

PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

0. NOTAZIONI

0.1 INSIEMI

Un **insieme** è una collezione non ordinata di oggetti o elementi. Gli insiemi sono scritti tra { }, ed i suoi elementi inseriti tra esse.

Per ogni insieme S , $w \in S$ indica che w è un **elemento** di S .

Nota: Notazione di insiemi per specificare un' insieme: $A = \{x | x \in R, f(x) = 0\}$, R è l'insieme dei numeri reali, f è una qualche funzione.

Ordine e ridondanza non contano:

- $\{a, b, c\}$ ha elementi a, b, c ;
- $\{a, b, c\}$ e $\{b, a, b, c, c\}$ sono lo stesso insieme;
- $\{a\}$ ed a sono cose diverse;
- $\{a\}$ insieme che contiene solo elemento a .

Esempio:

- L'insieme dei numeri naturali è $N = \{0, 1, 2, 3, 4, \dots\}$.
- L'insieme dei numeri pari è $\{0, 2, 4, 6, 8, 10, 12, \dots\} = \{2n | n = 0, 1, 2, 3, \dots\} = \{2n | n \in N\}$.
- L'insieme dei pari positivi è $\{2, 4, 6, 8, 10, 12, \dots\} = \{2n | n = 1, 2, 3, \dots\} = \{2n | n \in N^+\}$.
- L'insieme dei numeri dispari è $\{1, 3, 5, 7, 9, 11, 13, \dots\} = \{2n+1 | n = 0, 1, 2, \dots\} = \{2n+1 | n \in N\}$.
- Se $A = \{2n | n \in N\}$, allora $4 \in A$, ma $5 \notin A$.

La **cardinalità** $|S|$ di S è il numero di elementi in S .

Esempio:

- Se $S = \{ab, bb\}$ allora $|S| = 2$
- Se $T = \{an | n > 1\}$, allora $|T| = \infty$
- Se $T = \emptyset$, allora $|T| = 0$

Un insieme S è **finito** se $|S| < \infty$. Se S non è finito, allora è detto **infinito**.

Esempio:

- Se $S = \{ab, bb\}$ allora $|S| = 2$ e S è finito.
- Se $T = \{an | n > 1\}$, allora $|T| = \infty$ e T è infinito.

0.2 ALFABETO E STRINGHE

Un **alfabeto** è un insieme finito di elementi fondamentali (chiamati **lettere** o **simboli**).

Esempio:

- L'alfabeto delle lettere romane minuscole è $\Sigma = \{a, b, c, \dots, z\}$.
- L'alfabeto delle cifre arabe è $\Sigma = \{0, 1, \dots, 9\}$.
- L'alfabeto binario è $\Sigma = \{0, 1\}$

Una **stringa** su un **alfabeto** è una sequenza finita di simboli dell'alfabeto.

Esempio:

- cat, food, c, babbz sono stringhe sull'alfabeto $A = \{a, b, c, \dots, z\}$.
- 0131 è una stringa sull'alfabeto $B = \{0, 1, 2, \dots, 9\}$.
- 0101 è una stringa sull'alfabeto $B = \{0, 1\}$.

Date una stringa s , la **lunghezza** di s è il numero di simboli in s . La lunghezza di s è denotata con **lunghezza(s)** o **|s|**.

Esempio:

$$\text{lunghezza(mom)} = |\text{mom}| = 3.$$

Nota. La **stringa vuota** ϵ è la stringa contenente nessun simbolo, $|\epsilon| = 0$.

Dato alfabeto Σ , la **chiusura di Kleene** di Σ è Σ^* : l'insieme di tutte le possibili stringhe su Σ .

Esempio:

$$\Sigma = \{a, b\}, \text{ allora } \Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\}$$

Date due stringhe u e v , la **concatenazione** di u e v è la stringa uv .

Esempio:

- $u = abb$ e $v = ab$, allora $uv = abbab$ e $vu = ababb$
- $u = \epsilon$ e $v = ab$, allora $uv = ab$
- $u = bb$ e $v = \epsilon$, allora $uv = bb$
- $u = \epsilon$ e $v = \epsilon$, allora $uv = \epsilon$; cioè $\epsilon\epsilon = \epsilon$

Per una stringa w , definiamo w^n per $n \geq 0$ induttivamente:

$$w^0 = \epsilon$$

$$w^{n+1} = w^n w, \text{ per ogni } n \geq 1$$

Esempio:

- Se $w = \text{cat}$, allora $w^0 = \epsilon$,
 $w^1 = \text{cat}$,
 $w^2 = \text{catcat}$,
 $w^3 = \text{catcatcat}$,
...
- Dato simbolo a , allora $a^0 = \epsilon$ e $a^3 = \text{aaa}$.

Data stringa s , una **sottostringa** di s è una qualsiasi parte di simboli consecutivi della stringa s cioè, w è una sottostringa di s se esistono stringhe x e y (eventualmente vuote) tali che: $s = xwy$.

Esempio:

- 567 è sottostringa di 56789
- 567 è sottostringa di 45678
- 567 è sottostringa di 34567
- Stringa 472 ha sottostringhe ϵ , 4, 7, 2, 47, 72, 472, ma 42 non è sottostringa di 472.

0.3 LINGUAGGI

Un **Linguaggio formale** (Linguaggio) è un insieme di stringhe su un alfabeto.

Esempio:

Linguaggi per computer, quali C, C++ o Java, sono linguaggi formali con alfabeto:
 $\{a, b, \dots, z, A, B, \dots, Z, 0, 1, 2, \dots, 9, >, <, =, +, -, *, /, (,), \dots\}$

Le **regole della sintassi** definiscono le regole del linguaggio. L'insieme di nomi validi di variabili è un linguaggio formale.

Nota: I linguaggi non sono insiemi finiti.

Esempio:

- Alfabeto $A = \{x\}$.
Linguaggio $L = \{\epsilon, x, xx, xxxx, xxxxx, \dots\} = \{x^n \mid n = 0, 1, 2, 3, \dots\}$.
Nota: $x^0 = \epsilon$, quindi stringa vuota in L .
- Alfabeto $A = \{x\}$.
Linguaggio $L = \{x, xxx, xxxx, \dots\} = \{x^{2n+1} \mid n = 0, 1, 2, 3, \dots\}$.
- Alfabeto $A = \{0, 1, 2, \dots, 9\}$. Linguaggio $L = \{\text{qualsiasi stringa che non inizia con } 0\} = \{\epsilon, 1, 2, 3, \dots, 9, 10, 11, \dots\}$
- Sia $A = \{a, b\}$, definiamo Linguaggio L formato da tutte le stringhe che iniziano con a seguita da 0 o più b .
Cioè $L = \{a, ab, abb, abbb, \dots\} = \{ab^n \mid n \geq 0\}$

Nota. L'insieme vuoto \emptyset è l'insieme che non contiene alcun elemento.

$\emptyset = \{\epsilon\}$, poiché \emptyset non ha elementi. In generale, $\epsilon \notin \emptyset$.

0.4 INSIEMI: RELAZIONI ED OPERAZIONI

Siano S e T insiemi. Diciamo che $S \subseteq T$ (S **sottoinsieme** di T) se $w \in S$ implica $w \in T$. Cioè ogni elemento di S è anche un elemento T .

Esempio:

- $S = \{ab, ba\}$ e $T = \{ab, ba, aaa\}$ allora $S \subseteq T$ ma $T \not\subseteq S$.
- $S = \{ba, ab\}$ e $T = \{aa, ba\}$ allora $S \not\subseteq T$ e $T \not\subseteq S$.

Insiemi S e T sono **uguali** ($S = T$) se $S \subseteq T$ e $T \subseteq S$.

Esempio:

- Siano $S = \{ab, ba\}$ e $T = \{ba, ab\}$, allora $S \subseteq T$ e $T \subseteq S$; quindi $S = T$.
- Siano $S = \{ab, ba\}$ e $T = \{ba, ab, aaa\}$, allora $S \subseteq T$ ma $T \not\subseteq S$; quindi $S \neq T$.

Dati due insiemi S e T , la loro **unione** $S \cup T = \{w \mid w \in S \text{ oppure } w \in T\}$.

$S \cup T$ contiene tutti gli elementi contenuti in S oppure in T (o in entrambi).

Esempio:

- $S = \{ab, bb\}$ e $T = \{aa, bb, a\}$ allora $S \cup T = \{ab, bb, aa, a\}$
- $S = \{a, ba\}$ e $T = \emptyset$, allora $S \cup T = S$.
- $S = \{a, ba\}$ e $T = \{\epsilon\}$ allora $S \cup T = \{\epsilon, a, ba\}$

Dati due insiemi S e T , la loro **intersezione** $S \cap T = \{w \mid w \in S \text{ e } w \in T\}$. $S \cap T$ contiene tutti gli elementi comuni ad S e T .

Insiemi S e T si dicono **disgiunti** se $S \cap T = \emptyset$

Esempio:

- Sia $S = \{ab, bb\}$ e $T = \{aa, bb, a\}$ allora $S \cap T = \{bb\}$.
- Sia $S = \{ab, bb\}$ e $T = \{aa, ba, a\}$ allora $S \cap T = \emptyset$, quindi S e T sono disgiunti.

Lemma. Se S e T sono disgiunti (cioè $S \cap T = \emptyset$), allora $|S \cup T| = |S| + |T|$.

Lemma. Se S e T sono tali che $S \cap T < \infty$, allora $|S \cup T| = |S| + |T| - |S \cap T|$.

Dati due insiemi S e T , $S - T = \{w \mid w \in S \text{ e } w \notin T\}$.

Esempio:

- Sia $S = \{a, b, bb, bbb\}$ e $T = \{a, bb, bab\}$, allora $S - T = \{b, bbb\}$.
- Sia $S = \{ab, ba\}$ e $T = \{ab, ba\}$ allora $S - T = \emptyset$

Dato un insieme universale U , il **complemento** di un insieme $S \subseteq U$ è $C(S) = \{w \mid w \in U, w \notin S\}$.

$C(S)$ è l' insieme di tutti gli elementi considerati (elementi di U) che non sono in S (quindi $C(S) = U - S$).

Esempio:

- U : insieme delle stringhe su alfabeto $\{a, b\}$.
 S : insieme delle stringhe su alfabeto $\{a, b\}$ che iniziano con b .
 $C(S)$: insieme delle stringhe su alfabeto $\{a, b\}$ che non iniziano con b .

N.B.: NON insieme stringhe che iniziano con a (es. stringa vuota ϵ)

Dati 2 insiemi S e T di stringhe, la **concatenazione** (o **prodotto**) di S e T è $ST = \{uv \mid u \in S, v \in T\}$

ST è l' insieme di stringhe che possono essere divise in 2 parti: la prima parte coincide con una stringa in S la seconda parte coincide con una stringa in T .

Esempio:

- Se $S = \{a, aa\}$ e $T = \{\epsilon, a, ba\}$, allora $ST = \{a, aa, aba, aaa, aaba\}$, $TS = \{a, aa, aaa, baa, baaa\}$
- $aba \in ST$, ma $aba \notin TS$. Quindi $ST \neq TS$

0.5 SEQUENZE E TUPLE

Una **sequenza** di oggetti è una lista di questi oggetti in qualche ordine.

Ordine e ridondanza sono importanti in una sequenza (non in un insieme).

Sequenze finite sono dette tuple. Una k -tuple ha k elementi nella sequenza.

Esempio:

- $(4, 2, 7)$ è una 3-tupla o tripla
- $(9, 23)$ è una 2-tupla o coppia

0.6 PRODOTTO CATESIANO

Dati due insiemi A e B , il **prodotto Cartesiano** $A \times B$ è insieme di coppie: $A \times B = \{(x, y) \mid x \in A, y \in B\}$.

Esempio:

- Siano $A = \{a, ba, bb\}$ e $B = \{\epsilon, ba\}$, allora:
 $A \times B = \{(a, \epsilon), (a, ba), (ba, \epsilon), (ba, ba), (bb, \epsilon), (bb, ba)\}$
 $B \times A = \{(\epsilon, a), (\epsilon, ba), (\epsilon, bb), (ba, a), (ba, ba), (ba, bb)\}$.

Nota. $(ba, a) \in B \times A$, ma $(ba, a) \notin A \times B$, quindi $B \times A \neq A \times B$.

Nota. Il prodotto Cartesiano è diverso dalla Concatenazione $AB = \{a, aba, ba, baba, bb, bbba\} = A \times B$.

Nota. $|A \times B| = |A| |B|$, possiamo anche definire prodotto cartesiano di più di 2 insiemi. $A_1 \times \dots \times A_k$ è l' insieme di k -tuple:
 $A_1 \times \dots \times A_k = \{(x_1, \dots, x_k) \mid x_i \in A_i, 1 \leq i \leq k\}$

Esempio:

- Siano $A_1 = \{ab, ba, bbb\}$, $A_2 = \{a, bb\}$, $A_3 = \{ab, b\}$, allora:
 $A_1 \times A_2 \times A_3 = \{(ab, a, ab), (ab, a, b), (ab, bb, ab), (ab, bb, b),$
 $(ba, a, ab), (ba, a, b), (ba, bb, ab), (ba, bb, b),$
 $(bbb, a, ab), (bbb, a, b), (bbb, bb, ab), (bbb, bb, b)\}$

0.7 INSIEME POTENZA

Per ogni insieme S , l' **insieme potenza** $P(S)$ è $P(S) = \{A \mid A \subseteq S\}$, cioè l' insieme di tutti possibili sottoinsiemi di S (inclusi \emptyset e S stesso).

Esempio:

Se $S = \{a, bb\}$, allora $P(S) = \{\emptyset, \{a\}, \{bb\}, \{a, bb\}\}$

Lemma. Se $|S| < \infty$, allora $|P(S)| = 2^{|S|}$. Cioè , ci sono $2^{|S|}$ differenti sottoinsiemi di S .

0.8 CHIUSURA O KLEENE STAR

Dato insieme S di stringhe, sia

$$S^0 = \{\epsilon\},$$

$$S^k = \{w_1, w_2 \dots w_k \mid w_i \in S, i = 1, 2, \dots, k\} = SS\dots S, k > 1.$$

concatenazione di S con se stesso per k volte.

Nota. S^k è insieme di stringhe ottenute concatenando k stringhe di S , con possibili ripetizioni. In particolare, $S^1 = S$.

Esempio:

Se $S = \{a, bb\}$, allora

$$S_0 = \{\epsilon\},$$

$$S_1 = \{a, bb\},$$

$$S_2 = \{aa, abb, bba, bbbb\},$$

$$S_3 = \{aaa, aabb, abba, abbb, bbaa, bbabb, bbbbba, bbbbb\}.$$

La **Chiusura (o Kleene star)** di un insieme di stringhe S è $S^* = S_0 \cup S_1 \cup S_2 \cup S_3 \cup \dots$

Nota. S^* è l' insieme di tutte le stringhe ottenute concatenando zero o più stringhe di S , potendo usare la stessa stringa più volte.

$$S^* = \{w_1, w_2 \dots w_k \mid k \geq 0, w_i \in S, i = 1, 2, \dots, k\}, \text{ dove per } k = 0, \text{ la stringa } w_1, w_2 \dots w_k = \epsilon \text{ è la stringa vuota.}$$

Esempio:

- Se $S = \{ba, a\}$, allora $S^* = \{\epsilon, a, aa, ba, aaa, aba, baa, aaaa, aaba, \dots\}$
- Se $A = \{a, b\}$, allora $A^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots\}$, tutte le possibili stringhe su alfabeto A .
- Se $S = \emptyset$, allora $S^* = \{\epsilon\}$.
- Se $S = \{\epsilon\}$, allora $S^* = \{\epsilon\}$.

0.9 ALTRE PROPRIETÀ UTILI

$S^{**} = (S^*)^*$ è l' insieme di stringhe formate concatenando stringhe di S^* .

Nota. $S^* = S^*$ per ogni insieme S di stringhe.

S^+ è l' insieme di stringhe formate concatenando una o più stringhe di S .

Esempio:

Se $S = \{x\}$, allora $S^+ = \{x, xx, xxx, xxxx, \dots\}$

Per ogni stringa w , l'**inverso di w** , scritto **reverse(w) o w^R** , è la stessa stringa di simboli scritta in ordine inverso .

Se $w = w_1, w_2, \dots w_n$, dove ogni w_i è un simbolo, allora $w^R = w_n w_{n-1} \dots w_1$.

Esempio:

- $(cat)^R = tac$
- $\epsilon^R = \epsilon$.

1. AUTOMI

Linguaggio delle soluzioni:

Un **cammino** è rappresentato da qualche $x \in \{l, p, c, n\}^*$.

{ $x \in \{l, p, c, n\}^*$ | iniziando in stato start e seguendo transizioni di x , terminano nello stato finale}

Nota. Il linguaggio delle soluzioni è infinito.

Da un **Problema** possiamo ricavarne un **Linguaggio**:

- **Problema:** trovare sequenza di mosse che permette di trasportare capra, cavolo e lupo;
- **Linguaggio:** insieme delle stringhe sull'alfabeto $\{l, p, c, n\}$. Le stringhe corrispondono a sequenze di mosse. Tra esse cerchiamo una stringa che corrisponde ad una soluzione del problema.

Un **Automa Completamente Specificato**, fornisce una procedura computazionale per decidere se una stringa è una soluzione o meno:

- Si parte dallo **stato start**;
- Si segue una **transizione** per ogni simbolo di input;
- Se alla fine si arriva in uno **stato obiettivo** (accettante) accetta, altrimenti rifiuta l'input.

1.1 AUTOMI FINITI

Modello semplice di calcolatore avente una quantità finita di memoria. È noto come **macchina a stati finiti** o **automa finito**.

Idea di base del funzionamento:

- Input= stringa w su un alfabeto Σ ;
- Legge i simboli di w da sinistra a destra;
- Dopo aver letto l'ultimo simbolo, l'automa indica se accetta o rifiuta la stringa input w .

1.1.1 AUTOMA FINITO DETERMINISTICO (DFA)

Sia A un **automa finito deterministico (DFA)** è 5-tupla, $M = (Q, \Sigma, f, q_1, F)$, dove:

1. Q è **insieme finito** di stati.
2. Σ è **alfabeto**, e il DFA processa stringhe su Σ .
3. $f: Q \times \Sigma \rightarrow Q$ è la **funzione di transizione**.
4. $q_1 \in Q$ è lo **stato start (o iniziale)**.
5. F (sottoinsieme di Q) è l'insieme di **stati accettanti (o finali)**.

Nota. DFA è chiamato anche **automa finito (FA)**.

Funzione di transizione $f: Q \times \Sigma \rightarrow Q$:

Definisce le regole per il cambio di stato. Se l'automa finito ha un arco da uno stato q_1 a uno stato q_2 , etichettato col simbolo di input b , questo significa che se l'automa è nello stato q_1 e legge una b , allora si muove nello stato q_2 .

