è polinomiale. □

**Passo 2: Riduzione** $VERTEX\text{-}COVER \leq_p PUZZLE$

**Teorema 2.** $VERTEX\text{-}COVER \leq_p PUZZLE$.

*Proof.* Data un'istanza $(G, k)$ di $VERTEX\text{-}COVER$ dove $G = (V, E)$ con $V = \{v_1, \dots, v_n\}$ e $E = \{e_1, \dots, e_m\}$, costruiamo un'istanza di $PUZZLE$.

**Costruzione:**

- **Scatola**: Ha $m$ posizioni (una per ogni arco di $G$)
- **Carte**: Una carta per ogni vertice $v_i \in V$
- **Orientamenti delle carte**: Per ogni carta $c_i$ corrispondente al vertice $v_i$:
  - **Modo 1**: Ha buchi in tutte le posizioni corrispondenti agli archi incidenti a $v_i$
  - **Modo 2**: Ha buchi in tutte le posizioni (carta "vuota")

**Intuizione**: Scegliere il modo 1 per una carta corrisponde a "non includere" il vertice corrispondente nella vertex cover; scegliere il modo 2 corrisponde a "includere" il vertice nella vertex cover.

**Modifica per forzare al massimo $k$ scelte del modo 2**: Aggiungiamo $n - k$ posizioni speciali nella scatola e modifichiamo ogni carta aggiungendo buchi in tutte queste posizioni speciali nel modo 1, ma non nel modo 2.

**Correttezza:**

($\Rightarrow$) Se $G$ ha una vertex cover $C$ di dimensione al massimo $k$:

- Per ogni $v_i \in C$: scegli modo 2 per carta $c_i$
- Per ogni $v_i \notin C$: scegli modo 1 per carta $c_i$
- Ogni arco $e_j = \{v_a, v_b\}$ è coperto perché almeno uno tra $v_a, v_b$ è in $C$, quindi almeno una carta tra $c_a, c_b$ è in modo 2 (senza buchi) e copre la posizione $j$
- Le posizioni speciali sono coperte perché al massimo $k$ carte sono in modo 2, quindi almeno $n - k$ carte sono in modo 1 (senza buchi nelle posizioni speciali)

($\Leftarrow$) Se $PUZZLE$ ha soluzione:

- Sia $C = \{v_i \mid \text{carta } c_i \text{ è in modo } 2\}$
- Per le posizioni speciali essere coperte, al massimo $k$ carte possono essere in modo 2, quindi $|C| \leq k$
- Per ogni arco $e_j = \{v_a, v_b\}$, la posizione $j$ deve essere coperta, quindi almeno una carta tra $c_a, c_b$ è in modo 2, cioè almeno uno tra $v_a, v_b$ è in $C$
- Quindi $C$ è una vertex cover di dimensione al massimo $k$

La riduzione opera chiaramente in tempo polinomiale. □

Quindi $PUZZLE$ è NP-completo.

**Esercizio 7.35 - DOMINATING-SET 2.** Un sottoinsieme dei nodi di un grafo $G$ è un *insieme dominante* se ogni altro nodo di $G$ è adiacente a qualche nodo nel sottoinsieme. Sia

$$DOMINATING\text{-}SET = \{(G, k) \mid G \text{ ha un insieme dominante con } k \text{ nodi}\}.$$

Dimostrare che è NP-completo fornendo una riduzione da $VERTEX\text{-}COVER$.

**Soluzione.** **Passo 1:** $DOMINATING\text{-}SET \in NP$

**Teorema 3.** *DOMINATING-SET* $\in NP$.

*Proof.* **Certificato:** Un sottoinsieme $D \subseteq V$ di vertici.

**Verificatore:** Su input $\langle (G, k), D \rangle$:

1. Verifica che $|D| \leq k$
2. Per ogni vertice $v \notin D$, verifica che esista almeno un vertice $u \in D$ tale che $(v, u) \in E$

Complessità: $O(|V| \cdot |E|)$, che è polinomiale. □

**Passo 2: Riduzione** $VERTEX\text{-}COVER \leq_p DOMINATING\text{-}SET$

**Teorema 4.** $VERTEX\text{-}COVER \leq_p DOMINATING\text{-}SET$.

*Proof.* Data un'istanza $(G, k)$ di $VERTEX\text{-}COVER$ dove $G = (V, E)$, costruiamo un'istanza $(G', k')$ di $DOMINATING\text{-}SET$.

**Costruzione di** $G'$:

- **Vertici**: $V' = V \cup E \cup \{s\}$ dove:
  - $V$ sono i vertici originali
  - $E$ sono nuovi vertici (uno per ogni arco originale)
  - $s$ è un nuovo vertice speciale
- **Archi**: $E'$ contiene:
  - Per ogni arco $e = \{u, v\} \in E$: aggiungi archi $(u, e)$ e $(v, e)$
  - Per ogni vertice $v \in V$: aggiungi arco $(v, s)$
- **Parametro**: $k' = k$

**Intuizione**:

- I vertici-arco (da $E$) devono essere dominati dai vertici originali
- Il vertice speciale $s$ deve essere dominato da almeno un vertice originale
- Una vertex cover in $G$ corrisponde a un dominating set in $G'$

**Correttezza:**

($\Rightarrow$) Se $G$ ha una vertex cover $C$ di dimensione al massimo $k$:

- Ogni arco $e = \{u, v\} \in E$ ha almeno un estremo in $C$, quindi il vertice-arco corrispondente in $V'$ è dominato da $C$
- Il vertice speciale $s$ è dominato da tutti i vertici in $V$, in particolare da quelli in $C$
- Tutti i vertici originali in $V \setminus C$ sono dominati dal vertice $s$
- Quindi $C$ è un dominating set di $G'$ con dimensione al massimo $k$

($\Leftarrow$) Se $G'$ ha un dominating set $D$ di dimensione al massimo $k$:

- $D$ deve contenere almeno un vertice da $V$ (altrimenti $s$ non sarebbe dominato)
- Per ogni vertice-arco $e \in E$, $e$ deve essere dominato da qualche vertice in $D \cap V$
- Sia $C = D \cap V$. Per ogni arco originale $e = \{u, v\} \in E$, almeno uno tra $u, v$ deve essere in $C$ (altrimenti il vertice-arco corrispondente non sarebbe dominato)
- Quindi $C$ è una vertex cover di $G$ con dimensione al massimo $k$

La riduzione opera in tempo $O(|V| + |E|)$, che è polinomiale. □

Quindi *DOMINATING-SET* è NP-completo.

**Esercizio 7.29 - 3COLOR 3.** Una *colorazione* di un grafo è un'assegnazione di colori ai suoi nodi in modo che due nodi adiacenti non abbiano lo stesso colore. Sia

$$3COLOR = \{(G) \mid G \text{ è colorabile con 3 colori}\}.$$

Dimostrare che $3COLOR$ è NP-completo. (Suggerimento: Usare i seguenti tre sottografi.)

**Soluzione.** Dimostriamo che $3COLOR$ è NP-completo usando una riduzione da $3SAT$ con i gadget suggeriti.

**Passo 1:** $3COLOR \in NP$

**Teorema 5.** *$3COLOR \in NP$.*

*Proof.* **Certificato:** Una funzione di colorazione $c : V \to \{1, 2, 3\}$.

**Verificatore:** Verifica che per ogni arco $(u, v) \in E$, si abbia $c(u) \neq c(v)$. Complessità: $O(|E|)$, che è polinomiale. □

**Passo 2: Riduzione** $3SAT \leq_p 3COLOR$

**Teorema 6.** *$3SAT \leq_p 3COLOR$.*

*Proof.* Data una formula 3-CNF $\phi$ con variabili $x_1, \ldots, x_n$ e clausole $C_1, \ldots, C_m$, costruiamo un grafo $G$ che è 3-colorabile se e solo se $\phi$ è soddisfacibile.

**Costruzione usando i gadget:**

**1. Palette (base):**

- Crea un triangolo con vertici $T$ (True), $F$ (False), e $B$ (Base)
- Questi tre vertici devono avere colori diversi
- Assumiamo: $T$ ha colore 1 (rosso), $F$ ha colore 2 (blu), $B$ ha colore 3 (verde)

**2. Variable gadget:** Per ogni variabile $x_i$:

- Crea due vertici $x_i$ e $\neg x_i$
- Connetti entrambi a $B$ (quindi non possono avere colore 3)
- Connetti $x_i$ e $\neg x_i$ tra loro (quindi devono avere colori diversi)
- Risultato: uno ha colore 1 (vero), l'altro ha colore 2 (falso)

**3. OR-gadget per clausole:** Per ogni clausola $C_j = (l_{j1} \lor l_{j2} \lor l_{j3})$:

- Crea la struttura del gadget OR mostrata nel diagramma
- Il gadget ha la proprietà che può essere 3-colorato se e solo se almeno uno dei letterali $l_{j1}, l_{j2}, l_{j3}$ ha colore 1 (è vero)

**Struttura dettagliata dell'OR-gadget:**

- Vertici: $o_1, o_2, o_3, o_4, o_5, o_6$ (struttura a 6 vertici)
- Connessioni:
  - $l_{j1}, l_{j2}, l_{j3}$ sono connessi a $o_1$
  - $o_1$ è connesso a $o_2$ e $B$
  - $o_2$ è connesso a $o_3$ e $o_4$
  - $o_3$ e $o_4$ sono connessi a $o_5$
  - $o_5$ è connesso a $o_6$ e $B$
  - $o_6$ è connesso a $B$