Possiamo indicare la stessa cosa con la funzione di transizione, dicendo che $f(q_1, b) = q_2$, dove $q_1 \in Q$ ed $b \in \Sigma$.

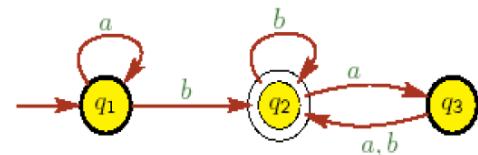
Se esiste almeno un arco uscente da q_1 con label b , allora la macchina è **deterministica**, ovvero una macchina è deterministica se il suo comportamento è specificatamente definito.

Esempio:

Possiamo descrivere M_1 formalmente ponendo $M_1 = (Q, \Sigma, f, q_1, F)$, dove:

1. $Q = \{q_1, q_2, q_3\}$;
2. $\Sigma = \{a, b\}$
3. f è descritto come segue:

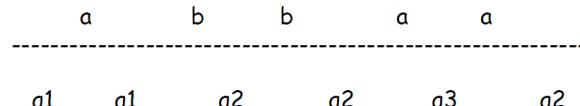
	a	b
q1	q1 q2	q2 q2
q2	q3 q2	q2 q2
q3	q2	q2



4. q_1 è lo stato start (ha freccia entrante da esterno);
5. $F = \{q_2\}$ (doppio cerchio)

DFA con alfabeto $\Sigma = \{a, b\}$:

- Stringa arriva in input
- DFA legge 1 simbolo alla volta dal primo all'ultimo
- DFA accetta o rifiuta la stringa



Esempi di input: la stringa abaa è accettata se e solo se esiste una sequenza di stati q_1, q_1, q_2, q_3, q_2 .

Definizione formale del funzionamento di un automa:

Sia $M = (Q, \Sigma, f, q_0, F)$, consideriamo la stringa $w = w_1 w_2 \dots w_n$ su Σ , dove ogni w_i in Σ .

Allora M accetta w se esiste una **sequenza di stati** r_0, r_1, \dots, r_n in Q con tre condizioni:

1. $r_0 = q_0$ (primo stato della sequenza è quello iniziale)
2. $r_n \in F$ (ultimo stato in sequenza è uno stato accettante)
3. $f(r_i, w_{i+1}) = r_{i+1}$, per ogni $i = 0, 1, 2, \dots, n-1$ (sequenza di stati corrisponde a transizioni valide per la stringa w).

Verificate le tre condizioni, si può affermare che M riconosce il **linguaggio** A , se $A = \{w \mid M \text{ accetta } w\}$.

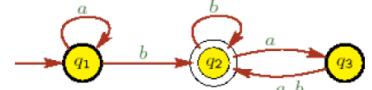
Linguaggio della Macchina:

Se L è l'insieme di tutte le stringhe che la macchina M accetta, allora si dice che L è il **linguaggio della macchina** M e scriviamo $L(M)=A$, e diciamo che M riconosce L .

Un linguaggio è chiamato un **linguaggio regolare** se un automa finito DFA lo riconosce.

Esempio1:

$L(M)$ è l'insieme di tutte le stringhe sull'alfabeto $\{a,b\}$ della forma: $\{a\}^*\{b\}\{b,aa,ab\}^*$



Esempio2:

Si consideri il seguente DFA M_1 con alfabeto $\Sigma = \{0, 1\}$:

Osservazione: $L(M_1)$ è il Linguaggio di stringhe su Σ in cui il numero totale di 1 è dispari.



Esempio3:

DFA M_2 con alfabeto $\Sigma = \{0, 1\}$:

Nota: $L(M_2)$ è Linguaggio su A che ha lunghezza 1, cioè $L(M_2) = \{w \in A^* \mid |w| = 1\}$

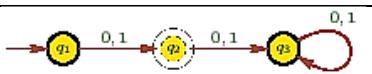
Si ricordi che $C(L(M_2))$, il complemento di $L(M_2)$, è l'insieme di stringhe su A che non sono in $L(M_2)$.

Esempio3.1:

Si consideri il seguente DFA M_3 con alfabeto $A = \{0, 1\}$

$L(M_3)$ è il Linguaggio su Σ che **non** ha lunghezza 1, più di uno stato accetta, stato start anche stato accetta.

DFA accetta ε se e solo se lo stato start è anche lo stato accetta.



Funzione di transizione estesa:

La **funzione di transizione f** può essere **estesa** ad f^* che opera su stati e stringhe (invece che su stati e simboli). In generale, è possibile definirla induttivamente:

$$f^*(q, \varepsilon) = q$$

$$f^*(q, xa) = f(f^*(q, x), a)$$

Il linguaggio accettato da M è quindi: $L(M) = \{w : f^*(q_0, w) \in F\}$

Informalmente, la funzione di transizione estesa dà come risultato lo stato dell'automa dopo aver letto tutti i simboli di una stringa, partendo da un dato stato dell'automa.

Esempio:

Si consideri il DFA M_1 precedentemente definito. La funzione di transizione estesa f^* è:

- $f^*(q_1, \varepsilon) = q_1;$
- $f^*(q_1, 001) = f(f^*(q_1, 00), 1) = f(q_1, 1) = q_2;$ // si è proceduto a ritroso
- $f^*(q_1, 00) = f(f^*(q_1, 0), 0) = f(q_1, 0) = q_1;$ // si è proceduto a ritroso
- $f^*(q_1, 0) = f(f^*(q_1, \varepsilon), 0) = f(q_1, 0) = q_1.$

1.1.2 COMPLEMENTO DFA

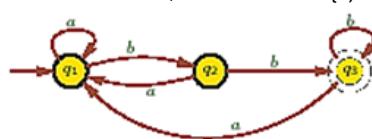
In generale, dato un DFA M per Linguaggio L, possiamo costruire un DFA M' per $C(L)$ da M:

- rendendo tutti gli stati accetta in M non-accetta in M' ;
- rendendo tutti stati non-accetta in M stati accetta in M' .

Se si ha il linguaggio L su alfabeto Σ , ha un DFA $M = (Q, \Sigma, f, q_1, F)$, allora il DFA per il **complemento** di $C(L)$ è: $M' = (Q, \Sigma, f, q_1, Q-F)$.

Esempio1:

Si consideri il DFA M_4 con alfabeto $\Sigma = \{a, b\}$:

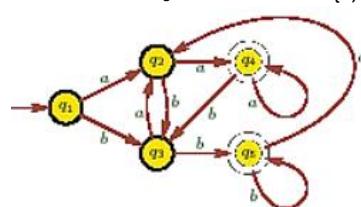


$L(M_4)$: Linguaggio di stringhe su Σ che terminano con bb , sarà:

$$L(M_4) = \{w \in \Sigma^* \mid w = sbb \text{ per qualche stringa } s\}$$

Esempio2:

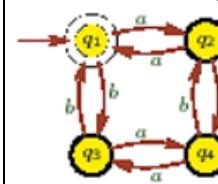
Si consideri il DFA M_5 con alfabeto $\Sigma = \{a, b\}$:



$$L(M_5) = \{w \mid w = saa \text{ o } w = sbb \text{ per una stringa } s\}$$

Esempio3:

Si consideri il DFA M_8 con alfabeto $\Sigma = \{a, b\}$:



Ogni a muove verso destra o sinistra.
Ogni b muove verso l'alto o il basso

DFA riconosce il Linguaggio di stringhe su Σ con numero pari di a e numero pari di b.

Particolari casi di automi:

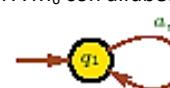
Si consideri il seguente DFA M_6 con alfabeto $\Sigma = \{a, b\}$:



Accetta tutte le possibili stringhe su Σ , cioè $L(M_6) = \Sigma^*$

In generale, ogni DFA in cui tutti gli stati sono stati accetta allora riconosce il linguaggio Σ^* .

Si consideri il seguente DFA M_7 con alfabeto $\Sigma = \{a, b\}$:



DFA non accetta stringhe su Σ , cioè $L(M_7) = \emptyset$

In generale, un DFA può non avere stati accetta.

1.2 OPERAZIONI REGOLARI (SU LINGUAGGI)

Siano A e B linguaggi:

- **Unione:** $A \cup B = \{w \mid w \in A \text{ o } w \in B\}$
- **Concatenazione:** $AB = \{vw \mid v \in A, w \in B\}$
- **Kleene star:** $A^* = \{w_1 w_2 \cdots w_k \mid k \geq 0 \text{ e ogni } w_i \in A\}$

Nota: Una collezione S di oggetti è **chiusa** per un'operazione f se applicando f a membri di S, f restituisce oggetto in S.

Esempio:

$N = \{0, 1, 2, \dots\}$ chiuso per addizione ($1+2=3 \in N$), non per sottrazione ($1-2=-1 \notin N$)

TEOREMA:

L'insieme dei linguaggi regolari è **chiuso** per l'operazione di **complemento**.

Abbiamo visto che dati DFA M_1 per Linguaggio L , possiamo costruire DFA M_2 per Linguaggio complemento L' :

- Rendi tutti stati accetta in M_1 in non-accetta in M_2 .
- Rendi tutti stati non-accetta in M_1 in accetta in M_2 .

Quindi L regolare $\rightarrow C(L)$ regolare.

TEOREMA:

La classe dei **linguaggi regolari** è **chiusa** per **l'unione**, cioè, se L_1 e L_2 sono linguaggi regolari, allora lo è anche $L_1 \cup L_2$.

Idea:

- L_1 ha DFA M_1
- L_2 ha DFA M_2

Una stringa w è in $L_1 \cup L_2$ sse w è accettata da M_1 oppure M_2 . Serve DFA M_3 che accetta w sse w è accettata da M_1 o M_2 . Costruiamo M_3 tale:

- Accetti un input esattamente quando M_1 o M_2 lo accetterebbero;
- Deve tener traccia di dove l'input si trovi contemporaneamente in M_1 ed in M_2 .

Dimostrazione:

Supponiamo che M_1 riconosca L_1 , dove $M_1 = (Q_1, \Sigma, f_1, q_1, F_1)$ ed M_2 riconosca L_2 , dove $M_2 = (Q_2, \Sigma, f_2, q_2, F_2)$.

Costruiamo M_3 che riconosce $L_1 \cup L_2$, dove $M_3 = (Q_3, \Sigma, f_3, q_3, F_3)$:

1. $Q_3 = Q_1 \times Q_2 = \{(x, y) \mid x \in Q_1, y \in Q_2\}$;
2. Alfabeto di M_3 è Σ , ovvero lo stesso di M_1 ed M_2 .
3. M_3 ha funzione di transizione $f_3: Q_3 \times \Sigma \rightarrow Q_3$, tale che per ogni x in Q_1 e y in Q_2 , a in Σ :
4. Lo stato start di M_3 è $q_3 = (q_1, q_2)$ in Q_3 .
5. L'insieme di stati accetta di M_3 è $F_3 = \{(x, y) \in Q_3 \mid x \text{ in } F_1 \text{ o } y \text{ in } F_2\}$

(Prodotto cartesiano di Q_1 e Q_2)

(Se gli alfabeti sono diversi, era: $\Sigma = \Sigma_1 \cup \Sigma_2$)

$f_3((x, y), a) = (f_1(x, a), f_2(y, a))$;

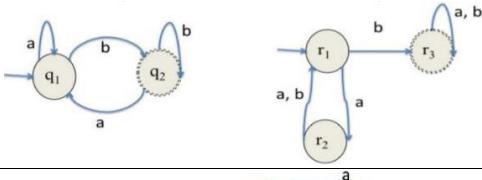
Poiché $Q_3 = Q_1 \times Q_2$, il numero di stati in M_3 è $|Q_3| = |Q_1| \cdot |Q_2|$.

Quindi, $|Q_3|$ è finito poiché $|Q_1|$ e $|Q_2|$ sono finiti.

Esempio:

Si considerino i seguenti DFA e linguaggi su $\Sigma = \{a, b\}$:

- DFA M_1 riconosce linguaggio $A_1 = L(M_1)$
- DFA M_2 riconosce linguaggio $A_2 = L(M_2)$



Il seguente è un DFA per l'unione di A_1 e A_2 . In particolare, si procede nel seguente modo.

1. Otteniamo gli stati dal prodotto cartesiano di Q_1 e Q_2 .

In particolare, lo stato iniziale è la coppia costituita dagli stati iniziali dei due automi;

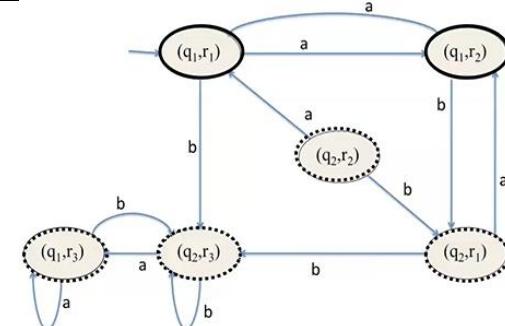
2a. Se ci si trova nello stato (q_1, r_1) e si legge a , allora si nota che il primo automa resta nello stato q_1 , mentre il secondo automa passa allo stato r_2 , quindi il risultato della funzione di transizione è lo stato (q_1, r_2) ed il nuovo automa passa in quest'ultimo stato;

2b. se ci si trova nello stato (q_1, r_1) e si legge b , allora si nota che il primo automa passa allo stato q_2 , mentre il secondo automa passa allo stato r_3 , quindi il risultato della funzione di transizione è lo stato (q_2, r_3) ed il nuovo automa passa in quest'ultimo stato;

...

3. Otteniamo gli stati finali come coppie dove è presente o lo stato finale dell'automa A o lo stato finale dell'automa B (o entrambi).

In questo esempio, gli stati finali sono: $(q_2, r_1), (q_2, r_2), (q_2, r_3), (q_1, r_3)$.



Nota: Lo stato (q_2, r_2) non è mai raggiungibile da alcuno stato dell'automa (compreso lo stato iniziale), per cui è possibile ometterlo.

TEOREMA:

La classe dei linguaggi regolari è **chiusa** per **l'intersezione**. Cioè, se L_1 e L_2 sono linguaggi regolari, allora lo è $L_1 \cap L_2$.

Idea:

- L_1 ha DFA M_1
- L_2 ha DFA M_2

Una stringa w è in $L_1 \cap L_2$ sse w è accettata sia da M_1 che M_2 . Serve un DFA M_3 che accetta w sse w è accettata da M_1 e M_2 : deve sapere se ambedue gli automi accettano la stringa w in input, e per ottenere ciò è possibile porre **in parallelo** M_1 e M_2 . Costruiamo M_3 tale:

- Accetti un input esattamente quando M_1 e M_2 lo accetterebbero;
- Deve tener traccia di dove l'input si trovi contemporaneamente in M_1 ed in M_2 .

Dimostrazione:

A tal punto, siano L_1 e L_2 definiti su uno stesso alfabeto Σ . Supponiamo che il DFA M_1 riconosca L_1 , dove $M_1 = (Q_1, \Sigma, f_1, q_1, F_1)$, e che il DFA M_2 riconosca L_2 , dove $M_2 = (Q_2, \Sigma, f_2, q_2, F_2)$. Costruiamo il DFA $M_3 = (Q_3, \Sigma, f_3, q_3, F_3)$ in questo modo:

- $Q_3 = Q_1 \times Q_2 = \{(x, y) \mid x \in Q_1, y \in Q_2\}$;

- l'alfabeto di M_3 è Σ ;

- M_3 ha $f_3 : Q_3 \times \Sigma \rightarrow Q_3$ tale che per ogni x in Q_1 , y in Q_2 , a in Σ : $f_3((x, y), a) = (f_1(x, a), f_2(y, a))$;

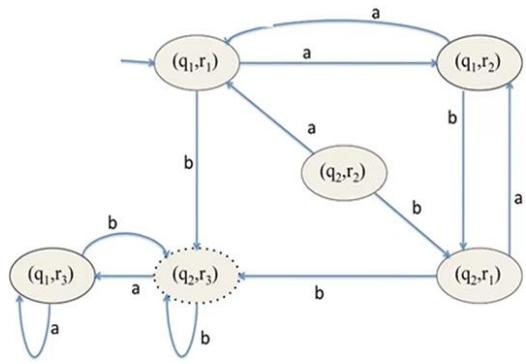
- lo stato iniziale di M_3 è $q_3 = (q_1, q_2)$ in Q_3 ;

- l'insieme di stati accetta di M_3 è $F_3 = \{(x, y) \in Q_3 \mid x \in F_1 \text{ e } y \in F_2\}$.

M_3 è un DFA che riconosce l'intersezione. Poiché $Q_3 = Q_1 \times Q_2$, allora il numero di stati in M_3 è $|Q_3| = |Q_1| \cdot |Q_2|$. Quindi, $|Q_3|$ è finito poiché $|Q_1|$ e $|Q_2|$ sono finiti.

Esempio:

Presi gli stessi automi, linguaggi ed alfabeto dall' esempio mostrato in precedenza, si nota che un automa che riconosce l'**intersezione** di A_1 e A_2 cambia soltanto gli stati finali rispetto all'unione. Bisogna, cioè, ottenere un automa che accetti stringhe che verrebbero accettate sia da M_1 sia da M_2 : quindi, gli stati finali sono le coppie dov'è presente sia lo stato finale di A_1 sia lo stato finale di A_2 ; lo stato finale è, in questo caso, (q_2, r_3) .



TEOREMA:

La classe dei linguaggi regolari è **chiusa** per la **concatenazione**. Cioè, se A_1 e A_2 sono linguaggi regolari, allora lo è anche A_1A_2 .

Per costruire un DFA che accetti la concatenazione di stringhe appartenenti ad automi diversi, lo si fa con un nuovo tipo di macchina, ovvero:

1.3 AUTOMI FINITI NON DETERMINISTICI

Il **non determinismo** è una generalizzazione del **determinismo**, quindi ogni automa finito deterministico è automaticamente un automa finito non deterministico.

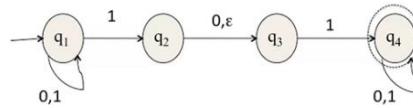
In un DFA, lo stato successivo occupato in corrispondenza di un dato input è unicamente determinato: quindi, le macchine sono deterministiche. La funzione di transizione in un DFA è $f: Q \times \Sigma \rightarrow Q$, che restituisce sempre un singolo stato.

Gli automi finiti non deterministici (NFA) permettono più scelte per il prossimo stato per un dato input. Per uno stato q , l'NFA può:

- avere più archi uscenti da q labellati con lo stesso simbolo a , per i vari $a \in \Sigma$;
- prendere ϵ -edge senza leggere simboli in input.

La **differenza** tra un automa deterministico e non è che:

- Lo stato q_1 ha un arco uscente per 0, ma ne ha 2 per 1;
- Lo stato q_2 ha un arco per 0 ma non ha alcuno per 1;



Gli Automi finiti non deterministici hanno uno stato che può avere 0 o più archi uscenti per ogni simbolo dell'alfabeto.

Questo NFA ha un arco con etichetta ϵ . In generale, un NFA può avere archi etichettati con elementi dell'alfabeto o ϵ . Zero, uno, o più archi possono uscire da ciascuno stato con l'etichetta ϵ .

Osservazione:

Se NFA è in stato con più scelte, allora la macchina si divide in più copie di sé stessa, ognuna di essa continua una computazione in maniera indipendente dalle altre.

NFA può essere un insieme di stati, invece di un singolo stato. NFA segue ogni possibile computazione in parallelo e al termine dell'input:

- se una copia giunge in stato accetta, NFA accetta la stringa;
- se nessun cammino giunge in stato accetta, allora NFA non accetta la stringa input.

Se in stato con ϵ -transition, senza leggere input,

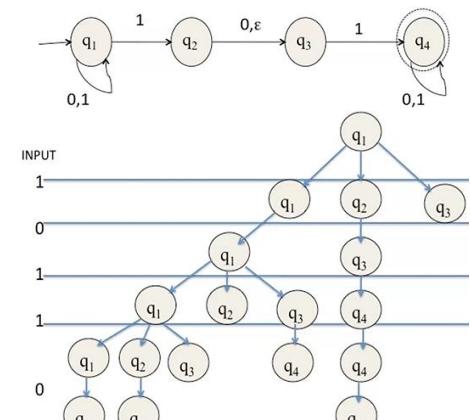
- NFA si divide in più copie, ognuna segue una possibile transizione,
- ogni copia continua indipendentemente da altre copie,
- NFA segue ogni possibile cammino in parallelo.
- NFA continua non deterministicamente come prima

Esempio1:

Sia definito il seguente NFA N_1 con alfabeto $A = \{0, 1\}$.

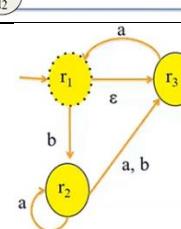
Si nota subito che lo stato iniziale q_1 , quando riceve in input il simbolo 1, può sia restare nello stato q_1 che passare agli stati q_2 e q_3 : questo perché quando l'automa è nello stato q_2 è possibile che venga ricevuta in input una ϵ vuota (una tale transizione prende il nome di epsilon-transition), di conseguenza si passa direttamente anche allo stato q_3 . Inoltre, può capitare che in uno stato l'automa non possa accettare un simbolo dell'alfabeto (in questo caso, nello stato q_3 non è possibile ricevere una label 0).

Se si riceve in input 10110, questa è la computazione che l'automa effettua →



Esempio2:

Dato l'NFA posto a lato, è possibile notare che esso accetta stringhe ϵ , a, aa, baa, baba, ...; non accetta, invece, stringhe b, ba, bb, ...



Per alfabeto Σ , sia Σ_ϵ ottenuto da Σ aggiungendo ϵ . Cioè, $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

Un automa finito non-deterministico NFA A è una 5-tupla $M = (Q, \Sigma, f, q_0, F)$, con

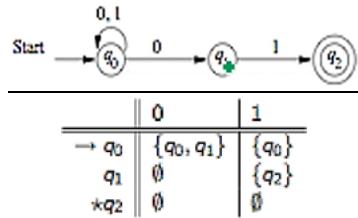
1. Q è l'insieme finiti di stati
2. Σ è un alfabeto, e il DFA processa stringhe su Σ ;
3. $f: Q \times \Sigma_\epsilon \rightarrow P(Q)$ funzione di transizione, dove $P(Q)$ è l'insieme di tutti i sottoinsiemi possibili di Q ;
4. $q_0 \in Q$ è stato start
5. $F \subseteq Q$ è insieme di stati accettanti.

Nota. La differenza tra DFA e NFA è nella funzione di transizione f , la quale:

- ammette mosse tipo ϵ
- restituisce Insieme di stati invece di un solo stato.

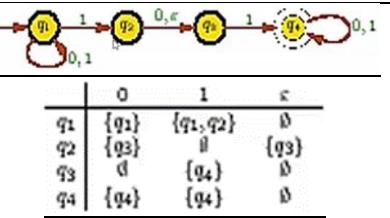
Esempio1:

Sia definito l'NFA $N = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, posto di seguito, dove δ è la funzione di transizione descritta nella tabella a lato.



Esempio2:

Sia definito l'NFA $N_2 = (\{q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_1, \{q_4\})$, posto di seguito, dove δ è la funzione di transizione descritta nella tabella a lato.



Nota: Si ricordi che, una volta che l'automa è nello stato q_2 , esso può passare allo stato q_3 in quanto è presente una ϵ -transition in tale stato.

Computazione di NFA:

Sia $N = (Q, \Sigma, f, q_0, F)$ un NFA e sia w una stringa sull'alfabeto Σ , allora N **accetta** w se possiamo scrivere w come $w = y_1, y_2, \dots, y_m$, dove ogni y_i è un elemento di Σ_ϵ ed esiste una sequenza di stati r_0, r_1, \dots, r_m in Q con tre condizioni:

1. $r_0 = q_0$ (La macchina inizia nello stato iniziale)
2. r_{i+1} in $f(r_i, y_{i+1})$ per ogni $i=0, 1, 2, \dots, m-1$ (Lo stato r_{i+1} è uno dei possibili stati successivi quando N è nello stato r_i e sta leggendo y_{i+1})
3. r_m in F (La macchina accetta il suo input se l'ultimo stato è uno stato accettante)

Informalmente, vuol dire che N accetta w se esiste una possibile computazione che porta allo stato di accettazione. Si noti che la stringa generica y_i non appartiene a Σ , bensì appartiene a Σ_ϵ in quanto è possibile che essa sia un ϵ per effettuare una epsilon-transition; è questa l'unica differenza tra la computazione di un DFA e la computazione di un NFA.

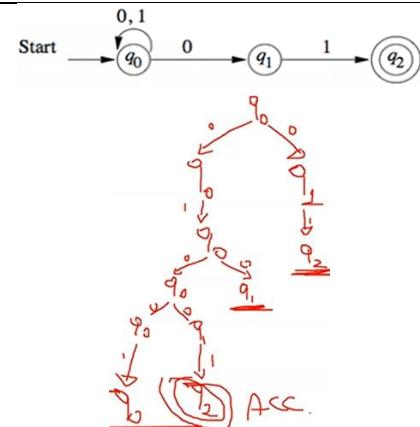
L'insieme di stringhe accettate da NFA N è il linguaggio **riconosciuto** da N ed è denotato con $L(N)$.

Nota: Ogni DFA è anche un NFA. Infatti, vedremo che è possibile esprimere un NFA attraverso un corrispondente DFA, e che i primi sono semplicemente un modo più compatto per descrivere i secondi.

Sia definito l'NFA N posto a lato. Allora esso accetta il linguaggio $\{x01 \mid x \in \Sigma^*\}$.

Ad esempio, se l'input è 01001 abbiamo il seguente albero di computazione:

- 0, da q_0 si passa sia a q_0 che a q_1 ;
- 1, da q_0 si passa nuovamente a q_0 , e da q_1 si passa a q_2 (in quest'ultimo stato, la computazione si arresta);
- 0, da q_0 si passa sia a q_0 che a q_1 ;
- 0, da q_0 si passa sia a q_0 che a q_1 , e da q_1 con 0 si arresta la computazione;
- 1, da q_0 si passa a q_0 e da q_1 si passa a q_2 , e in quest'ultimo stato la stringa viene accettata.



1.3.1 EQUIVALENZE DI DFA E NFA

Due macchine sono **equivalenti** se riconoscono lo stesso linguaggio.

TEOREMA: Per ogni automa finito non deterministico NFA esiste un automa finito deterministico DFA **equivalente**.

Ogni NFA N ha un equivalente DFA M , cioè se N è un NFA, allora esiste DFA M t.c. $L(M) = L(N)$.

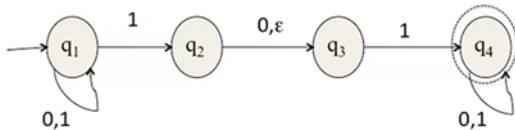
Idea:

Trasformare un NFA in un DFA equivalente che simula il NFA. Si dovrà capire quali saranno lo stato iniziale e quelli accettanti del DFA e la sua funzione di transizione.

Nota: Se k è il numero degli stati dell'NFA, esso avrà 2^k sottoinsiemi di stati. Ogni sottoinsieme corrisponde ad una delle possibilità che il DFA deve ricordare, quindi il DFA che simula l'NFA avrà 2^k stati.

Dimostrazione:

Costruiamo, a partire da N , il DFA M equivalente. L'idea è simile a quella usata per costruire l'automa che riconosce l'unione di due linguaggi: simuliamo tutte le computazioni fatte dal NFA in parallelo. Consideriamo il seguente automa.



L'automa deterministico dovrà, partendo dallo stato iniziale in presenza di input 1, ricordare all'interno dello stato che andrà ad occupare che l'automa non deterministico può essere in ognuno degli stati q_1, q_2 e q_3 : è possibile ottenere ciò andando a considerare uno stato che è dato dall'insieme degli stati $\{q_1, q_2, q_3\}$. Allo stesso modo, in presenza di input 0, seguendo la prima computazione allora si passa da q_1 a q_1 , mentre per la seconda si passa da q_2 a q_3 , e la terza computazione si arresta: questo si può considerare nell'automa deterministico come uno stato dato dall'insieme degli stati $\{q_1, q_3\}$. Si procede in questo modo per ogni possibile computazione. Otteniamo, quindi, il seguente DFA.

Si osservi che dallo stato $\{q_1, q_2, q_3\}$, in presenza di input 0, si passa allo stato $\{q_1, q_3\}$. Gli stati finali del DFA saranno tutti quelli in cui compare *almeno uno* tra gli stati finali del NFA: in questo caso, solo gli stati in cui compare q_4 .

Formalizzando, sia $L = L(N)$ per un NFA $N = (Q_N, \Sigma, f_N, q_N, F_N)$; allora, costruiamo un DFA $D = (Q_D, \Sigma, f_D, q_D, F_D)$.

Partiamo da un automa D' che non considera le ϵ -transition, dove, per ogni $r \in Q$ e $a \in \Sigma$:

- $Q = P(Q_N)$, ossia ogni stato in D' sarà un sottoinsieme dell'insieme potenza degli stati del NFA N ;
- $f(r, a) = \bigcup_{r' \in R} f_N(r, a)$, ossia la funzione di transizione darà come risultato l'unione dei risultati (per ogni elemento r dell'insieme R), della funzione di transizione $f_N(r, a)$;
- $q = \{q_N\}$, ossia lo stato iniziale sarà l'insieme in cui compare lo stato iniziale del NFA N ;
- $F = \{r \in Q \mid r \cap F_N \neq \emptyset\}$, ossia gli stati finali saranno tutti quegli stati che contengono al loro interno uno stato finale del NFA N .

Per ogni stato in f_N , se vi è una ϵ -transition allora dobbiamo considerarla. A tale scopo, definiamo l'insieme $E(r) = r \cup \{q \mid q \text{ è raggiungibile da uno stato in } r \text{ con 1 o più archi labellati } \epsilon\}$, cioè l'insieme che considera l'unione tra uno stato di D' (in quanto $r \in Q$) e l'insieme degli stati raggiungibili da uno stato in r che ammettono almeno una ϵ -transition.

Per la funzione di transizione estesa si ha che $f^*_N(q_1, 10110) = \{q_1, q_2, q_4\}$.

Formalmente, si ha che:

- $f_N^*(q_N, \epsilon) = E(\{q_N\})$, quindi non ci sono lettere lette ed applichiamo la sola epsilon-transition;
- $f_N^*(q_N, xa) = \bigcup_{r \in f_N^*(q_N, x)} E(f_N(r, a))$, quindi si prendono tutti gli stati in cui potrebbe trovarsi l'automa dopo aver letto la stringa $x \in \Sigma^*$, e per ognuno di questi stati applichiamo la funzione di transizione del NFA, applichiamo poi la funzione E , facciamo l'unione di ciò che abbiamo ottenuto ed otteniamo l'insieme di stati raggiungibili quando l'input è la stringa xa .

Risulta, per ogni $x \in \Sigma^*$, $f_D^*(q_D, x) = f_N^*(q_N, x)$, cioè che le funzioni di transizione estese del NFA e del DFA corrispondente coincidono (ciò significa che i due automi accettano le stesse stringhe); questo risultato si può dimostrare induttivamente.

Quindi, sappiamo che D simula N su ogni input x . Inoltre, D accetta x se e solo se N accetta x . Infine, il linguaggio $L = L(N) = L(D)$. Ciò significa che D e N sono equivalenti.

Dimostrazione:

Mostriamo per induzione su $|w|$, ponendo $q_0 = q_N$, che $f_D^*(\{q_0\}, w) = f_N^*(q_0, w)$.

Base: $w = \epsilon$. L'enunciato segue dalla definizione.

Passo: Supponiamo che l'uguaglianza è vera per una certa stringa x , e consideriamo una stringa xa :

$$\begin{aligned}
 f_D^*(\{q_0\}, xa) &= f_D(f_D^*(\{q_0\}, x), a) && (\text{per definizione}) \\
 f_D^*(\{q_0\}, xa) &= f_D(f_N^*(q_0, x), a) && (\text{per ipotesi induttiva}) \\
 f_D^*(\{q_0\}, xa) &= \bigcup_{r \in f_N^*(q_0, x)} E(f_N(r, a)) && (\text{per definizione}) \\
 f_D^*(\{q_0\}, xa) &= f_N^*(q_0, xa).
 \end{aligned}$$

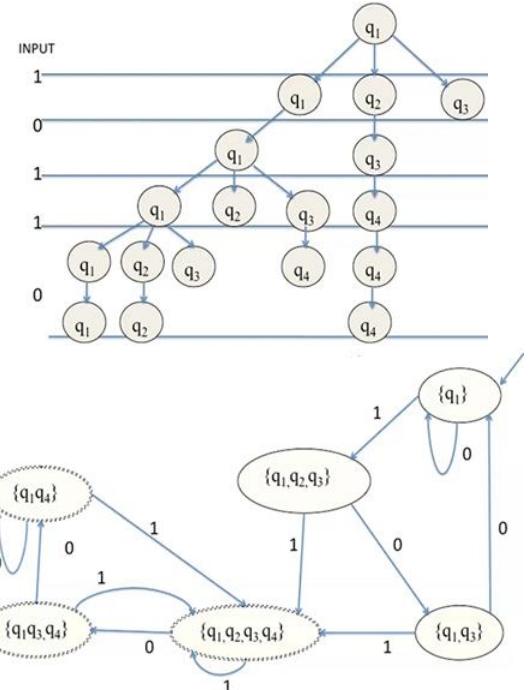
Esempio 1:

Considerando l'esempio precedente, si ha ad esempio che $E(\{q_1\}) = \{q_1\}$, e $E(\{q_2\}) = \{q_2, q_3\}$: questo perché q_1 non accetta epsilon-transition, a differenza di q_2 . Quindi, avremo il DFA D :

- $Q_D = P(Q_N)$, che coincide con quanto definito per D' ;
- $f_D(r, a) = \bigcup_{r' \in R} E(f_N(r, a))$, ossia la funzione di transizione darà come risultato l'unione dei risultati (per ogni elemento r dell'insieme R), della E della funzione di transizione $f_N(r, a)$;
- $q_D = E(\{q_N\})$, ossia lo stato iniziale sarà l'insieme in cui compare lo stato iniziale del NFA N ;
- $F_D = \{r \in Q_D \mid r \cap F_N \neq \emptyset\}$, che coincide con quanto definito per D' .

Considerando l'esempio precedente, si ha ad esempio che $f_N(q_1, 1) = \{q_1, q_2\}$ se non consideriamo le ϵ -transition.

Se consideriamo le ϵ -transition, invece, si ha che $E(f_N(q_1, 1)) = \{q_1, q_2, q_3\}$.



Esempio 2:

Considerando l'automa posto a lato, otteniamo che il suo stato iniziale sarà $E(\{r_1\}) = \{r_1, r_3\}$.

Abbiamo, inoltre, il suo DFA equivalente:

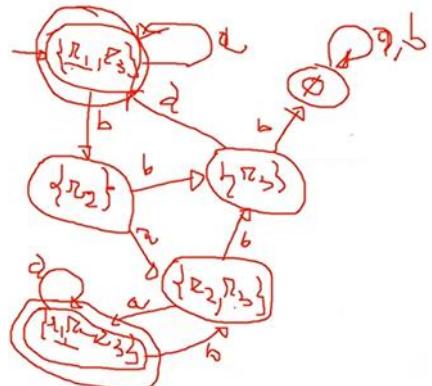
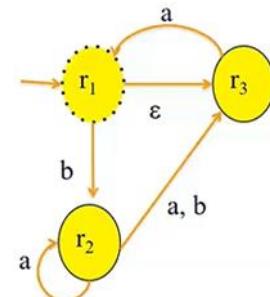
- $\{r_1, r_3\}$, con input 'a' si ha che da r_1 non si passa ad alcuno stato, mentre da r_3 si passa allo stato r_1 . A questo punto, si applica nuovamente la funzione E (in quanto r_1 accetta una ϵ -transition) e si passa ancora dallo stato r_1 allo stato r_3 . Il prossimo stato sarà, quindi, ancora $\{r_1, r_3\}$;
- $\{r_1, r_3\}$, con input 'b' si ha che da r_1 si passa a r_2 , mentre da r_3 non si passa ad alcuno stato. Lo stato r_2 non ammette ϵ -transition, quindi il nuovo stato sarà $\{r_2\}$;
- $\{r_2\}$, con input 'a' si ha che da r_2 si passa ad r_2 e da r_2 si passa ad r_3 , quindi il nuovo stato sarà $\{r_2, r_3\}$;
- $\{r_2\}$, con input 'b' si ha che da r_2 si passa ad r_3 , quindi il nuovo stato sarà $\{r_3\}$;
- $\{r_2, r_3\}$, con input 'a' da r_2 vale il punto precedente, mentre da r_3 si passa ad r_1 , ma in quest'ultimo caso applicando la E otteniamo che si passa anche da r_1 a r_3 .

Di conseguenza, il nuovo stato sarà $\{r_1, r_2, r_3\}$;

- $\{r_2, r_3\}$, con input 'b' da r_2 vale il punto precedente, mentre da r_3 non c'è nessuna transizione. Di conseguenza, il nuovo stato sarà $\{r_3\}$;
- $\{r_3\}$, con input 'a' da r_3 si passa a r_1 , ma applicando la E otteniamo che si passa anche da r_1 a r_3 . Il nuovo stato sarà, quindi, lo stato iniziale $\{r_1, r_3\}$;
- $\{r_3\}$, con input 'b' non si passa ad alcuno stato. Tuttavia, l'automa è deterministico per cui non possiamo ignorare un eventuale input, per cui si passa allo stato vuoto;
- considerando lo stato vuoto, per ogni input si passa nuovamente allo stato vuoto;
- $\{r_1, r_2, r_3\}$, con input 'a' ogni stato passa a quello successivo, per cui il nuovo stato sarà ancora $\{r_1, r_2, r_3\}$;
- $\{r_1, r_2, r_3\}$, con input 'b' si ha che da r_1 si passa ad r_2 , da r_2 si passa ad r_3 , mentre da r_3 non si passa ad alcuno stato. Quindi, il nuovo stato sarà $\{r_1, r_3\}$.