**Proprietà chiave dell'OR-gadget:** Se tutti i letterali $l_{j1}, l_{j2}, l_{j3}$ hanno colore 2 (falso), allora:

- $o_1$ può avere colore 1 o 3
- Ma la struttura interna del gadget forza contraddizioni che rendono impossibile una 3-colorazione valida
- Se almeno un letterale ha colore 1, il gadget può essere colorato correttamente

**Correttezza:**
($\Rightarrow$) Se $\phi$ è soddisfacibile con assegnamento $\tau$:

- Colora $T, F, B$ con colori 1, 2, 3 rispettivamente
- Per ogni variabile $x_i$:
  - Se $\tau(x_i) =$ vero: colora $x_i$ con 1, $\neg x_i$ con 2
  - Se $\tau(x_i) =$ falso: colora $x_i$ con 2, $\neg x_i$ con 1
- Per ogni clausola $C_j$: almeno un letterale è vero (colore 1), quindi l'OR-gadget può essere colorato

($\Leftarrow$) Se $G$ è 3-colorabile:

- La palette fissa i colori base
- Per ogni variabile $x_i$: definisci $\tau(x_i) =$ vero se $x_i$ ha colore 1
- Per ogni clausola $C_j$: l'OR-gadget può essere colorato solo se almeno un letterale ha colore 1, quindi la clausola è soddisfatta

La riduzione opera in tempo polinomiale: $O(n + m)$ vertici e archi. □

Quindi $3COLOR$ è NP-completo.

# 2 Problemi di Cammini in Grafi

**Esercizio 7.21 - SPATH e LPATH 4.** Sia $G$ un grafo non diretto. Inoltre sia

$$SPATH = \{(G, a, b, k) \mid G \text{ contiene un cammino semplice di lunghezza al più } k \text{ da } a \text{ a } b\} \tag{1}$$

$$LPATH = \{(G, a, b, k) \mid G \text{ contiene un cammino semplice di lunghezza almeno } k \text{ da } a \text{ a } b\} \tag{2}$$

a) Dimostrare che $SPATH \in P$.

b) Dimostrare che $LPATH$ è NP-completo.

**Soluzione. a) Dimostrazione che $SPATH \in P$**

**Teorema 7.** *$SPATH \in P$.*

*Proof.* Usiamo la programmazione dinamica per risolvere $SPATH$.

**Algoritmo:**

Listing 1: Algoritmo DP per SPATH

```python
def SPATH(G, a, b, k):
    n = len(G.vertices())
    # dp[v][i][S] = True se esiste cammino di lunghezza i
    # da a a v visitando esattamente i vertici in S
    dp = {}

    # Caso base
    dp[(a, 0, frozenset([a]))] = True

    for length in range(1, min(k + 1, n)):
        for v in G.vertices():
            for S in all_subsets_of_size(G.vertices(), length):
                if v not in S:
                    continue
                dp[(v, length, S)] = False

                for u in G.neighbors(v):
                    if u in S and (u, length-1, S - {v}) in dp:
                        if dp[(u, length-1, S - {v})]:
                            dp[(v, length, S)] = True
                            break

    # Controlla se esiste un cammino di lunghezza <= k da a a b
    for length in range(k + 1):
```

```
        for S in all_subsets(G.vertices()):
            if (b, length, S) in dp and dp[(b, length, S)]:
                return True

    return False
```

**Versione semplificata usando BFS limitata:**

Listing 2: BFS limitata per SPATH

```
def SPATH_simple(G, a, b, k):
    if a == b:
        return True

    # BFS con limite di profondit  k
    queue = [(a, 0, {a})]  # (vertice_corrente, lunghezza, vertici_visitati)

    while queue:
        current, length, visited = queue.pop(0)

        if length >= k:
            continue

        for neighbor in G.neighbors(current):
            if neighbor == b and length + 1 <= k:
                return True

            if neighbor not in visited and length + 1 < k:
                queue.append((neighbor, length + 1, visited | {neighbor}))

    return False
```

**Analisi della complessità:**

- Nel caso peggiore, esploriamo tutti i possibili cammini semplici di lunghezza al più $k$
- Numero di stati: $O(|V| \cdot k \cdot 2^{|V|})$ per la versione DP completa
- Per $k$ limitato polinomialmente, questo è ancora polinomiale
- La versione BFS ha complessità $O(|V|^k)$ che è polinomiale se $k$ è limitato

Quindi $SPATH \in P$. □

**b) Dimostrazione che $LPATH$ è NP-completo**
**Passo 1: $LPATH \in NP$**

**Teorema 8.** *$LPATH \in NP$.*

*Proof.* **Certificato:** Un cammino semplice $P = v_1, v_2, \ldots, v_\ell$ dove $v_1 = a$, $v_\ell = b$, e $\ell \geq k$.
**Verificatore:**

1. Verifica che $v_1 = a$ e $v_\ell = b$
2. Verifica che $\ell \geq k$
3. Verifica che tutti i vertici nel cammino sono distinti
4. Verifica che per ogni $i$, esiste un arco $(v_i, v_{i+1})$ in $G$