Ora, bisogna identificare gli stati finali: sono tutti quegli stati al cui interno è presente lo stato r_1 .

L'automa deterministico corrispondente avrà, quindi, come stati finali $\{r_1, r_3\}$ e $\{r_1, r_2, r_3\}$.



Nota: $f^*N(q_N, x) =$ insieme di stati raggiungibili da q_N su input x . Ciò vuol dire che la funzione estesa di un automa non deterministico può dare come risultato più stati, in quanto è possibile avere diverse copie di computazione.

Corollario:

Un linguaggio è regolare se e solo se qualche automa finito non deterministico lo riconosce.

Dimostrazione:

Se L è regolare, allora esiste un DFA che lo riconosce; ma ogni DFA è anche un NFA, quindi esiste un NFA per L .

Dal teorema precedente, ogni NFA ha un equivalente DFA. Quindi, se esiste un NFA che riconosce L , allora esiste un DFA che riconosce L .

Formalmente: $L = L(N) = L(D) \Rightarrow \exists D: L(D) = L(N) = L$; quindi, L è un linguaggio regolare.

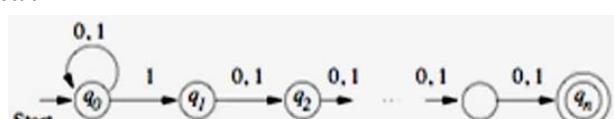
Si osservi che, sebbene sia possibile ottenere un DFA equivalente da un determinato NFA, è bene (spesso) utilizzare questi ultimi perché sono più facili da rappresentare.

Nota. Esiste un NFA N con $n+1$ stati che non ha nessun DFA equivalente con meno di 2^n stati.

Il linguaggio dell'automa a dx è $L(N) = \{x1c_2c_3 \dots | x \in \{0, 1\}^*, c_i \in \{0, 1\}\}$.

Questo automa è simile all'automa che riconosce tutte le stringhe che terminano con 01, ma è costruito in modo tale che lo stato iniziale cicli sia con 0 che con 1, dopodiché ha una transizione con 1 da q_0 a q_1 (quindi si ha uno sdoppiamento della computazione); inoltre, dallo stato q_1 allo stato q_n si avanza con qualsiasi simbolo che riceve in input. Quindi, una stringa per essere accettata deve avere qualunque simbolo all'inizio, un 1 e poi $n-1$ simboli qualsiasi.

In sintesi, mentre un NFA può sdoppiarsi ad ogni stato ed avere più computazioni, un DFA deve necessariamente rappresentare gli stati in modo tale che possa ricordare gli ultimi n simboli che ha letto. Questo non è semplice da realizzare, in quanto ci sono 2^n sequenze di bit (da rappresentare con opportuni stati) $a_1a_2 \dots a_n$ da ricordare.



1.4 OPERAZIONI REGOLARI CON GLI AUTOMI

Lo scopo delle operazioni regolari è quello di provare che **unione**, **concatenazione** e **kleene star** di linguaggi regolari sono ancora regolari.

L'uso del non determinismo rende le prove molto più semplici.

1.4.1 OPERAZIONE DI UNIONE PER GLI AUTOMI

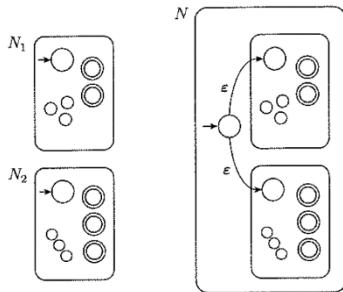
TEOREMA:

La classe dei **linguaggi regolari** è chiusa rispetto all'operazione di **unione**.

Idea:

Dati due linguaggi regolari A_1 e A_2 riconosciuti da N_1 e N_2 , vogliamo provare che $A_1 \cup A_2$ è **regolare** componendo un nuovo NFA N .

La macchina N deve accettare il suo input se N_1 o N_2 accetta questo input. La nuova macchina ha un nuovo stato iniziale che si dirama negli stati iniziali delle vecchie macchine con ϵ -archi. In questo modo, la nuova macchina ipotizza non deterministicamente quale delle due macchine accetta l'input. Se una di essi accetta, allora anche N lo accetterà.



Dimostrazione:

Sia $N_1 = (Q_1, \Sigma, f_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \Sigma, f_2, q_2, F_2)$ che riconosce A_2 .

Costruiamo $N = (Q, \Sigma, f, q, F)$ per $A_1 \cup A_2$:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$
Gli stati di N sono tutti gli stati di N_1 ed N_2 , con l'aggiunta di un nuovo stato iniziale q_0 .
2. Lo stato q_0 è lo stato iniziale di N .
3. L'insieme degli stati accettanti $F = F_1 \cup F_2$
Gli stati accettanti di N sono tutti gli stati accettanti di N_1 ed N_2 . In questo modo, N accetta se N_1 accetta o N_2 accetta.
4. Definiamo f in modo che per ogni q in Q e ogni a in Σ_ϵ :

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ e } a = \epsilon \\ \emptyset & q = q_0 \text{ e } a \neq \epsilon. \end{cases}$$

1.4.2 OPERAZIONE DI CONCATENAZIONE PER GLI AUTOMI

Si ricordi che, dati due linguaggi A, B regolari, la concatenazione è l'insieme $AB = \{vw \mid v \in A, w \in B\}$.

TEOREMA:

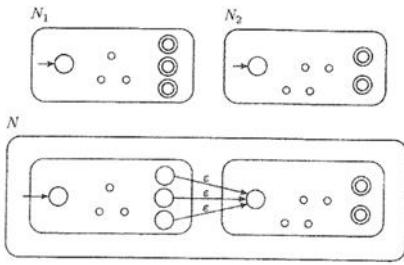
La classe dei **linguaggi regolari** è chiusa rispetto all'operazione di **concatenazione**.

Idea:

Dati due linguaggi regolari A_1 e A_2 riconosciuti da N_1 e N_2 , vogliamo provare che $A_1 \circ A_2$ è **regolare** componendo un nuovo NFA N .

Poniamo come stato iniziale di N , lo stato iniziale di N_1 . Gli stati accettanti di N_1 hanno degli ulteriori ϵ -archi che non deterministicamente permettono di diramarsi in N_2 ogni volta che N_1 è in uno stato accettante, indicando che ha trovato un pezzo iniziale dell'input che costituisce una stringa in A_1 .

Gli stati accettanti di N sono solo gli stati accettanti di N_2 . Quindi, esso accetta quando l'input può essere diviso in due parti, la prima accettata da N_1 e la seconda da N_2 . Possiamo pensare che N non deterministicamente ipotizza dove effettuare la divisione.



Dimostrazione:

Sia $N_1 = (Q_1, \Sigma, f_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \Sigma, f_2, q_2, F_2)$ che riconosce A_2 .

Costruiamo $N = (Q, \Sigma, f, q, F)$ per $A_1 \circ A_2$:

1. $Q = Q_1 \cup Q_2$
Gli stati di N sono tutti gli stati di N_1 ed N_2 .
2. L'alfabeto Σ è lo stesso dei due linguaggi L_1 ed L_2 .
3. Lo stato q_0 è uguale allo stato iniziale di N_1 .
4. Gli stati accettanti F_2 sono uguali agli stati accettanti di N_2 .
5. Definiamo f in modo che per ogni q in Q e ogni a in Σ_ϵ :

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ e } a \neq \epsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ e } a = \epsilon \\ \delta_2(q, a) & q \in Q_2. \end{cases}$$

La funzione di transizione $\delta(q, a)$, dato un generico stato q ed una stringa a :

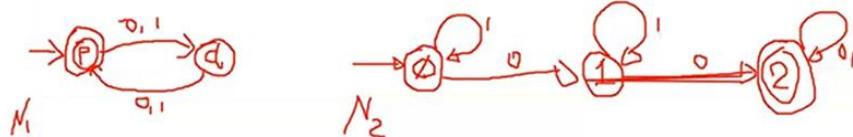
- se q è uno stato non finale del primo automa, allora la transizione è quella specificata dal primo automa;
- se q è uno stato finale del primo automa e a è diversa da epsilon, allora la transizione è quella specificata dal primo automa;
- se q è uno stato finale del primo automa e a è ϵ , allora alle transizioni che già si avevano per il primo automa vanno aggiunte anche le ϵ -transition verso lo stato iniziale del secondo automa;
- se q è uno stato del secondo automa, allora le transizioni sono quelle specificate dal secondo automa.

Analogamente, supponiamo di avere una stringa $w = uv$, non appartenente a L_1L_2 : questo significa che $u \notin L_1$ OR $v \notin L_2$. Accadrà, quindi, che la stringa non verrà accettata da (almeno) uno dei sottoautomi, di conseguenza w non verrà accettata dall'automa N .

La concatenazione è stata definita per due linguaggi, ma in realtà è possibile iterare ed andare a concatenare più di due linguaggi: per concatenare L_1, L_2, L_3 , ad esempio, si può prima concatenare L_1 ed L_2 per poi procedere con la concatenazione di L_3 . In generale, questo risultato si può estendere alla Kleene-star: $L^* = \{x_1x_2 \dots x_k \mid k \geq 0, x_i \in L, i = 1, 2, \dots, k\}$.

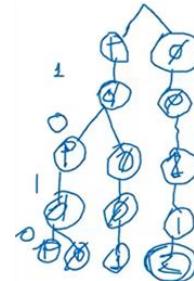
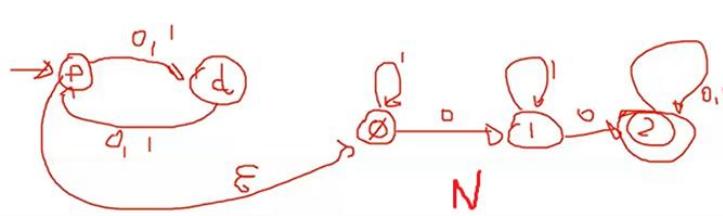
Esempio 1:

Siano $L_1 = \{w \mid |w| \text{ è pari}\}$ e $L_2 = \{w \mid w \text{ contiene almeno due } 0\}$. Vogliamo costruire gli automi per questi due linguaggi, e a partire da questi ultimi un automa per la loro concatenazione L_1L_2 . A tal scopo, sia N_1 l'automa per L_1 , e sia N_2 l'automa per L_2 . Otteniamo il seguente risultato.



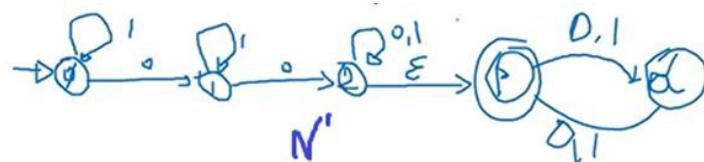
Quindi, sia $L(N) = L_1 L_2$. Dall'idea dimostrativa del teorema precedente, otteniamo il seguente automa N.

Notiamo che, ad esempio, la stringa 10100 è accettata da N, dato che è gestibile come stringa costituita dalle sottostringhe $10 \in L_1$ e $100 \in L_2$. Questa stringa è gestita dall'automa come descritto dall'albero a destra.



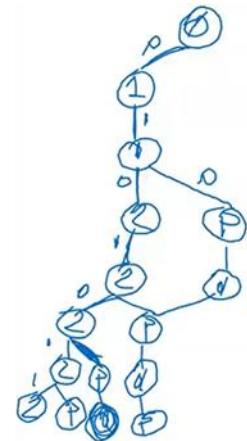
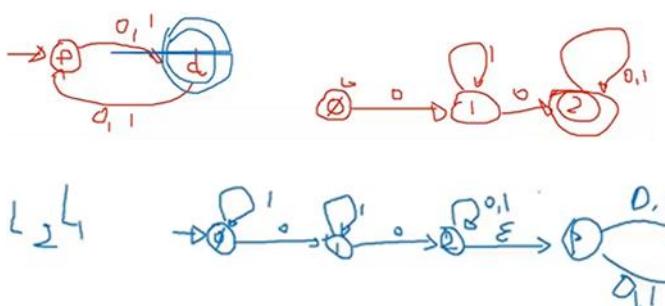
Esempio 2:

Dati i linguaggi L_1 ed L_2 precedenti, vogliamo costruire un automa per la loro concatenazione $L_2 L_1$. A tal scopo, sia N_1 l'automa per L_1 , e sia N_2 l'automa per L_2 . Quindi, sia $L(N') = L_2 L_1$. Le stringhe accettate devono, quindi, essere costituite da una sottostringa contenente due o più 0, e da una sottostringa ad essa concatenata di lunghezza pari. Dall'idea dimostrativa del teorema precedente, otteniamo il seguente automa N' .



Esempio 3:

Siano $L_1 = \{w \mid |w| \text{ è dispari}\}$ e $L_2 = \{w \mid w \text{ contiene almeno due } 0\}$. Vogliamo costruire gli automi per questi due linguaggi, e a partire da questi ultimi un automa per la loro concatenazione $L_1 L_2$. A tal scopo, sia N_1 l'automa per L_1 , e sia N_2 l'automa per L_2 . Otteniamo il risultato sottostante. Ad esempio, la stringa 1010111 (gestibile come $1010 \in L_2$ e $111 \in L_1$) viene accettata. La 0101011 (gestibile come $0101 \in L_2$ e $011 \in L_1$) segue l'albero computazionale di cui a destra.



1.4.3 OPERAZIONE STAR PER GLI AUTOMI

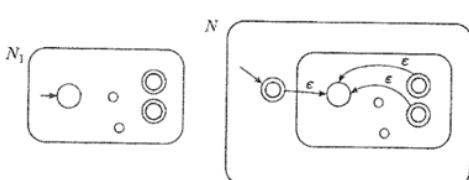
TEOREMA:

La classe dei **linguaggi regolari** è chiusa rispetto all'operazione **star**.

Idea:

Dato un linguaggio regolare A_1 e vogliamo provare che anche A_1^* è regolare.

Prendiamo un NFA N_1 per A_1 e lo modifichiamo per riconoscere A_1^* . L'NFA N risultante accetterà il suo input quando esso può essere diviso in varie parti ed N_1 accetta ogni parte. Possiamo costruire N come N_1 con ϵ -archi supplementari che dagli stati accettanti ritornano allo stato iniziale. In questo modo, quando l'elaborazione giunge alla fine di una parte che N_1 accetta, la macchina N ha la scelta di tornare indietro allo stato iniziale per provare a leggere un'altra parte che N_1 accetta. Inoltre, si aggiunge un nuovo stato iniziale, che è anche uno stato accettante, e che ha un ϵ -arco entrante nel vecchio stato iniziale.



Dimostrazione:

Sia $N_1 = (Q_1, \Sigma, f_1, q_1, F_1)$ che riconosce A_1 .

Costruiamo $N = (Q, \Sigma, f, q_0, F)$ per A_1^* :

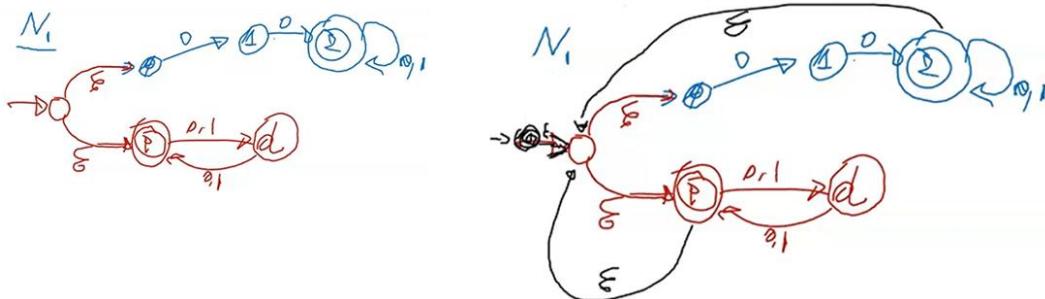
1. $Q = \{q_0\} \cup Q_1$
Gli stati di N sono tutti gli stati di N_1 più un nuovo stato iniziale.
2. Lo stato q_0 è il nuovo stato iniziale.
3. $F = \{q_0\} \cup F_1$
Gli stati accettanti sono i vecchi stati accettanti più il nuovo stato iniziale.
4. Definiamo f in modo che per ogni q in Q e ogni a in Σ :

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ e } a \neq \epsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ e } a = \epsilon \\ \{q_1\} & q = q_0 \text{ e } a = \epsilon \\ \emptyset & q = q_0 \text{ e } a \neq \epsilon. \end{cases}$$

Esempio 1:

Sia $L = \{w \mid |w| \text{ è pari OR } w = 00x\}$. Un automa non deterministico che riconosca L è il seguente (N_1 , a sinistra), dove in rosso è posto il sottoautoma che accetta le stringhe pari, e in blu è posto l'automa che riconosce le stringhe che iniziano con 00.

Notiamo che l'OR, in un automa non deterministico, è semplicemente rappresentabile aggiungendo uno stato iniziale generico collegato con una ϵ -transition agli "ex stati iniziali" dei sottoautomati. Questo è possibile in quanto, con i NFA, la computazione si sdoppia direttamente all'inizio e prosegue nei due sottoautomati. A destra, invece, è posto l'automa N che riconosce la Kleene-star di L , cioè $L^* = L(N)$.



Ad esempio, la stringa 0011 (considerabile come unica stringa che inizia con due 0, oppure come stringa costituita da $x_1 = 00$ concatenata con $x_2 = 11$) viene accettata.

Più in generale, per la costruzione dell'OR (**unione**) supponiamo di avere due linguaggi, L_1 ed L_2 , riconosciuti rispettivamente da due automi N_1 e N_2 . Per costruire l'automa N , che riconosce il linguaggio $L_1 \cup L_2$:

1. si crea un nuovo stato, che sarà lo stato iniziale dell'automa N ;
2. colleghiamo lo stato iniziale di N con i due stati iniziali di N_1 e N_2 .

In questo modo, abbiamo un NFA che riconosce tutte le stringhe dell'unione, in quanto (per ogni stringa) la computazione si sdoppia ed avremo due rami computazionali: uno eseguito da N_1 e uno eseguito da N_2 . Ora:

- se la stringa appartiene all'unione, allora essa o arriva in uno stato finale di N_1 o arriva in uno stato finale di N_2 (o in entrambi), e viene accettata dall'automa N ;
- se la stringa non appartiene all'unione, allora essa non arriva in alcuno stato finale di N_1 o N_2 , e non viene accettata dall'automa N .

2. ESPRESSIONI REGOLARI

Una **espressione regolare** è un modo dichiarativo per descrivere un **linguaggio regolare**. Possiamo usare le **operazioni regolari** per costruire espressioni che descrivono **linguaggi**, che sono chiamate **espressioni regolari**.

$$(0 \cup 1)^*$$

Il valore di questa espressione è un linguaggio, che consiste di tutte le stringhe che iniziano con un 0 o un 1, seguito da qualsiasi numero di simboli uguali a 0. La prima parte, ovvero i simboli 0 e 1 sono le abbreviazioni per gli insiemi {0} e {1}, quindi $(0 \cup 1)$ significa $\{\{0\} \cup \{1\}\}$. Il valore di questa parte è il linguaggio {0, 1}. La seconda parte, ovvero 0^* denota $\{0\}^*$, il suo valore è il linguaggio che consiste di tutte le stringhe contenenti, cioè un numero qualsiasi di simboli uguali a 0 (anche nessuno).

Sia $A = \{0, 1\}$. Per brevità, nelle espressioni regolari: 0 indica {0}, 1 indica {1}. In pratica, eliminiamo la notazione insiemistica per brevità.

Esempio: $0 \cup 1$ indica $\{0\} \cup \{1\}$, cioè {0, 1}.

Esempio: $(0 \cup 1)0^*$ è $\{0, 1\}\{0\}^*$ (dove $\{0\}^* = \{\epsilon, 0, 00, 000, \dots\}$), cioè l'insieme di stringhe binarie che iniziano con 0 oppure 1 e continuano con degli 0 (anche nessuno).

Esempio: $(0 \cup 1)^*$ è $\{0, 1\}^*$, cioè l'insieme di tutte le stringhe su {0, 1}.

2.1 DEFINIZIONE FORMALE DI UNA ESPRESSIONE REGOLARE (DEFINIZIONE INDUTTIVA)

Le espressioni regolari possono essere definite formalmente utilizzando una definizione induttiva. La base è:

- a è espressione regolare per ogni a nell'alfabeto, e denota l'insieme {a};
- ϵ è espressione regolare, e denota l'insieme $\{\epsilon\}$;
- \emptyset è espressione regolare, e denota l'insieme $\{\emptyset\}$.

Siano, ora, R_1 e R_2 espressioni regolari che rappresentano i linguaggi L_1 e L_2 . Si ha che:

- l'**unione** ($R_1 \cup R_2$) rappresenta l'insieme $L_1 \cup L_2$;
- la **concatenazione** (R_1R_2) rappresenta l'insieme L_1L_2 ;
- la **chiusura di Kleene** (R_1) * rappresenta l'insieme $L_1^* = \{0\}^*$.

Ogni espressione regolare può essere costruita a partire dal risultato ottenuto.

Esempio:

Sia $R_1 = 0$ l'e.r. che rappresenta $L_1 = \{0\}$, e sia $R_2 = 1$ l'e.r. che rappresenta $L_2 = \{1\}$. Allora:

- l'unione delle due espressioni regolari ($R_1 \cup R_2$) = $0 \cup 1$ rappresenta l'insieme $L_1 \cup L_2 = \{0, 1\}$;
- la chiusura di Kleene (R_1) * = 0^* rappresenta l'insieme $L_1^* = \{0\}^*$.

Siano $E_1 = 0 \cup 1$ e $E_2 = 0^*$ espressioni regolari che rappresentano rispettivamente gli insiemi $L_1' = \{0, 1\}$ e $L_2' = \{0\}^*$; allora si ha che la concatenazione $E_1E_2 = (0 \cup 1)0^*$ è un'espressione regolare che rappresenta il linguaggio $L_1'L_2' = \{0, 1\}\{0\}^*$.

Se R è un'espressione regolare, allora $L(R)$ denota il linguaggio generato da R .

Esempio:

$R = (0 \cup 1)0^*$ è un'espressione regolare che genera il linguaggio $L(R) = \{0, 1\}\{0\}^*$.

Definizione induttiva di espressione regolare (e.r.):

BASE:

- ϵ e \emptyset sono espressioni regolari: $L(\epsilon) = \{\epsilon\}$ e $L(\emptyset) = \emptyset$;
- se $a \in \Sigma$, allora a è un'espressione regolare: $L(a) = \{a\}$.

PASSO: Se R_1 e R_2 sono espressioni regolari, allora:

- $R_1 \cup R_2$ è un'espressione regolare che rappresenta $L(R_1) \cup L(R_2)$;
- R_1R_2 è un'espressione regolare che rappresenta $L(R_1)L(R_2)$;
- R_1^* è un'espressione regolare che rappresenta $(L(R_1))^*$.

Le espressioni regolari possono essere create applicando le operazioni di unione, concatenazione e Kleene star. Possiamo utilizzare le parentesi per cambiare l'ordine usuale di precedenza per le operazioni regolari, che è il seguente: 1. **Kleene star** (*); 2. **Concatenazione** (·); 3. **Unione** (U).

Esempio:

1. $01^* \cup 1$ è raggruppato in $(01)^* \cup 1$, cioè il linguaggio che riconosce la stringa 1 oppure le stringhe che iniziano con 0 e terminano con zero o più 1.

2. $00 \cup 101^*$ è il linguaggio formato da una stringa 00 o da stringhe iniziante con 10 seguite da zero o più 1.

3. Sia $0(0 \cup 101)^*$ un'espressione regolare. Prima si effettua la concatenazione di 101, poi si effettua l'unione tra 0 e 101, di quest'unione si effettua la Kleene star, ed infine si effettua la concatenazione. Allora:

- 0101001010 appartiene al linguaggio;
 - 00101001 non appartiene al linguaggio;
 - 0000000 appartiene al linguaggio;
 - 101 non appartiene al linguaggio.
4. Sia $A = \{0, 1\}$. Allora:
1. $(0 \cup 1)$ genera il linguaggio {0, 1};
 2. 0^*10^* genera il linguaggio {w | w ha un solo 1};
 3. A^*1A^* genera il linguaggio {w | w ha almeno un 1};
 4. A^*001A^* genera il linguaggio {w | w ha 001 come sottostringa};
 5. $(AA)^*$ genera il linguaggio {w | |w| è pari}, in quanto $(AA)^* = (\{0, 1\}\{0, 1\})^* = \{00, 01, 10, 11\}^*$;
 6. $(AAA)^*$ genera il linguaggio {w | |w| è multiplo di 3}.

5. Sia EVEN-EVEN su $A = \{a, b\}$ l'insieme di stringhe con un numero pari di a e un numero pari di b. Ad esempio, aababbaaababab \in EVEN-EVEN. Si ha che l'e.r. $(aa \cup bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba))^*$ genera il linguaggio EVEN-EVEN.

Questo esempio fa comprendere che le espressioni regolari consentono di definire una classe di linguaggi che è esattamente quella dei linguaggi regolari.

2.2 EQUIVALENZA CON GLI AUTOMI

Ogni espressione regolare può essere trasformata in un automa finito che riconosce il linguaggio che essa descrive, e viceversa.

Teorema di Kleene:

Un linguaggio è **regolare** se e solo se qualche **espressione regolare** lo descrive: $L \leftrightarrow E.R.$

Questo teorema va dimostrato in entrambe le direzioni, dimostrandolo in due lemma separati.

Lemma 1:

Se un linguaggio è descritto da un'**espressione regolare**, allora esso è **regolare**.

Idea:

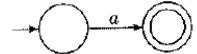
Supponiamo di avere un'espressione regolare R che descrive un linguaggio A . Trasformiamo R in un NFA che riconosce A .

Dimostrazione:

Consideriamo i sei casi nella definizione di espressione regolare.

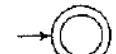
- $R = a$ per qualche $a \in \Sigma$, allora $L(R) = \{a\}$ e il seguente NFA riconosce $L(R)$.

Nota: questa macchina soddisfa la definizione di NFA ma non quella di DFA, perché ha qualche stato con nessun arco uscente per ogni possibile simbolo di input. Con NFA è più facile da descrivere.



- $R = \epsilon$, allora $L(R) = \{\epsilon\}$ e il seguente NFA riconosce $L(R)$.

- $R = \emptyset$, allora $L(R) = \emptyset$ e il seguente NFA riconosce $L(R)$.



- $R = R_1 \cup R_2$.

- $R = R_1 \circ R_2$.

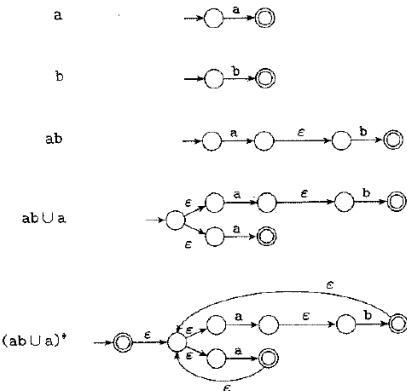
- $R = R_1^*$.



Per questi ultimi 3 casi si usano le costruzioni date nelle prove che la classe dei linguaggi regolari è chiusa rispetto alle operazioni regolari.

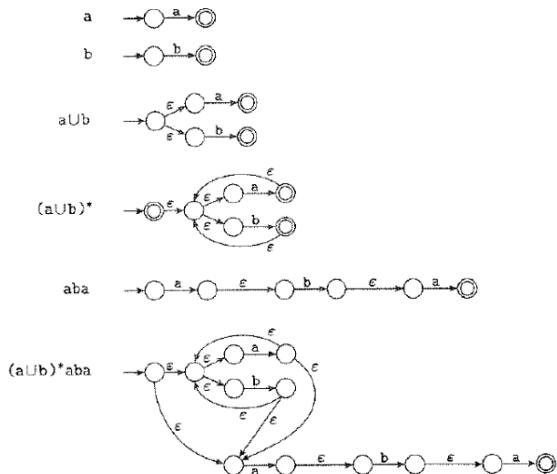
Esempio 1:

Trasformiamo l'espressione regolare $(ab \cup a)^*$ in un NFA in una sequenza di passi. Costruiamo l'automa dalle sotto-espressioni più piccole alle sotto-espressioni più grandi, finché non abbiamo un NFA per l'espressione iniziale.



Esempio 2:

Trasformiamo l'espressione regolare $(a \cup b)^*aba$ in un NFA.



Lemma 2:

Se un linguaggio è **regolare**, allora è descritto da un'**espressione regolare**.

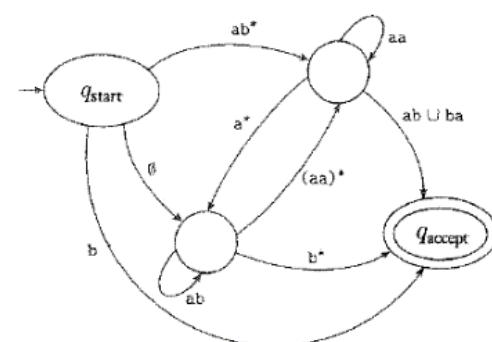
Idea:

Dobbiamo mostrare che il linguaggio A è regolare, allora un'espressione regolare lo descrive. Poiché A è regolare, esso è accettato da un DFA.

Per trasformare un DFA in una espressione regolare equivalente, si divide la procedura in due parti, usando un nuovo tipo di **automata** quello **finito non deterministico generalizzato**, **GNFA**, ovvero un NFA che può avere archi delle transizioni con espressioni regolari come etichette, invece che solo elementi dell'alfabeto o ϵ .

Per comodità i GNFA devono soddisfare le seguenti condizioni:

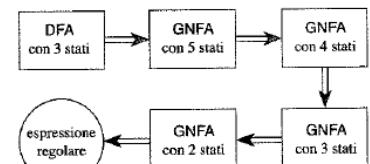
- Lo stato iniziale ha archi di transizione uscenti verso un qualsiasi altro stato ma nessun arco entrante proveniente da un qualsiasi altro stato.
- Esiste un solo stato accettante, ed esso ha archi entranti provenienti da un qualsiasi altro stato, ma nessun arco uscente verso un qualsiasi altro stato. Inoltre, lo stato accettante non è uguale allo stato iniziale.
- Eccetto che per lo stato iniziale e lo stato accettante, un arco va da ogni stato ad ogni altro stato e anche da ogni stato in sé stesso.



Possiamo facilmente trasformare un DFA in un GNFA nella forma speciale. Aggiungiamo semplicemente un nuovo stato iniziale con un ϵ -arco che entra nel vecchio stato iniziale e un nuovo stato accettante con ϵ -archi entranti, provenienti dai vecchi stati accettanti.

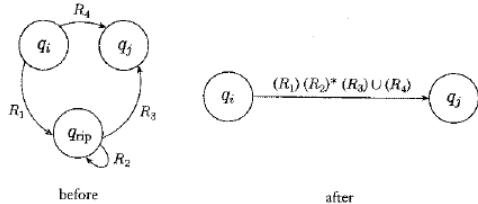
Per trasformare un **GNFA** in un'**espressione regolare**, supponiamo che GNFA abbia k stati. Poiché un GNFA deve avere uno stato iniziale e uno accettante ed essi devono essere diversi tra loro, sappiamo che k è maggiore o uguale a 2. Se k è maggiore di 2, costruiamo un GNFA equivalente con $k-1$ stati. Questo passo viene ripetuto sul nuovo GNFA fino a quando esso è ridotto a due stati, e l'etichetta dallo stato iniziale al finale rappresenta l'espressione regolare equivalente.

Il passo cruciale è costruire un GNFA equivalente con uno stato in meno quando k è maggiore di 2. Lo facciamo scegliendo uno stato, estraendo dalla macchina, e modificando il resto in modo che sia ancora riconosciuto lo stesso linguaggio. Ogni stato può essere scelto, purché non sia lo stato iniziale o quello accettante.

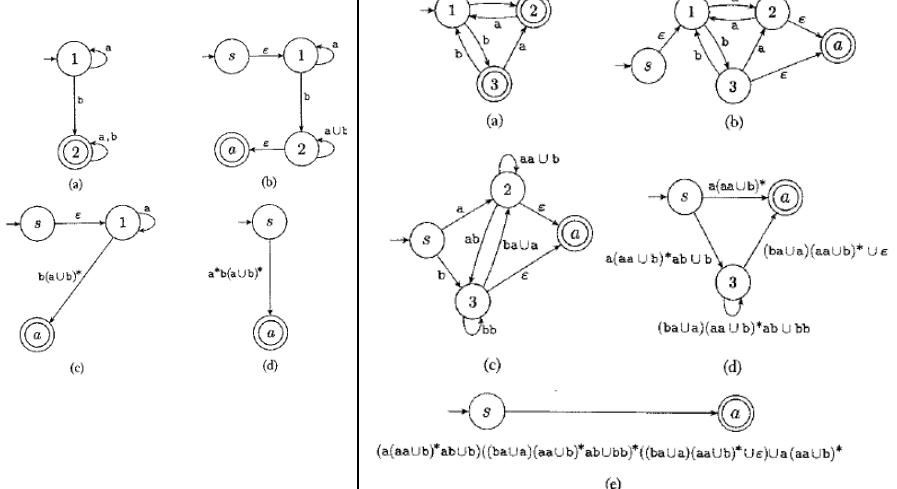


Esempio:

Assumiamo che q_{rip} sia lo stato rimosso, dopo averlo rimosso modifichiamo la macchina cambiando le espressioni regolari che etichettano ciascuno dei restanti archi. Le nuove etichette controbilanciano l'assenza di q_{rip} reintegrando le computazioni perse. La nuova etichetta che va da uno stato q_i ad uno stato q_j è una espressione regolare che descrive tutte le stringhe che porterebbero la macchina da q_i a q_j o direttamente o tramite q_{rip} .



Altri esempi:



2.3 LINGUAGGI NON REGOLARI

Esistono alcuni linguaggi che non possono essere riconosciuti da alcun automa finito. Consideriamo il linguaggio $B = \{0^n 1^n \mid n \geq 0\}$, se si cerca di trovare un DFA che riconosca B , si osserva che la macchina ha bisogno di ricordare quanti simboli uguali a 0 ha visto fin quando legge input. Poiché il numero di simboli uguali a 0 non è limitato, la macchina dovrà tenere traccia di un numero infinito di possibilità. Ma non può farlo con un numero finito di stati.

Esempio:

Il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ non è regolare, cioè non è riconosciuto da alcun automa finito: non è possibile, infatti, avere un automa che riconosca un linguaggio di questo tipo. Se tentiamo di costruire un DFA che riconosce $L = \{a^n b^n \mid n \geq 0\}$, l'automa deve verificare che la stringa è fatta da un numero n di a seguito da uno stesso numero n di b: ci servirebbero infiniti stati per contare il numero di a ed il numero di b.

2.3.1 PUMPING LEMMA

Per provare la **non regolarità** si usa la tecnica che deriva dai linguaggi regolari, chiamato **pumping lemma**. Questo teorema afferma che tutti i linguaggi regolari hanno una proprietà speciale, se si mostra che un certo linguaggio non soddisfa tale proprietà, si ha la certezza che esso non è regolare.

La proprietà afferma che tutte le stringhe nel linguaggio possono essere "replicate" se la loro lunghezza raggiunge almeno uno specifico valore speciale, chiamato la "**lunghezza del pumping**". Questo significa che ogni tale stringa contiene una parte che può essere ripetuta un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

TEOREMA Pumping Lemma:

Se A è un **linguaggio regolare**, allora esiste un numero p (**lunghezza del pumping**) tale che s è una qualsiasi stringa in A di lunghezza almeno p , allora s può essere divisa in tre parti, $s = xyz$, che soddisfa le seguenti condizioni:

1. Per ogni $i \geq 0$, $xy^i z \in A$;
2. $|y| > 0$;
3. $|xy| \leq p$.

Nota: Ricordiamo che $|s|$ rappresenta la lunghezza della stringa s , y^i indica i copie di y concatenate insieme, e y^0 è uguale a ϵ .

Dimostrazione:

Siano M un automa che riconosce L , e p il numero di stati di M .

Considerando una stringa $w = xyz$ tale che $|w| \geq p$, allora sappiamo che nella computazione esiste (almeno) uno stato ripetuto, cioè esiste (almeno) uno stato che viene visitato più di una volta (sia r il primo stato ripetuto). A questo punto, sappiamo che:

- x (la prima parte della stringa w) porta la computazione dallo stato iniziale q_1 allo stato r (cioè, $f^*(q_1, x) = r$);
- y (la seconda parte della stringa w) porta la computazione dallo stato r allo stato r (cioè, $f^*(r, y) = r$);
- z (la terza parte della stringa w) porta la computazione dallo stato r allo stato finale (cioè, $f^*(r, z) \in F$).

Verifichiamo che questa suddivisione soddisfa le tre proprietà del lemma:

- $|xy| \leq p$ (con r primo stato ripetuto) è soddisfatta, in quanto se così non fosse allora avremmo lo stato ripetuto prima;
- $|y| \geq 1$ segue dalla precedente, in quanto abbiamo bisogno di almeno un simbolo per poter ciclare su r ;
- $xy^i z$ appartiene al linguaggio L , dato che tale stringa porta dallo stato iniziale ad r , da r ad r per i volte (anche 0 volte), e infine da r a uno stato finale.