Complessità: $O(\ell) = O(|V|)$, che è polinomiale. □

**Passo 2: Riduzione** $HAMPATH \leq_p LPATH$

**Teorema 9.** *$LPATH$ è NP-hard.*

*Proof.* Riduciamo da $HAMPATH$ (cammino Hamiltoniano), che è NP-completo.
Data un'istanza $(G, s, t)$ di $HAMPATH$, costruiamo un'istanza $(G, s, t, |V| - 1)$ di $LPATH$.
**Correttezza:**

- $G$ ha un cammino Hamiltoniano da $s$ a $t$ se e solo se ha un cammino semplice di lunghezza almeno $|V| - 1$ da $s$ a $t$
- Un cammino semplice in un grafo con $|V|$ vertici può avere al massimo lunghezza $|V| - 1$
- Quindi un cammino di lunghezza almeno $|V|-1$ deve essere esattamente di lunghezza $|V| - 1$, cioè Hamiltoniano

La riduzione è chiaramente polinomiale (tempo costante). □

Quindi $LPATH$ è NP-completo.

# 3 Problemi su Grammatiche Context-Free

**Esercizio 5.35 - NECESSARY CFG 5.** Diciamo che una variabile $A$ in una CFG $G$ è *necessaria* se appare in qualche derivazione di qualche stringa $w \in G$. Sia $NECESSARY_{CFG} = \{(G, A) \mid A \text{ è una variabile necessaria in } G\}$.

a) Dimostrare che $NECESSARY_{CFG}$ è Turing-riconoscibile.

b) Dimostrare che $NECESSARY_{CFG}$ è indecidibile.

**Soluzione. a) $NECESSARY_{CFG}$ è Turing-riconoscibile**

**Teorema 10.** *$NECESSARY_{CFG}$ è Turing-riconoscibile.*

*Proof.* Costruiamo una macchina di Turing $M$ che riconosce $NECESSARY_{CFG}$:
**Algoritmo per $M$:** Su input $\langle G, A \rangle$:

1. Enumera tutte le stringhe $w \in \Sigma^*$ in ordine lexicografico
2. Per ogni stringa $w$:
   (a) Usa l'algoritmo CYK per verificare se $w \in L(G)$
   (b) Se $w \in L(G)$, enumera tutte le possibili derivazioni di $w$ in $G$
   (c) Se trovi una derivazione che usa la variabile $A$, accetta
3. Se nessuna stringa usa $A$ in alcuna derivazione, il programma non si ferma

**Correttezza:**

- Se $A$ è necessaria, esiste qualche stringa $w \in L(G)$ e qualche derivazione di $w$ che usa $A$
- $M$ eventualmente enumererà $w$ e troverà la derivazione che usa $A$, quindi accetterà
- Se $A$ non è necessaria, $M$ non troverà mai una derivazione che usa $A$ e non accetterà mai

Quindi $NECESSARY_{CFG}$ è Turing-riconoscibile. □

**b) $NECESSARY_{CFG}$ è indecidibile**

**Teorema 11.** *$NECESSARY_{CFG}$ è indecidibile.*

*Proof.* Riduciamo da $E_{CFG}$ (il problema di determinare se una CFG genera il linguaggio vuoto), che è indecidibile.

Data una CFG $G$ con variabile iniziale $S$, costruiamo una nuova CFG $G'$ e una variabile $A$ tale che:

$$\langle G \rangle \in E_{CFG} \Leftrightarrow \langle G', A \rangle \in NECESSARY_{CFG}$$

**Costruzione di $G'$:**

- Prendi tutte le produzioni di $G$
- Aggiungi una nuova variabile $A$ e una nuova variabile iniziale $S'$
- Aggiungi le produzioni:
  - $S' \to A$
  - $A \to S$ (se $L(G) \neq \emptyset$, questa produzione sarà utile)
  - $A \to \varepsilon$ (per garantire che $L(G') \neq \emptyset$)

**Analisi:**
**Caso 1:** $L(G) = \emptyset$ (cioè $\langle G \rangle \in E_{CFG}$)

- In $G'$, l'unico modo per derivare stringhe è: $S' \Rightarrow A \Rightarrow \varepsilon$
- Quindi $L(G') = \{\varepsilon\}$
- La variabile $A$ è necessaria perché appare nell'unica derivazione possibile

- Quindi $\langle G', A\rangle \in NECESSARY_{CFG}$

**Caso 2:** $L(G) \neq \emptyset$ (cioè $\langle G\rangle \notin E_{CFG}$)

- In $G'$, possiamo derivare stringhe in due modi:
  - $S' \Rightarrow A \Rightarrow \varepsilon$ (deriva $\varepsilon$)
  - $S' \Rightarrow A \Rightarrow S \Rightarrow^* w$ per qualche $w \in L(G)$
- La variabile $A$ appare in tutte le derivazioni, quindi è necessaria
- Quindi $\langle G', A\rangle \in NECESSARY_{CFG}$