TEOREMA:

Sia L l'insieme di tutte le stringhe $0^n 1^n$ (con $n \geq 0$) su $\{0, 1\}$ aventi un uguale numero di 0 e di 1. Il linguaggio L **non è regolare**.

Dimostrazione:

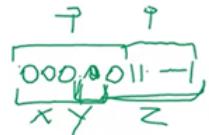
Dimostriamo questo teorema per contraddizione usando il Pumping Lemma (PL). Supponiamo, a tal scopo, che L è regolare, quindi applichiamo il lemma. Sia, quindi, p la costante del PL. Consideriamo una stringa $w = 0^p 1^p$. Il pumping lemma implica che esistono $xyz = 0^p 1^p$, tali che $|xy| \leq p$, $y \neq \epsilon$ e che $xy^k z$ appartiene a L per ogni $k \geq 0$.

Ma se la stringa presa è $0^p 1^p$, allora i primi p simboli sono tutti 0. Tuttavia, se il PL dice che la lunghezza della sottostringa xy è effettivamente minore o uguale a p , allora sappiamo che essa sta entro quei p simboli 0; inoltre, siccome $y \neq \epsilon$, abbiamo che y è una sottostringa di almeno uno 0.

Ora, consideriamo la stringa $xy^k z$. Presa in input tale stringa, per contraddirre il PL occorre esibire un valore k per cui la stringa $xy^k z$ non appartiene al linguaggio.

Consideriamo $k = 2$ (quindi, y si ripete due volte). Per appartenere al linguaggio, questa stringa dovrà avere un numero p di 0 ed un numero p di 1: abbiamo una stringa del tipo $0^{|x|}0^{|y|}0^{|y|}0 \dots 0^p1^p = 0^{(p+|y|)}1^p$; poiché $|y| > 1$, si ha che questa stringa non appartiene al linguaggio L , dato che $p + |y| > p$.

Ricapitolando, preso un qualsiasi $k > 1$, allora xy^kz e xz hanno (tra di loro) diverso numero di 0 e stesso numero di 1. Questa è una **contraddizione**.



Esempio:

Sia $L = \{a^nba^m \mid n < m\}$. Usiamo il Pumping Lemma per dimostrare che il linguaggio L non è regolare.

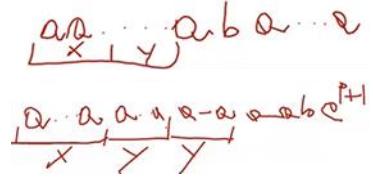
Supponiamo P.A. che L sia regolare; di conseguenza, esso soddisfarebbe il PL, quindi esisterebbe una costante p tale che, per ogni stringa w in L di lunghezza $|w| \geq p$, esistono stringhe x, y, z tali che $w = xyz$, che soddisfano: $|y| > 0, |xy| \leq p, xy^i z$ in L (per ogni $i \geq 0$).

Sia definita la stringa, legata a p , $w = a^pba^{p+1} = xyz$, dove $|xy| \leq p$; da questo segue che xy è costituita da un numero finito di a .

Bisogna dimostrare che la terza condizione non è soddisfatta, cioè che esiste $i \geq 0$ tale che la stringa $xy^i z \notin L$.

Sia, ad esempio, $i = 2$, quindi xy^2z è la stringa a lato.

Notiamo che questa stringa non appartiene al linguaggio, dato che $(|x| + |y| + \# a \text{ prima del simbolo } b) = p$, e quindi il numero totale di a prima del simbolo b è $p + |y|$; siccome $|y| > 1$, abbiamo che $p + |y| \geq p + 1$, quindi la condizione $n < m$ del linguaggio non è soddisfatta. Dal Pumping Lemma, otteniamo che il linguaggio L non è regolare. Questa è una contraddizione.



3. MACCHINE DI TURING

La **Macchina di Turing** è simile ad un **automa finito**, ma con una memoria illimitata e senza restrizioni. Tuttavia, anche essa non è in grado di risolvere alcuni problemi.

Funzionamento della macchina:

Utilizza un nastro infinito come propria memoria illimitata. Ha una testina che è in grado di leggere e scrivere simboli ed è libera di muoversi lungo il nastro. Inizialmente il nastro contiene solo la stringa di input e tutto il resto è vuoto. Se la macchina deve memorizzare una informazione la può scrivere sul nastro. Per leggere l'informazione che ha scritto la macchina può spostare la testina su di essa.

La macchina continua a computare finché non decide di produrre un output. Gli output **accetta** e **rifiuta** sono ottenuti occupando appositi **stati di accettazione** e **rifiuto**. Se non raggiunge uno di questi stati la macchina andrà avanti per sempre.

Differenza tra un automa finito e macchina di Turing:

1. Una macchina di Turing può sia scrivere che leggere sul nastro;
2. La testina di lettura-scrittura può muoversi sia verso sinistra che verso destra;
3. Il nastro è infinito;
4. Gli stati speciali di accettazione e rifiuto sono immediati.

Definizione formale di macchine di Turing:

Una **macchina di Turing** è una 7-tupla $(Q, \Sigma, \Gamma, f, q_0, q_{\text{accept}}, q_{\text{reject}})$, dove Q, Σ, Γ sono tutti insiemi finiti e:

1. Q è l'**insieme degli stati**;
2. Σ è l'**alfabeto di input** non contenente il **simbolo blank** '_';
3. Γ è l'**alfabeto del nastro** con $_ \in \Gamma$ e $\Sigma \subseteq \Gamma$ contenente tutti i simboli che possono essere scritti all'interno di una cella di memoria (teoricamente, l'unione tra l'alfabeto di lavoro e l'insieme dei simboli non processabili dalla macchina, come '_);
4. $f: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ è la **funzione di transizione**, dove le lettere lette sono all'interno del nastro (in ogni istante si hanno dei simboli nelle varie celle, e la cella a cui punta la testina rappresenta lo stato q in cui si trova la macchina).;
5. $q_0 \in Q$ è lo **stato iniziale**;
6. $q_{\text{accept}} \in Q$ è lo **stato di accettazione**;
7. $q_{\text{reject}} \in Q$ è lo **stato di rifiuto**, con $q_{\text{reject}} \neq q_{\text{accept}}$.

Nota: q_{accept} e q_{reject} sono stati di arresto.

Sia M una Macchina di Turing definita da $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$; allora:

- ad ogni istante, M occupa uno degli stati in Q ;
- la testina si trova in un quadrato del nastro contenente un qualche simbolo $y \in \Gamma$;
- la funzione di transizione $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ dipende dallo stato q e dal simbolo di nastro y .

Il **range** della funzione di transizione sono triple (q', y', d) , con:

- $q' \in Q$;
- $y' \in \Gamma$ è il simbolo scritto dalla testina sulla cella del nastro su cui la testina si trova *all'inizio* della transizione;
- $d \in \{L, R\}$ è la direzione in cui la testina muove un passo, con $L = \text{left}$ ed $R = \text{right}$.

Esempio1:

Dato l'esempio posto a destra, si ha che i passi 1, 2, 3, per passare dal tempo 0 al tempo 1, sono rappresentati da una funzione di transizione $\delta(q, a) = (r, k, L)$, con q ed r stati generici per supposizione.

Esempio2:

Dato l'esempio posto a destra, se la testina puntasse alla seconda cella, scrivesse f e si muovesse a destra, allora avremmo $\delta(r, b) = (r', f, R)$, con r ed r' stati generici per supposizione.

La computazione parte sempre da uno stato iniziale q_0 , con l'input scritto sul nastro: quest'ultimo inizialmente contiene l'input, ed in particolare l'inizio dell'input è scritto all'inizio del nastro. La testina si trova nella prima cella a sinistra del nastro (cella 0). A questo punto, la macchina è pronta per l'esecuzione, ed effettuerà le transizioni in accordo alla funzione δ .

Esempio1:

Con questo esempio, vediamo come la funzione δ può essere rappresentata in modo compatto all'interno del diagramma degli stati.

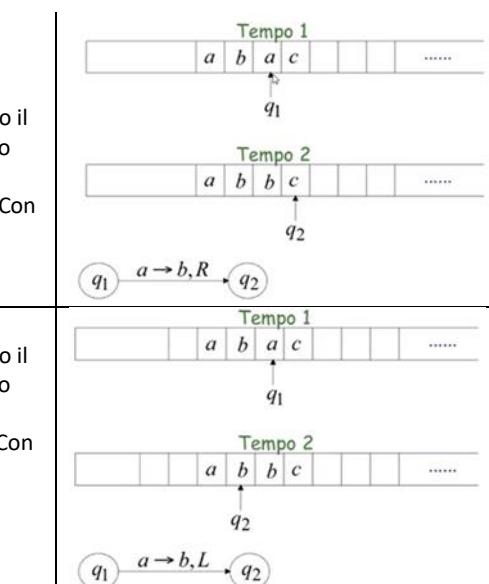
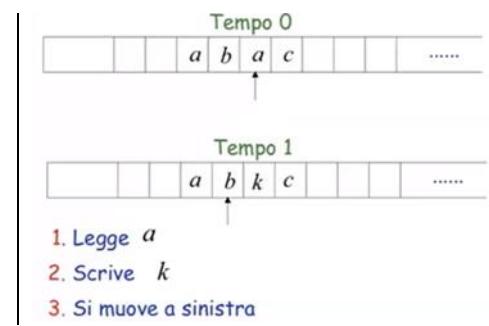
La figura a destra dice che la testina, al tempo 1, si trova nello stato q_1 e sulla cella 2, dove è scritto il simbolo 'a'; al tempo 2, la testina passa allo stato q_2 e si trova sulla cella 3, dove è scritto il simbolo 'c', mentre nella cella dove essa puntava al tempo 1 è stato scritto il simbolo 'b'.

Ciò è sintetizzabile secondo la transizione in basso nella figura, che equivale a $\delta(q_1, a) = (q_2, b, R)$. Con $a \rightarrow b$ indichiamo che la nuova lettera scritta sarà b , ci si muove verso R e si passa da q_1 a q_2 .

Esempio2:

La figura a destra dice che la testina, al tempo 1, si trova nello stato q_1 e sulla cella 2, dove è scritto il simbolo 'a'; al tempo 2, la testina passa allo stato q_2 e si trova sulla cella 1, dove è scritto il simbolo 'b', mentre nella cella dove essa puntava al tempo 1 è stato scritto il simbolo 'b'.

Ciò è sintetizzabile secondo la transizione in basso nella figura, che equivale a $\delta(q_1, a) = (q_2, b, L)$. Con $a \rightarrow b$ indichiamo che la nuova lettera scritta sarà b , ci si muove verso L e si passa da q_1 a q_2 .



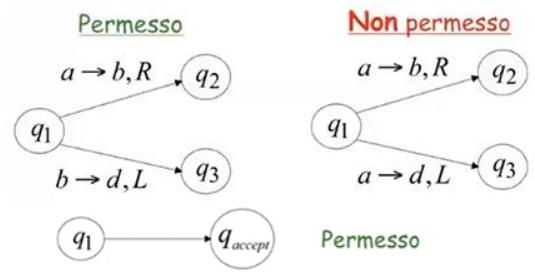
La computazione termina quando M raggiunge: uno stato accettazione q_{accept} (che implica una **Computazione Accept**), o uno stato rifiuto q_{reject} (che implica una **Computazione Reject**).

Macchina di Turing Deterministica:

Parliamo di Macchina di Turing deterministica, in cui *non ci sono ε-transition*: nella funzione di transizione, *per ogni stato e per ogni lettera esiste uno ed un solo risultato*. Nella figura a destra, notiamo che non è permesso che, nello stato q_1 , leggendo a si possa scrivere b o d e muoversi o verso R o verso L ; ciò avviene in quanto abbiamo due possibili mosse per $\delta(q_1, a)$.

Gli **stati di arresto** non hanno archi uscenti: per definizione, in uno stato di arresto (accept o reject) la computazione termina.

Una macchina di Turing può accettare, non accettare o avere problemi con una stringa



Esempio1:

L'esempio di cui a destra mostra un caso in cui la stringa non viene accettata.

L'alfabeto di lavoro è $\Sigma = \{a, b\}$, mentre l'alfabeto del nastro è $\Gamma = \{a, b, _\}$.

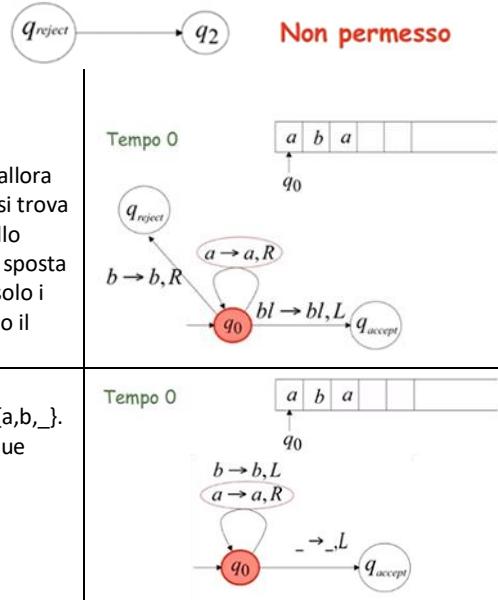
Possiamo denotare il blank sia con $_$ che con bl . Lo stato iniziale è q_0 : se si trova in q_0 e legge ' a ', allora non si cambia il contenuto della cella e la testina si sposta a R , e non si ha un cambio di stato; se si trova in q_0 e legge ' b ', allora non si cambia il contenuto della cella e la testina si sposta a R , passando allo stato q_{reject} ; se si trova in q_0 e legge ' bl ', allora non si cambia il contenuto della cella e la testina si sposta a L , passando allo stato q_{accept} . Si noti che la stringa, in questo esempio, viene rifiutata leggendo solo i primi due simboli, per cui non è necessario arrivare a fine stringa per determinare l'accettazione o il rifiuto. Se avessimo avuto in input la stringa $aaa_\!$, allora la stringa sarebbe stata accettata.

Esempio2:

Nell'esempio di cui a destra, l'alfabeto di lavoro è $\Sigma = \{a, b\}$, mentre l'alfabeto del nastro è $\Gamma = \{a, b, _\}$.

Lo stato q_{accept} non è raggiungibile, in quanto si entra in un loop in cui si leggono sempre i primi due simboli. La computazione non termina, di conseguenza l'input non viene accettato.

Ciò mostra che la macchina di Turing può andare in loop: questo è uno dei grandi problemi che evidenzia che esistono problemi non risolvibili con un calcolatore.



Esiste un'**eccezione** nella funzione di transizione, secondo cui se ci troviamo all'inizio del nastro (prima cella) e leggiamo un simbolo qualsiasi per poi muoverci a sinistra, il risultato sarà sempre che la testina punterà alla prima cella. Nell'esempio precedente, se avessimo avuto in input la stringa $bb_\!$ avremmo ottenuto che la funzione $\delta(q_0, b) = (q_0, b, L)$ non avrebbe spostato la testina, che quindi punterà sempre alla cella corrente (e, in questo esempio, entrerebbe comunque in un loop). Questo non è considerato un errore.

Esempio1:

Strategia per accettare $\{a^n b^n \mid n \geq 0\}$. Si consideri l'alfabeto $\Sigma = \{a^n b^n \mid n \geq 0\}$. Possiamo *cancellare ripetutamente* la prima occorrenza di a e l'ultima occorrenza di b : se la stringa appartiene all'alfabeto Σ , allora non rimangono simboli. Tutto ciò lo possiamo formulare in cinque passi:

1. Se leggi $_$, vai al punto 5. Se leggi a , scrivi $_$ e vai al punto 2;
2. Spostati a destra (R) di tutti a e b . Al primo $_$, muovi a sinistra (L) e vai al punto 3;
3. Se leggi b , scrivi $_$ e vai al punto 4;
4. Spostati a sinistra (L) di tutti a e b . Leggendo $_$, muovi R e vai al punto 1;
5. Accept.

La figura a destra mostra una macchina di Turing che implementa tale algoritmo.

Lo stato $reject$ manca: spesso, questo non viene spesso considerato per non avere un numero elevato di stati. Prima di costruire la MdT, si può assumere che *tutte le transizioni che non compaiono vanno implicitamente in uno stato, non inserito nel diagramma, di rifiuto*. Ad esempio, notiamo che se siamo nello stato 3 e leggiamo a , allora non c'è una transizione: in questo caso, si assume che la stringa viene rifiutata.

Esempio2:

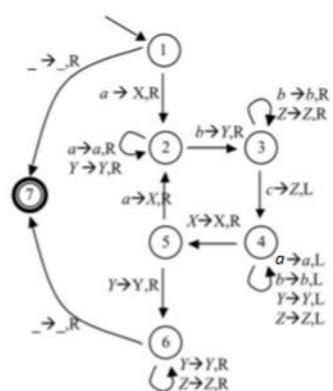
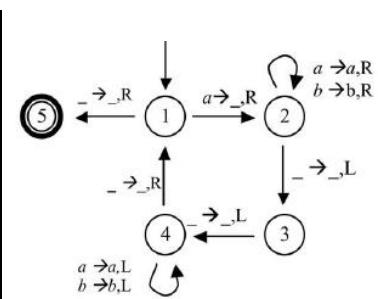
La seguente è una MdT per stringhe $a^n b^n c^n$. Dunque, operiamo sull'alfabeto $\Sigma = \{a, b, c\}$. Inoltre, l'alfabeto del nastro è $\Gamma = \{_, a, b, c, X, Y, Z\}$.

Il "cammino" computazionale di tale macchina può riassumersi in 4 passi:

- 1) cancella la prima a . In questo esempio, sovrascriviamo le a con una X ;
- 2) avanza (scorrendo la stringa), e cerca la prima occorrenza di b . Quando viene trovata la b , la sovrascriviamo con una Y ;
- 3) avanza (scorrendo la stringa), e cerca la prima occorrenza di c . Quando viene trovata la c , la sovrascriviamo con una Z ;
- 4) ritorna al passo 1).

Ad esempio, si riceve in input la stringa $aabbcc$. Al primo passo, si sostituisce la prima a con una X , e ci si sposta a destra; successivamente, si avanza fin quando non si trova una b (in questo caso, soltanto una volta). Trovata la b , la si sovrascrive con Y e ci si sposta a destra; successivamente, si avanza fin quando non si trova una c (in questo caso, soltanto una volta). Trovata la c , la si sovrascrive con Z e ci si sposta a sinistra. A questo punto, l'iterazione è finita e la testina può iniziare a spostarsi a sinistra per tornare all'inizio del nastro e ad effettuare una nuova iterazione (il passo 4 serve proprio a tornare indietro sul nastro, per qualsiasi simbolo letto a parte una X , che inizia a far muovere la testina verso destra).

Anche in questo esempio, ogni transizione da uno stato ad un altro non inserita, se si verifica porta la macchina in uno stato q_{reject} che rifiuta la stringa.