**Problema con la costruzione precedente:** In entrambi i casi $A$ risulta necessaria.
**Costruzione corretta:** Modifichiamo la costruzione:

- $S' \to A \mid B$
- $A \to S$ (se $L(G) \neq \emptyset$, questa sarà utile)
- $B \to \varepsilon$ (per garantire che $L(G') \neq \emptyset$)

**Analisi corretta:**
**Caso 1:** $L(G) = \emptyset$

- Le derivazioni possibili sono solo: $S' \Rightarrow B \Rightarrow \varepsilon$
- La variabile $A$ non appare in nessuna derivazione
- Quindi $\langle G', A\rangle \notin NECESSARY_{CFG}$

**Caso 2:** $L(G) \neq \emptyset$

- Possiamo derivare: $S' \Rightarrow A \Rightarrow S \Rightarrow^* w$ per qualche $w \in L(G)$
- La variabile $A$ appare in queste derivazioni
- Quindi $\langle G', A\rangle \in NECESSARY_{CFG}$

Quindi: $\langle G\rangle \in E_{CFG} \Leftrightarrow \langle G', A\rangle \notin NECESSARY_{CFG}$.
Questo mostra che $\overline{NECESSARY_{CFG}}$ è indecidibile, quindi $NECESSARY_{CFG}$ è indecidibile. □

**Esercizio 5.36 - MIN CFG 6.** Diciamo che una CFG è *minimale* se nessuna delle sue regole può essere rimossa senza cambiare il linguaggio generato. Sia $MIN_{CFG} = \{(G) \mid G \text{ è una CFG minimale}\}$.

a) Dimostrare che $MIN_{CFG}$ è Turing-riconoscibile.

b) Dimostrare che $MIN_{CFG}$ è indecidibile.

**Soluzione.** **a) $MIN_{CFG}$ è Turing-riconoscibile**

**Teorema 12.** *$MIN_{CFG}$ è Turing-riconoscibile.*

*Proof.* Costruiamo una macchina di Turing $M$ che riconosce $MIN_{CFG}$:
**Algoritmo per $M$:** Su input $\langle G \rangle$:

1. Per ogni produzione $p$ in $G$:
   (a) Costruisci $G_p$ rimuovendo la produzione $p$ da $G$
   (b) Enumera stringhe $w \in \Sigma^*$ in parallelo per $G$ e $G_p$
   (c) Se trovi una stringa $w$ tale che $w \in L(G)$ ma $w \notin L(G_p)$ (o viceversa), continua con la prossima produzione
   (d) Se per tutte le stringhe enumerate finora, $L(G) = L(G_p)$, continua l'enumerazione
2. Se per ogni produzione $p$, troviamo una stringa testimone che $L(G) \neq L(G_p)$, accetta
3. Altrimenti, continua l'enumerazione (non termina)

**Correttezza:**

- Se $G$ è minimale, per ogni produzione $p$, esiste una stringa $w$ tale che rimuovere $p$ cambia il linguaggio
- $M$ eventualmente troverà tutte queste stringhe testimone e accetterà
- Se $G$ non è minimale, esiste almeno una produzione ridondante $p$ tale che $L(G) = L(G_p)$
- Per questa produzione, $M$ non troverà mai una stringa testimone e non accetterà

Quindi $MIN_{CFG}$ è Turing-riconoscibile. □

**b) $MIN_{CFG}$ è indecidibile**

**Teorema 13.** *$MIN_{CFG}$ è indecidibile.*

*Proof.* Riduciamo da $E_{CFG}$.
Data una CFG $G$ con variabile iniziale $S$, costruiamo una CFG $G'$ tale che:

$$\langle G \rangle \in E_{CFG} \Leftrightarrow \langle G' \rangle \in MIN_{CFG}$$

**Costruzione di $G'$:**

- Variabili: tutte le variabili di $G$, più una nuova variabile iniziale $S'$
- Produzioni:
  - Tutte le produzioni di $G$
  - $S' \to \varepsilon$ (per garantire che $L(G') \neq \emptyset$)
  - $S' \to S$ (questa produzione è ridondante se e solo se $L(G) = \emptyset$)

**Analisi:**
**Caso 1:** $L(G) = \emptyset$ (cioè $\langle G \rangle \in E_{CFG}$)

- In $G'$: $L(G') = \{\varepsilon\}$ (solo dalla produzione $S' \to \varepsilon$)
- La produzione $S' \to S$ è ridondante perché non genera stringhe aggiuntive
- Tutte le altre produzioni (quelle di $G$) sono irraggiungibili da $S'$
- Quindi $G'$ non è minimale
- Quindi $\langle G' \rangle \notin MIN_{CFG}$