3.1 CONFIGURAZIONI MACCHINA DI TURING

Una **configurazione** di una Macchina di Turing è una descrizione concisa di stato e contenuto del nastro. Trattasi di una stringa $C = uqv$, dove:

- q è lo *stato* occupato dalla macchina M;
- uv è il *contenuto del nastro* (sinistra – destra);
- la testina punta sul primo (cioè, più a sinistra) simbolo di v (su primo blank _ se $v = \epsilon$);
- dopo v sono presenti solo simboli blank _.

Esempio1:

Dato l'esempio precedente (MdT per stringhe $a^n b^n c^n$), al tempo in cui il nastro contiene la stringa XXYYZZ e punta alla seconda cella (cella 1, contenente l'ultimo X), la configurazione di MdT in questo tempo è

$C = X4XXYYZZ$, dove $u = X$, $q = 4$, $v = XYZZ$.

A partire da questa configurazione C, sapendo che $\delta(4, X) = (5, X, R)$, possiamo ottenere la configurazione successiva $C' = XX5YYZZ$, dove $u = XX$, $q = 5$, $v = YYZZ$.

In generale, la configurazione cambia ad ogni mossa. Si dice che C_1 produce C_2 (si indica con $C_1 \rightarrow C_2$) se una mossa della MdT può far andare la macchina da C_1 a C_2 .

Esempio2:

Se $a, b, c \in \Gamma$ (cioè, sono simboli), $u, v \in \Gamma^*$ (cioè, sono stringhe), $q_i, q_j \in Q$ (cioè, sono stati),
allora $uaq,bv \rightarrow uq,acv$ se $\delta(q_i, b) = (q_j, c, L)$ è una *mossa a sinistra*,
oppure $uaq,bv \rightarrow uacq,jv$ se $\delta(q_i, b) = (q_j, c, R)$ è una *mossa a destra*.

Casi particolari di configurazioni:

Data una configurazione $q,bv \rightarrow q,cv$, se la testina è ad inizio nastro e la prossima è una *mossa a sinistra*, allora la posizione della testina non cambia (senza che la Macchina di Turing se ne accorga).

La configurazione uaq è equivalente a $uaq,$, cioè la parte vuota del nastro viene riempita con simboli blank (_).

Una configurazione di M si dice **di start** su un input w ($q_0 w$) se e solo se lo stato q_0 è lo stato iniziale, il nastro contiene w e la testina è posizionata sulla prima cella del nastro.

Una configurazione di M si dice **di accettazione** (Accept) se essa raggiunge uno stato q_{accept} .

Una configurazione di M si dice **di rifiuto** (Reject) se essa raggiunge uno stato q_{reject} .

Una configurazione di M si dice **di Halt** se essa raggiunge o uno stato q_{reject} o uno stato q_{accept} (cioè, una qualsiasi configurazione Accept o Reject).

Una Macchina di Turing M **accetta** una parola w se esiste una computazione (*sequenza di configurazioni*) di M del tipo C_1, C_2, \dots, C_k tale che:

1. $C_1 = q_0 w$ è la configurazione iniziale di M con input w ;
2. $C_i \rightarrow C_{i+1}$ per ogni $i = 1, 2, \dots, k - 1$;
3. C_k è la configurazione di accettazione (Accept) di M.

Una Macchina di Turing M **rifiuta** una parola w se esiste una computazione (*sequenza di configurazioni*) di M del tipo C_1, C_2, \dots, C_k tale che:

1. $C_1 = q_0 w$ è la configurazione iniziale di M con input w ;
2. $C_i \rightarrow C_{i+1}$ per ogni $i = 1, 2, \dots, k - 1$;
3. C_k è la configurazione di rifiuto (Reject) di M.

Una Macchina di Turing M può raggiungere tre *risultati computazionali*:

1. M **accetta** – se si ferma in q_{accept} ;
2. M **rifiuta** – se si ferma in q_{reject} ;
3. M **cicla/loop** – se non si ferma mai.

Mentre M funziona, non si può dire se essa è in loop, in quanto si potrebbe fermare in seguito oppure no.

Una MdT M accetta una stringa w se esiste una computazione (sequenza di configurazione) di M: C_1, \dots, C_k tale che

1. $C_1 = q_0 w$ è la configurazione iniziale di M con input w ;
2. Ogni C_i produce C_{i+1} per ogni $i=1, \dots, k-1$;
3. C_k è una configurazione di accept.

Data una Macchina di Turing M, il **linguaggio di M** è l'insieme delle stringhe che M accetta, e viene denotato con $L(M)$.

Un linguaggio si dice Turing-riconoscibile se esiste una macchina di Turing che lo riconosce.

Formalmente, un linguaggio L si dice Turing riconoscibile se esiste una Macchina di Turing M tale che $L(M) = L$. Di conseguenza, la macchina accetta tutte le stringhe del linguaggio. Se, invece, una stringa $w \notin L$, allora la stringa può essere o *rifiutata* o mandare la macchina *in loop*.

Abbiamo visto che è difficile dire se una MdT è in loop. Possiamo evitare questo problema costruendo le macchine che si fermano (accettando o rifiutando) *su ogni input*: trattasi dei **deciders**.

Si dice che un decider **decide** il linguaggio L se esso riconosce L.

Un linguaggio si dice **Turing decidibile** se esiste una MdT che lo decide. Formalmente, un linguaggio L si dice Turing decidibile se esiste una Macchina di Turing M tale che $L(M) = L$ e M è un decider.

La differenza tra un linguaggio L Turing riconoscibile e Turing decidibile risiede nel fatto che i primi possono mandare le MdT che li riconoscono in loop, mentre i secondi possono far sì che le MdT che li decidono o accettino o rifiutino le loro stringhe *su ogni input*.

Un linguaggio si dice Turing-decidibile o semplicemente decidibile se esiste una macchina di Turing che lo decide.

Esempio1:

Consideriamo il linguaggio $L = \{0^{2^n} \mid n > 0\}$ dell'insieme di stringhe di 0 la cui lunghezza è potenza di 2. Vogliamo costruire una MdT M_2 che lo decide.

Nota. Il linguaggio non è regolare (ciò è dimostrabile mediante Pumping Lemma).

→ Si noti che n è potenza di 2 se e solo se ripetute divisioni per 2 danno resto 1.

Sia w l'input; allora, un algoritmo tale che M_2 decida tale linguaggio può essere il seguente:

1. Scorrere il nastro da sinistra a destra cancellando ogni SECONDO 0; //divide per 2 il numero di 0
2. Se rimane solo uno 0, allora ACCEPT;

3. Se rimane un numero di 0 dispari ≥ 3 , allora REJECT; //in quanto il numero di 0 non è potenza di 2

4. Se rimane un numero pari di 0, allora riporta la testina all'inizio del nastro;

5. Ritorna al passo 1.

Allora, l'implementazione della MdT M_2 può essere la seguente.

Si noti che spesso si utilizza la forma contratta $x \rightarrow R$, che equivale a $x \rightarrow x, R$ (cioè, quando si sovrascrive lo stesso carattere letto).

L'alfabeto della macchina è $\Sigma = \{0\}$. Inoltre, l'alfabeto del nastro è $\Gamma = \{0, x, _\}$.

Ad esempio, sia $w = 0000$. Quando ci si trova nello stato iniziale q_1 , con un passo tecnico si sostituisce (il primo) 0 con $_$ per poi passare alla cella successiva (ed allo stato q_2). Passati allo stato q_2 , si sostituisce (il secondo) 0 con x e si passa alla cella successiva (ed allo stato q_3). Nello stato q_3 , si lascia (il terzo) 0 invariato e si passa alla cella successiva (ed allo stato q_4). Nello stato q_4 , si sostituisce (il quarto) 0 con x e si passa alla cella successiva (ed allo stato q_5). Ora, la testina punta su $_$, quindi abbiamo scandito tutta la stringa in input; dobbiamo iterare di nuovo: si sposta la testina all'indietro (passando allo stato q_5 , dove

per ogni simbolo letto si sposta la testina a sinistra, fino ad arrivare all'inizio) fin quando si riconosce il simbolo $_$ (questo è il motivo per cui abbiamo utilizzato un carattere diverso, per simboleggiare l'inizio della stringa), tornando allo stato q_2 : qui siamo pronti ad una nuova iterazione.

La testina si sposta fin quando non trova uno 0 (in questo caso, il terzo), che viene sostituito con x e si muove la testina a destra (e si passa allo stato q_3). Nello stato q_3 , con le x si avanza: si arriva alla fine del nastro, quindi si legge il simbolo $_$ e si torna in q_5 , dove si torna all'inizio della stringa e si passa quindi a q_2 (muovendo la testina a destra, quindi al secondo simbolo della stringa, cioè la prima x); tornati in q_2 , infine, si scandisce ogni simbolo x sul nastro e si arriva alla fine della stringa: si legge, quindi, $_$ e si passa allo stato q_{accept} , e quindi la stringa 0000 viene accettata.

Esempio 2:

Consideriamo il linguaggio $L = \{w\#w \mid w \in \{0, 1\}^*\}$. Di seguito un'idea per verificare se una stringa $s \in L$:

1. Leggi il primo carattere;
2. Memorizzalo e cancellalo;
3. Cerca $\#$ e guarda il carattere successivo;
4. Se esso è uguale al carattere memorizzato, allora cancellalo;
5. Ritorna all'inizio al primo carattere non cancellato;
6. Ripeti 1-5 fino a considerare tutta la stringa in input: se si trova qualcosa di "inatteso", allora REJECT; altrimenti, ACCEPT.

Vediamo, ora, come costruire una MdT M_1 per il linguaggio L . Sia definita in input una stringa w ; allora, M_1 deve:

1. Memorizzare il simbolo più a sinistra e cancellarlo (scriviamo x);
2. Avanzare sul nastro fino a superare $\#$, se non si trova allora REJECT;
3. Confronta il primo simbolo diverso da x con il simbolo memorizzato, se è diverso allora REJECT;
4. Se è uguale, cancella il simbolo confrontato e ritorna a inizio nastro;
5. Vai al punto 1.

Vediamo, ora, la funzione di transizione per M_1 . Si ha che $\Sigma = \{0, 1, \#\}$, e $\Gamma = \{0, 1, \#, x, _\}$. Nella descrizione, lo stato q_{reject} e tutte le transizioni in ingresso sono state omesse. Ovunque vi sia una transizione mancante, va in q_{reject} .

Nota. Abbiamo utilizzato il termine "memorizzare": le MdT non possono memorizzare simboli in quanto non hanno memoria, ma può essere ottenuta una memorizzazione progettando il diagramma degli stati in un modo preciso: in particolare, si useranno stati diversi a seconda che l'input sia un 1 o uno 0. A tal scopo, gli stati q_2 e q_3 memorizzano il bit 0, mentre gli stati q_4 e q_5 memorizzano il bit 1.

In altre parole, questi due segmenti sono identici, e in ogni segmento si utilizza il valore memorizzato.

Esempio del funzionamento alle slide 116 – 148.

Le Macchine di Turing possono avere le proprie funzioni di transizione rappresentate utilizzando una **tavola**. Data una funzione di transizione generica $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, è possibile costruire una tabella di 5 colonne ed n righe (le n righe rappresentano le etichette complessive della funzione), dove:

- la prima colonna rappresenta lo stato attuale ($q_i \in Q$);
- la seconda colonna rappresenta il simbolo letto ($s \in \Gamma$);
- la terza colonna rappresenta il nuovo stato ($q_j \in Q$);
- la quarta colonna rappresenta il simbolo da scrivere ($t \in \Gamma$);
- la quinta colonna rappresenta il movimento della testina ($\{L, R\}$).

Ad esempio, data la tabella a lato otteniamo che:

- la riga 4 rappresenta la funzione $\delta(q_1, 0) = (q_1, 0, L)$;
- la riga 6 rappresenta la funzione $\delta(q_1, _) = (q_2, _, R)$;

- ...

3.2 CALCOLO FUNZIONI CON MACCHINA DI TURING

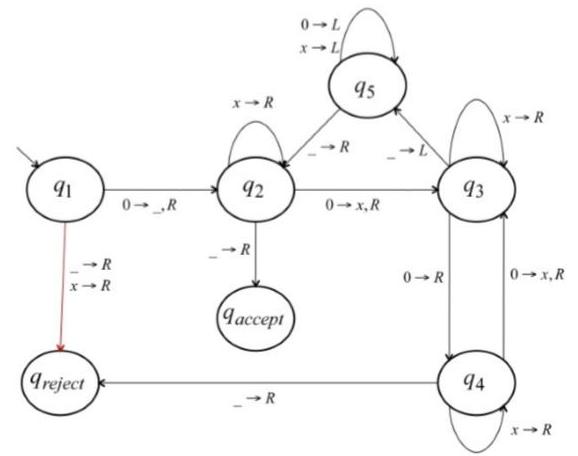
Sia definita una funzione $f: D \rightarrow S$ che opera su stringhe; cioè, data una stringa $w \in D$ si ottiene un'altra stringa $f(w) \in S$.

Una funzione può dipendere da più variabili. Ad esempio, la funzione addizione $f(x, y) = x + y$ è una funzione a due variabili.

Consideriamo come l'insieme degli interi. Se usiamo la notazione decimale e vogliamo dare in output 5, allora il simbolo della MdT sarà 5 (con alfabeto $\Sigma = \{0, 1, \dots, 9\}$). Se usiamo la notazione binaria e vogliamo dare in output 5, allora il simbolo della MdT sarà 101 (con alfabeto $\Sigma = \{0, 1\}$). Noi useremo la **notazione unaria**, avente come alfabeto $\Sigma = \{1\}$, che dà in output tanti 1 quant'è il valore del numero in input (es. $5 \rightarrow 11111$): questa notazione è più semplice, anche se non è molto efficace.

Idea:

Una funzione f si dice **calcolabile** se esiste una macchina di Turing M tale che, per ogni $w \in D$, essa parte con un input w in uno stato iniziale q_0 e termina in uno stato finale q_{accept} avendo sul nastro il valore $f(w)$.



stato	simbolo	stato	simbolo	movimento
q0	0	q0	1	R
q0	1	q0	0	R
q0	_	q1	_	L
q1	0	q1	0	L
q1	1	q1	1	L
q1	_	q2	_	R



Esempio1:

La funzione $f(x, y) = x + y$ è calcolabile. Infatti, dati x, y interi, allora possiamo costruire una Macchina di Turing che calcola questa funzione: ad una stringa in input $x0y$ (unario) è associata una stringa in output $xy0$ (unario). Lo 0 viene usato all'inizio semplicemente per separare le due stringhe, mentre viene usato al termine per eseguire eventualmente altre operazioni.

Ad esempio, dato l'input $2 + 2$ (11011) ricaviamo l'output 4 (1111).

A destra è posta la MdT che esegue l'operazione appena vista.

È importante che, al termine del calcolo dell'output, la testina che si trova alla fine torni all'inizio del nastro per andare nella situazione finale (q_{accept}).

Esempio del funzionamento alle slide 11 – 24.

Esempio2:

La funzione $f(x) = 2x$ è calcolabile. Infatti, dato un numero x intero, allora possiamo costruire una Macchina di Turing che calcola questa funzione: ad una stringa in input x (unario) è associata una stringa in output xx (unario).

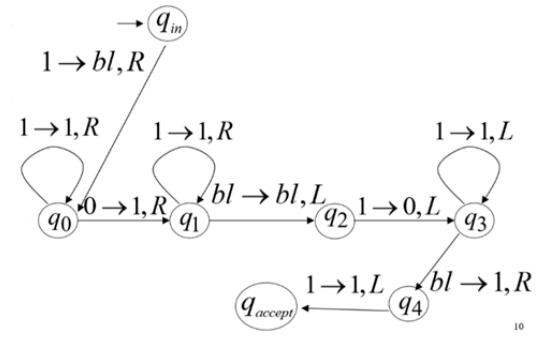
Si scriva, per esercizio, la Macchina di Turing corrispondente.

Esempio3:

La funzione $f(x, y) = \begin{cases} 1 & \text{se } x > y \\ 0 & \text{se } x \leq y \end{cases}$ è calcolabile. La MdT, ad esempio, parte con un input 111011 e dà in output 1 (in quanto $111 = 3 > 2 = 11$).

Dunque, bisogna vedere se il numero di 1 prima dello 0 è maggiore rispetto al numero di 1 dopo lo 0.

Si scriva, per esercizio, la Macchina di Turing corrispondente.



4. VARIANTI DI MACCHINE DI TURING

Esistono definizioni alternative di Macchina di Turing, chiamate *varianti*. Tra queste, vedremo delle MdT a più nastri e MdT non deterministiche.

Mostreremo, inoltre, che tutte le varianti "ragionevoli" hanno la stessa capacità computazionale.

Quando proviamo che una MdT ha una certa proprietà, non ci poniamo domande sulla grandezza di tale macchina o su quanto è complesso programmarla. Per il momento, ci interessano solo alcune proprietà essenziali.

Uno **stayer** è una Macchina di Turing la cui testina può rimanere sulla stessa cella del nastro durante una transizione; formalmente, essa è la stessa settupla $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, con la sola differenza che la funzione di transizione è $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, S, R\}$, dove **S** indica che la testina resta ferma durante la transizione (**S** sta per "stay").

Diciamo che il potere computazionale di due modelli è lo stesso se riconoscono la stessa classe di linguaggi.

Una MdT può essere facilmente simulata da uno stayer (basta banalmente non usare la possibilità di stare nella funzione di transizione), per cui lo stayer include, per definizione, una MdT semplice. Da ciò consegue che il potere computazionale di uno stayer è almeno pari al potere computazionale di una MdT convenzionale. Resta da provare il seguente.

Corollario:

Il potere computazionale di una MdT è almeno pari al potere computazionale di uno stayer \Leftrightarrow per ogni stayer esiste una MdT che riconosce lo stesso linguaggio.

Dimostrazione:

Assumiamo che M sia uno **stayer**, e mostriamo un MdT M' che simula M .

M' è definita esattamente come M , tranne che ogni transizione **S** è sostituita da due transizioni in M' :

1. la prima porta M' in uno stato speciale (addizionale) e muove la testina a destra (**R**);
2. la seconda ritorna allo stato originale muovendo la testina a sinistra (**L**).

Quindi, M e M' riconoscono lo stesso linguaggio.

In generale, quando vogliamo provare che due varianti di MdT hanno lo stesso potere computazionale, mostriamo che possiamo simulare una con l'altra.

In alcune occasioni avremo bisogno di **memorizzare** l'informazione letta sul nastro: *possiamo usare gli stati per memorizzare informazioni*. L'approccio è simile a quello visto nell'esempio sulle MdT convenzionali, dove per memorizzare il bit 0 andavamo in una parte della macchina, mentre per memorizzare il bit 1 andavamo in un'altra parte della macchina.