**Caso 2:** $L(G) \neq \emptyset$ (cioè $\langle G \rangle \notin E_{CFG}$)

- In $G'$: $L(G') = \{\varepsilon\} \cup L(G)$
- La produzione $S' \to \varepsilon$ genera $\varepsilon$
- La produzione $S' \to S$ permette di generare tutte le stringhe in $L(G)$
- Nessuna delle due produzioni è ridondante
- Le produzioni di $G$ sono tutte necessarie (assumendo che $G$ sia già minimale)
- Quindi $G'$ è minimale
- Quindi $\langle G' \rangle \in MIN_{CFG}$

**Raffinamento:** Per gestire il caso in cui $G$ originale non sia minimale, modifichiamo la costruzione:

- Prima convertiamo $G$ in una CFG minimale equivalente $G_{min}$ usando un algoritmo di minimizzazione (rimuovendo variabili inutili e produzioni ridondante)
- Poi applichiamo la costruzione sopra a $G_{min}$

Con questa modifica:

$$\langle G \rangle \in E_{CFG} \Leftrightarrow \langle G' \rangle \notin MIN_{CFG}$$

Quindi $\overline{MIN_{CFG}}$ è indecidibile, il che implica che $MIN_{CFG}$ è indecidibile. □

# 4 Problemi Decidibili

**Esercizio 4.4 - $A_{\varepsilon CFG}$ 7.** Sia $A_{\varepsilon CFG} = \{(G) \mid G$ è una CFG che genera $\varepsilon\}$. Dimostrare che $A_{\varepsilon CFG}$ è decidibile.
**Soluzione.**

**Teorema 14.** *$A_{\varepsilon CFG}$ è decidibile.*

*Proof.* Presentiamo un algoritmo che decide $A_{\varepsilon CFG}$:
**Algoritmo:** Su input $\langle G \rangle$ dove $G = (V, \Sigma, R, S)$:

Listing 3: Algoritmo per $A_{\varepsilon CFG}$

```python
def decides_A_epsilon_CFG(G):
    V, Sigma, R, S = G
    nullable = set()

    # Trova tutte le variabili che possono generare epsilon
    changed = True
    while changed:
        changed = False
        for production in R:
            lhs, rhs = production

            # Se la produzione   A -> epsilon
            if rhs == []:
                if lhs not in nullable:
                    nullable.add(lhs)
                    changed = True

            # Se la produzione   A -> B1 B2 ... Bk e tutti i Bi sono nullable
            elif all(symbol in nullable for symbol in rhs if symbol in V):
                if lhs not in nullable:
                    nullable.add(lhs)
                    changed = True

    # Verifica se la variabile iniziale   nullable
    return S in nullable
```

**Spiegazione dell'algoritmo:**

1. Inizializza un insieme *nullable* vuoto
2. Ripeti fino a quando non ci sono cambiamenti:
   - Per ogni produzione $A \to \alpha$:
     - Se $\alpha = \varepsilon$, aggiungi $A$ a *nullable*
     - Se $\alpha = B_1 B_2 \cdots B_k$ e tutti i $B_i$ che sono variabili sono in *nullable*, aggiungi $A$ a *nullable*
3. Restituisci vero se e solo se $S \in \mathit{nullable}$

**Correttezza:**

- L'algoritmo calcola esattamente l'insieme delle variabili che possono derivare $\varepsilon$
- Una variabile $A$ può derivare $\varepsilon$ se e solo se:
  - Esiste una produzione $A \to \varepsilon$, oppure

  - Esiste una produzione $A \to B_1 B_2 \cdots B_k$ dove ogni $B_i$ può derivare $\varepsilon$

- Il linguaggio $L(G)$ contiene $\varepsilon$ se e solo se la variabile iniziale $S$ può derivare $\varepsilon$

**Complessità temporale:**

- Il ciclo while viene eseguito al massimo $|V|$ volte (ogni iterazione aggiunge almeno una variabile a *nullable*)
- Ogni iterazione richiede $O(|R| \cdot \text{lunghezza massima delle produzioni})$ tempo
- Complessità totale: $O(|V| \cdot |R| \cdot \ell)$ dove $\ell$ è la lunghezza massima delle produzioni

Quindi $A_{\varepsilon CFG}$ è decidibile in tempo polinomiale. □

**Esercizio 4.16 - Espressioni Regolari con 111 8.** Sia $A = \{(R) \mid R$ è un'espressione regolare che de $x111y$ per qualche $x$ e $y)\}$. Dimostrare che $A$ è decidibile.
**Soluzione.**

**Teorema 15.** *$A$ è decidibile.*

*Proof.* Presentiamo un algoritmo che decide $A$:
**Algoritmo:** Su input $\langle R\rangle$:

1. Costruisci un DFA $M$ che riconosce $L(R)$ (il linguaggio descritto da $R$)
2. Costruisci un DFA $M'$ che riconosce il linguaggio $\Sigma^*111\Sigma^*$ (tutte le stringhe che contengono 111 come sottostringa)
3. Costruisci un DFA $M''$ che riconosce $L(M) \cap L(M')$ usando il prodotto cartesiano
4. Testa se $L(M'') = \emptyset$ usando l'algoritmo di raggiungibilità degli stati accettanti
5. Accetta se $L(M'') \neq \emptyset$, rifiuta altrimenti

**Dettagli della costruzione:**
**Passo 1:** Conversione da espressione regolare a DFA

- Usa l'algoritmo standard: $R \to NFA \to DFA$
- Complessità: esponenziale nella dimensione di $R$, ma finita

**Passo 2:** DFA per $\Sigma^*111\Sigma^*$

Listing 4: DFA per stringhe contenenti 111

```python
def construct_111_DFA():
    # Stati: q0 (inizio), q1 (visto 1), q2 (visto 11), q3 (
      visto 111, accettante)
    states = ['q0', 'q1', 'q2', 'q3']
    alphabet = ['0', '1']
    start_state = 'q0'
    accept_states = {'q3'}

```

```
    transitions = {
        ('q0', '0'): 'q0',
        ('q0', '1'): 'q1',
        ('q1', '0'): 'q0',
        ('q1', '1'): 'q2',
        ('q2', '0'): 'q0',
        ('q2', '1'): 'q3',
        ('q3', '0'): 'q3',
        ('q3', '1'): 'q3'
    }

    return (states, alphabet, transitions, start_state,
       accept_states)
```

**Passo 3:** Prodotto cartesiano

- Stati: $Q_1 \times Q_2$
- Stato iniziale: $(q_{01}, q_{02})$
- Stati accettanti: $F_1 \times F_2$
- Transizioni: $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$

**Passo 4:** Test di vuotezza

Listing 5: Test di vuotezza per DFA

```
def is_empty(DFA):
    states, alphabet, transitions, start, accept = DFA

    # BFS per trovare stati raggiungibili
    visited = set()
    queue = [start]
    visited.add(start)

    while queue:
        current = queue.pop(0)

        # Se raggiungiamo uno stato accettante, il linguaggio
           non    vuoto
        if current in accept:
            return False

        for symbol in alphabet:
            next_state = transitions.get((current, symbol))
            if next_state and next_state not in visited:
                visited.add(next_state)
                queue.append(next_state)

```

```
    # Nessuno stato accettante    raggiungibile
    return True
```

**Correttezza:**

- $L(R)$ contiene una stringa con sottostringa 111 se e solo se $L(R) \cap (\Sigma^*111\Sigma^*) \neq \emptyset$
- Il prodotto cartesiano calcola esattamente questa intersezione
- Il test di vuotezza determina se l'intersezione è vuota

**Complessità:**

- La conversione da regex a DFA può essere esponenziale, ma è finita per ogni input
- Il prodotto cartesiano e il test di vuotezza sono polinomiali nella dimensione dei DFA
- Quindi l'algoritmo termina sempre in tempo finito

Quindi $A$ è decidibile. □

# 5 Modelli di Computazione Alternativi

**Esercizio 3.11 - Macchina di Turing con Nastro Doppiamente Infinito 9.** Una *macchina di Turing con nastro doppiamente infinito* è simile a una macchina di Turing ordinaria, ma il suo nastro è infinito sia a sinistra che a destra. Il nastro è inizialmente riempito con blanks eccetto per la porzione che contiene l'input. La computazione è definita come al solito eccetto che la testina non incontra mai una fine del nastro mentre si muove verso sinistra. Dimostrare che questo tipo di macchina di Turing riconosce la classe dei linguaggi Turing-riconoscibili.
**Soluzione.**

**Teorema 16.** *Le macchine di Turing con nastro doppiamente infinito riconoscono esattamente la classe dei linguaggi Turing-riconoscibili.*

*Proof.* Dobbiamo dimostrare entrambe le direzioni:

1. Ogni linguaggio riconosciuto da una TM standard può essere riconosciuto da una TM con nastro doppiamente infinito
2. Ogni linguaggio riconosciuto da una TM con nastro doppiamente infinito può essere riconosciuto da una TM standard

**Direzione 1: TM standard $\Rightarrow$ TM doppiamente infinita**
Questo è immediato: una TM con nastro doppiamente infinito può simulare una TM standard semplicemente ignorando la parte sinistra del nastro e operando solo sulla parte destra.
**Direzione 2: TM doppiamente infinita $\Rightarrow$ TM standard**

Sia $M$ una TM con nastro doppiamente infinito. Costruiamo una TM standard $M'$ che simula $M$.

**Idea chiave:** Codifichiamo il nastro doppiamente infinito su un nastro standard usando due tracce.

**Codifica del nastro:**

- Il nastro standard ha due tracce:
  - **Traccia superiore**: contiene le celle $0, 1, 2, 3, \ldots$ del nastro doppiamente infinito
  - **Traccia inferiore**: contiene le celle $-1, -2, -3, \ldots$ del nastro doppiamente infinito (in ordine inverso)
- La cella 0 (posizione iniziale della testina) è nella prima posizione della traccia superiore

**Rappresentazione visiva:**

Nastro doppiamente infinito di $M$:

| | | | | | |
|---|---|---|---|---|---|
| $\cdots$ | $c_{-2}$ | $c_{-1}$ | $c_0$ | $c_1$ | $c_2$ |
| $\cdots$ | | | | | |

Nastro standard di $M'$ (due tracce):

| | | | | |
|---|---|---|---|---|
| Traccia sup. | $c_0$ | $c_1$ | $c_2$ | $\cdots$ |
| Traccia inf. | $c_{-1}$ | $c_{-2}$ | $c_{-3}$ | $\cdots$ |

**Simulazione dei movimenti:**

**Posizione corrente:** $M'$ mantiene informazioni aggiuntive nel suo stato per ricordare:

- Se la testina di $M$ è su una cella positiva (traccia superiore) o negativa (traccia inferiore)
- La posizione esatta sulla traccia corrispondente

**Movimento a destra (da cella $i$ a cella $i + 1$):**

- Se $i \geq 0$: rimane sulla traccia superiore, si sposta a destra
- Se $i = -1$: passa dalla traccia inferiore (posizione 1) alla traccia superiore (posizione 1)

**Movimento a sinistra (da cella $i$ a cella $i - 1$):**

- Se $i \geq 1$: rimane sulla traccia superiore, si sposta a sinistra
- Se $i = 0$: passa dalla traccia superiore (posizione 1) alla traccia inferiore (posizione 1)
- Se $i \leq -1$: rimane sulla traccia inferiore, si sposta a destra

**Algoritmo di simulazione:**

Listing 6: Simulazione TM doppiamente infinita

```
def simulate_doubly_infinite_TM(M, input_string):
    # Inizializza il nastro con due tracce
    upper_track = list(input_string) + [BLANK] * 1000  # espandibile
    lower_track = [BLANK] * 1000  # espandibile

    # Posizione corrente: (traccia, indice)
    current_pos = ('upper', 0)
    current_state = M.start_state

    while current_state not in M.halt_states:
        # Leggi simbolo corrente
        if current_pos[0] == 'upper':
            current_symbol = upper_track[current_pos[1]]
        else:
            current_symbol = lower_track[current_pos[1]]

        # Applica transizione di M
        new_state, new_symbol, direction = M.delta(current_state, current_symbol)

        # Scrivi nuovo simbolo
        if current_pos[0] == 'upper':
            upper_track[current_pos[1]] = new_symbol
        else:
            lower_track[current_pos[1]] = new_symbol

        # Muovi testina
        current_pos = move_head(current_pos, direction)
        current_state = new_state

    return current_state in M.accept_states

def move_head(current_pos, direction):
    track, index = current_pos

    if direction == 'R':
        if track == 'upper':
            return ('upper', index + 1)
        else:  # track == 'lower'
            if index == 0:
                return ('upper', 0)
            else:
                return ('lower', index - 1)

```

```python
    else:  # direction == 'L'
        if track == 'upper':
            if index == 0:
                return ('lower', 0)
            else:
                return ('upper', index - 1)
        else:  # track == 'lower'
            return ('lower', index + 1)
```

**Correttezza:**

- La codifica preserva tutte le informazioni del nastro doppiamente infinito
- Ogni movimento della testina doppiamente infinita è simulato correttamente
- La simulazione termina se e solo se la macchina originale termina
- Il risultato (accetta/rifiuta) è preservato

**Complessità:**

- Ogni passo di $M$ richiede al massimo un numero costante di passi in $M'$
- Se $M$ fa $t$ passi e usa $s$ celle, allora $M'$ fa $O(t)$ passi e usa $O(s)$ celle

Quindi le macchine di Turing con nastro doppiamente infinito hanno lo stesso potere computazionale delle macchine di Turing standard. □

# 6 Conclusioni

Attraverso la risoluzione di questi esercizi, abbiamo esplorato diversi aspetti fondamentali della teoria della computazione:

- **NP-completezza**: Tecniche di riduzione per dimostrare l'intrattabilità computazionale di problemi apparentemente diversi
- **Decidibilità vs Indecidibilità**: La distinzione cruciale tra problemi risolvibili algoritmicamente e problemi intrinsecamente irrisolvibili
- **Riconoscibilità**: Problemi che ammettono algoritmi di semi-decisione ma non algoritmi di decisione completa
- **Modelli computazionali**: L'equivalenza tra diversi formalismi di computazione e la robustezza della classe dei linguaggi Turing-riconoscibili

Questi risultati illustrano la ricchezza e la profondità della teoria della computazione, mostrando come principi matematici astratti abbiano implicazioni concrete per la progettazione di algoritmi e la comprensione dei limiti computazionali.