Esempio:

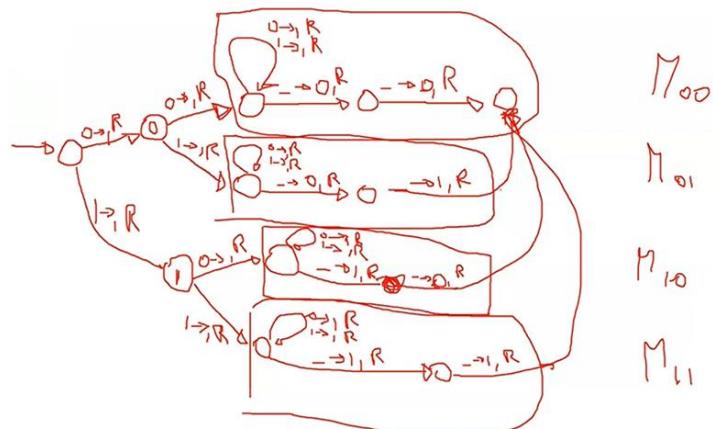
Supponiamo di avere una MdT M che deve leggere i primi due bit letti sul nastro e scriverli alla fine dell'input (al posto dei due **_** più a sinistra).

Un'idea può essere la seguente:

- costruiamo una MdT M_{00} che legge i primi due bit, cerca la fine dell'input e scrive 00 alla fine dell'input;
- replichiamo il passo precedente per tutte le possibili coppie (in questo caso M_{01} , M_{10} e M_{11});
- M : dopo aver letto i primi due bit in input, si sposta sulla replica corrispondente ai due bit letti (e quindi li scrive alla fine dell'input).

Notiamo che in base ai primi due bit in input si va in una determinata "sottomacchina".

Poiché i primi due bit, sull'alfabeto $\{0, 1\}$, possono dare 2^2 combinazioni diverse, si costruiscono 4 diverse parti di macchina che operano in dipendenza dei bit letti.



Nota: Questa tecnica è utilizzabile *in generale*.

Se vogliamo che M memorizzi una sequenza di k simboli, possiamo:

- avere una replica per ogni possibile sequenza di k simboli;
- M si sposta sulla replica che corrisponde alla sequenza mentre la legge.

Questo implica che sono necessari circa $|\Sigma|^k$ stati aggiuntivi. Infatti, se $|\Sigma|^k = c$, allora avremo c (stati dopo aver letto il primo simbolo) + c^2 (per ogni stato precedente, ci sono altri c stati) + ... + $c^k = c + c^2 + \dots + c^k = O(c^k)$.

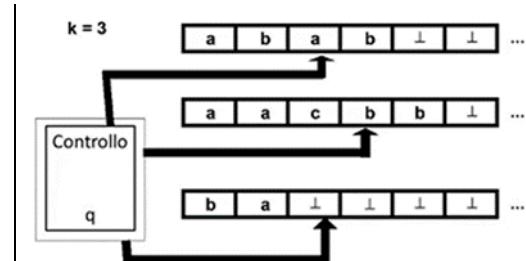
4.1 MACCHINA DI TURING MULTI-NASTRO

Una MdT **multinastro** (a k nastri) è una normale Macchina di Turing avente k nastri. Inizialmente, l'input compare sul primo nastro e gli altri sono vuoti. La sua funzione di transizione è $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, S, R\}^k$; dunque, essa muove (in modo indipendente) le testine dei vari nastri.

Questa variante è simile ad una MdT convenzionale: usa k nastri, con $k \geq 1$. Ciascun nastro ha una propria testina, e con una mossa si specificano (oltre al nuovo stato): i k simboli letti, i k simboli da scrivere e i k movimenti delle k testine. Come configurazione iniziale, essa avrà l'input sul primo nastro ed i rimanenti nastri saranno vuoti.

Nella figura a destra, ad esempio, abbiamo una MdT a 3 nastri, dove in quel momento lo stato occupato dalla macchina è q , la prima testina punta alla cella 2 (**a**), la seconda testina punta alla cella 3 (**b**), la terza testina punta alla cella 2 (**_**). In questo caso, quindi, la funzione di transizione $\delta(q, (a, b, _))$ ci restituirà un nuovo stato, tre simboli che saranno scritti in quelle celle, e i movimenti delle tre testine.

In maniera compatta, usiamo l'espressione $\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, D_1, \dots, D_k)$ per indicare che se la MdT a k nastri M si trova nello stato q_i , e la testina i legge a_i (per $i = 1, \dots, k$), allora M va nello stato q_j , la testina scrive b_i e si muove nella direzione $D_i \in \{L, S, R\}$ (per $i = 1, \dots, k$).



Esempio:

Premessa: spesso, si userà il simbolo \square per identificare il blank ($_$).

Questa macchina MdT a 2 nastri per 0^n1^n , la cui funzione di transizione è descritta dalla tabella a lato:

1. Scorre il primo nastro verso destra fino al primo 1: per ogni 0, scrive un 1 sul secondo nastro;
2. Scorre il primo nastro verso destra e il secondo nastro verso sinistra: se i simboli letti sono diversi, allora termina in uno stato non finale;
3. Se legge $_$ su entrambi i nastri, allora termina in uno stato finale.

stato	simboli	stato	simboli	movimenti
q0	(0, \perp)	q0	(\perp , 1)	(R, R)
q0	(1, \perp)	q1	(1, \perp)	(S, L)
q1	(1, 1)	q1	(\perp , \perp)	(R, L)
q1	(\perp , \perp)	q2	(\perp , \perp)	(S, S)

Nota: Le MdT multinastro sembrerebbero più potenti delle MdT ordinarie. In realtà, possiamo dimostrare che le due varianti sono equivalenti.

Teorema:

MdT e MdT multinastro sono modelli equivalenti.

Dimostrazione:

L'implicazione diretta è ovvia, in quanto una MdT è una MdT multinastro, con $k = 1$. Vogliamo dimostrare che per ogni MdT multinastro è possibile costruire una MdT convenzionale che riconosce lo stesso linguaggio. A tal scopo, effettuiamo una simulazione. Utilizziamo come esempio la MdT multinastro, con $k = 3$, definita in precedenza.

Una soluzione è immaginare i k nastri affiancati, ognuno con indicata la posizione della testina. Dobbiamo codificare questa informazione su un solo nastro. Per fare ciò, quindi, possiamo concatenare il contenuto dei k nastri, su k blocchi consecutivi separati da un carattere particolare (in questo esempio, #):

- ogni blocco avrà lunghezza variabile che dipende dal contenuto del nastro corrispondente;
- un elemento marcato (con \perp) nel blocco i -esimo indica la posizione della testina i -esima (ad esempio, se la testina S punta ad un elemento del primo blocco e legge y' , allora la testina del primo nastro è in questa posizione e legge y');
- usiamo un alfabeto esteso Γ_2 tale che $y' \in \Gamma_2$ per ogni $y \in \Gamma$.

Nella figura a destra, notiamo che il primo nastro (input 11) punta alla cella 1 (1), il secondo nastro (input ab) punta alla cella 1 (b), il terzo nastro (input uv) punta alla cella 0 (u). Sotto è posta la macchina M' convenzionale.

Per ogni istruzione della MdT multinastro M , la MdT M' :

1. scorre i k nastri e "raccoglie" informazioni sui k simboli letti;
2. applica la transizione, scorrendo i k nastri e applicando su ciascuno scrittura e spostamento.
3. infine la MdT entra nello stato che ricorda il nuovo stato di S e riposiziona la testina all'inizio del nastro.

Questo comporta molti più stati e mosse, ma otteniamo comunque che S simula M .

A lato è posto un esempio, dove si applica la transizione $\delta(q, a, b, b) = (s, c, a, c, R, L, R)$.

Nella macchina ad un nastro, prima legge il contenuto dei nastri (quindi, sa che sul primo nastro viene letta a, sul secondo viene letta b, sul terzo viene letta b); ora, sa che la mossa è $\delta(q, a, b, b) = (s, c, a, c, R, L, R)$, per cui si dovrà scrivere:

nastro 1) c al posto di a', per poi spostarsi a destra (quindi, si dovrà sostituire la b in cella 4 con una b');

nastro 2) a al posto di b', per poi spostarsi a sinistra (quindi, si dovrà sostituire la b in cella 6 con una b');

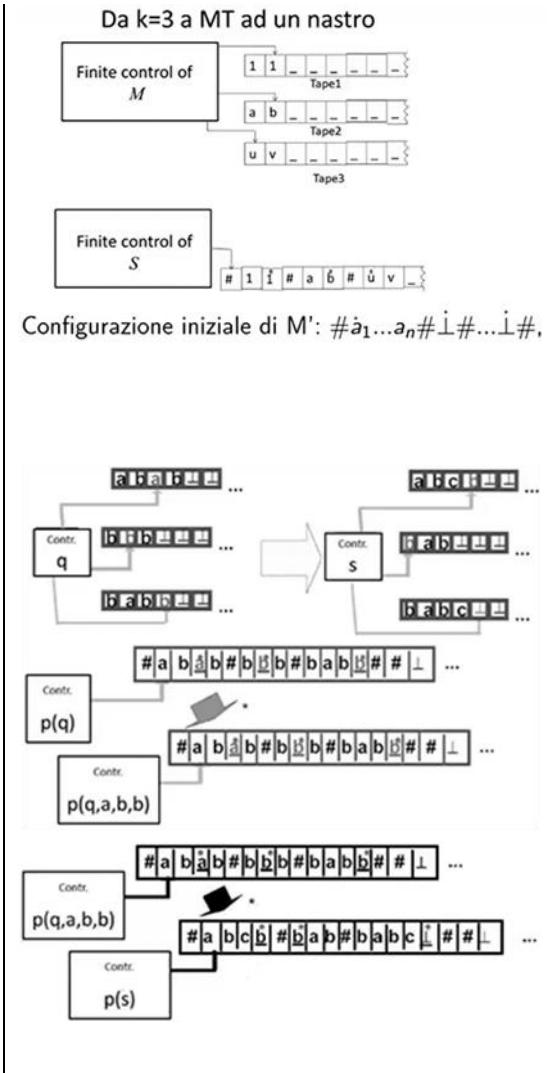
nastro 3) c al posto di b', per poi spostarsi a destra (quindi, si dovrà sostituire il # in qualche modo).

Nella parte finale della figura notiamo il cambiamento: nel nastro 3, abbiamo shiftato a destra tutti i caratteri dopo la c, ed inserito un \perp per indicare che quella è la fine del nastro. Questo è possibile, dato che la lunghezza del nastro di una MdT è variabile nel tempo.

Per ogni mossa del tipo $\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, D_1, \dots, D_k)$:

- S scorre il nastro verso destra, fino al primo \perp , memorizzando nello stato i simboli marcati sui singoli nastri. Si ricordi che la memorizzazione viene implementata con uno stato aggiuntivo per ogni possibile sequenza di $\leq k$ simboli di Γ ;
- la testina si riposiziona all'inizio del nastro;
- poi il nastro viene scorso di nuovo, eseguendo su ogni sezione (cioè un nastro di M) le azioni che simulano quelle delle testine di M , ossia scrittura e spostamento.

Durante il secondo passaggio, S scrive su tutti i simboli "marcati", e sposta il marcitore alla nuova posizione della testina corrispondente di M .



Nota: Se una testina di M muove su \perp più a sinistra del suo nastro, allora la testina virtuale (il puntino su un simbolo) sul segmento corrispondente del nastro di S si sposta sul delimitatore #. In questo caso, S deve spostare di una posizione tutto il suffisso del nastro a partire da # fino all'ultimo # a destra. Dopo, S scrive \perp nella prima posizione soggetta a shift (cioè, dove c'era #).

4.2 MACCHINA DI TURING NON DETERMINISTICA

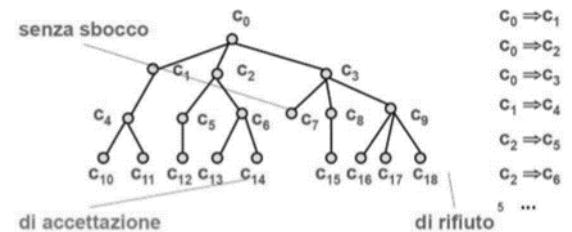
Una MdT **non deterministica** (NMdT) è una macchina di Turing convenzionale avente funzione di transizione $\delta: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$, dove P è l'insieme potenza di tale prodotto cartesiano, invece di $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$; il non determinismo, anche in questo caso, permette di avere associato ad uno stato e ad un simbolo letto sul nastro più transizioni: per ogni transizione, la computazione proseguirà autonomamente in parallelo. La computazione di una NMdT è ben descritta da un albero contenente ogni configurazione raggiungibile dalla iniziale (c_0) su un dato input: se un cammino (da c_0) raggiunge lo stato accetta, allora la NMdT accetta l'input, anche se altri cammini raggiungono uno stato reject.

Siccome l'insieme potenza contiene anche il vuoto, è possibile che una computazione si fermi a seguito di input incorretto: nella figura a lato, ad

esempio, c_7 non produce alcuna nuova configurazione.

In questo caso, la macchina si ferma dopo tre passi: avendo raggiunto una configurazione di accettazione (c_{14}), la stringa in input verrà accettata, indipendentemente da ciò che accade sugli altri cammini.

Le NMdT sembrerebbero più potenti delle MdT ordinarie. In realtà, possiamo dimostrare che le due varianti sono equivalenti.



Teorema:

Per ogni NMdT esiste una MdT deterministica equivalente.

Dimostrazione:

In un verso è ovvio, in quanto una MdT è anche una NMdT, per la definizione stessa.

Dimostriamo, ora, che se abbiamo un linguaggio riconosciuto da una NMdT allora esso è riconosciuto anche da una MdT. A tal scopo, guardiamo alla computazione di una NMdT N come a delle configurazioni raggiungibili da quella iniziale c_0 su input w: esaminiamo l'albero della figura precedente, e simuliamo questo comportamento seguendo l'albero delle configurazioni, eseguendo tutte le computazioni.

Visitiamo l'albero con strategia *BFS* – Breadth-First Search (cioè, *ricerca in ampiezza*) – ed eseguiamo un ramo per volta fin quando non lo esaminiamo per intero. Partendo dalla configurazione iniziale c_0 andiamo in c_1, c_2 e c_3 ; al secondo step, torniamo al primo nodo e visitiamo tutti i nodi che esso produce (c_4), torniamo al secondo nodo e visitiamo tutti i nodi che esso produce (c_5 e c_6), torniamo al terzo nodo e visitiamo tutti i nodi che esso produce (c_7, c_8, c_9); proseguiamo in questo modo fin quando non raggiungiamo una configurazione di accettazione (o di rifiuto).

Formalmente, per ogni input w:

- M deve eseguire tutte le possibili computazioni di N su w;
- M deve accettare se e solo se almeno una raggiunge lo stato accetta.

Utilizziamo una BFS dell'albero delle computazioni (eseguendo tutte le simulazioni in contemporanea), in quanto alcune delle computazioni di N possono essere infinite, e ciò implica che:

- alcuni cammini sono infiniti;
- se M esegue la simulazione e segue un cammino infinito, va in loop (anche se su un altro cammino raggiungerebbe lo stato accetta).

L'algoritmo specificato in precedenza è formalizzato come segue:

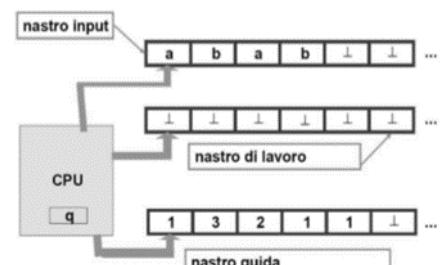
1. Esegui il primo passo di ogni computazione. Se almeno una accetta, allora accetta;
2. Esegui il secondo passo di ogni computazione. Se almeno una accetta, allora accetta;

...

- i. Esegui il passo i-esimo di ogni computazione. Se almeno una accetta, allora accetta.

Questa simulazione la facciamo mediante una MdT D a 3 nastri. Inizialmente, il nastro 1 contiene l'input di N e gli altri due sono vuoti; la simulazione di N procede come segue:

1. Copia l'input sul nastro 2 (usato per eseguire la computazione);
2. Esegui un prefisso di questa computazione, usando il contenuto del nastro 3. Se si raggiunge una configurazione accetta, allora accetta l'input;
3. Aggiorna il contenuto del nastro 3 per ottenere la successiva computazione (si incrementa il contenuto di 1, fin quando non si arriva al numero b, dato che la cifra è b-aria);
4. Vai al passo 1.



Per gestire il contenuto del nastro 3, formalmente, sappiamo che N ha più possibili transizioni da una stessa configurazione:

- per ogni configurazione di N, la MdT D codifica tutte le possibili transizioni e le enumera;
- sia COMP_i il prefisso lungo i di una computazione di N (dove i rappresenta il numero di passi da eseguire). Codifichiamo COMP_i con stringa $b_1 \dots b_i$ di i simboli, dove b è il massimo numero di transizioni possibili da qualsiasi configurazione. Allora:

- D parte con la configurazione iniziale;
- b_1 è il numero della transizione al primo passo di COMP_i ;
- per $2 \leq j \leq i$, b_j è il numero della transizione attuale al passo j -esimo di COMP_i .

Ad esempio, la stringa 421 codifica un prefisso di lunghezza 3 (quindi, $b = 3$) in cui:

- al primo passo, N esegue la quarta transizione nella lista delle transizioni possibili (dalla configurazione iniziale);
- al secondo passo, N esegue la seconda transizione nella lista delle transizioni possibili (dalla configurazione raggiunta al passo 1);
- al terzo passo, N esegue la prima transizione nella lista delle transizioni possibili (dalla configurazione raggiunta al passo 2).

Nota. Alcune configurazioni possono ammettere meno di b scelte, quindi non tutte le sequenze b-arie rappresentano un prefisso di computazione.

Riassumendo, la simulazione di N con D risulta:

1. Scrivi 1 sul nastro 3;
2. Copia input sul nastro 2;
3. Se il prefisso del numero b-ario è la codifica di un prefisso di una computazione di N, allora eseguo. Se la computazione accetta, allora accetta, altrimenti vai al passo 4;
4. Incrementa di 1 il numero b-ario sul nastro 3 (con i numeri che vanno da 1 a b);
5. Vai al passo 2.

Non abbiamo detto *come* funziona, in quanto la macchina deterministica diventa molto complicata da costruire. Non diamo maggiori dettagli, ma ci limitiamo a dire come dovrebbe funzionare: questo perché a noi basta sapere che una tale macchina esiste.

Si è dimostrato che questi tre modelli di calcolo sono tra loro equivalenti. A noi basta conoscere l'equivalenza tra MdT e calcolatori (programmi o algoritmi).

Tesi di Church-Turing

Se esiste un algoritmo per eseguire un calcolo, allora questo calcolo può essere eseguito da una MdT (o equivalenti).

Da questa Tesi ricaviamo la sua negazione: infatti, ciò è equivalente a dire che se non esiste una MdT allora non esiste un algoritmo. In seguito, vedremo, basandoci su questo fatto, che esistono problemi per cui non è possibile trovare un algoritmo che risolve quel problema: cioè, esistono problemi che *non sono computabili*.

5. LINGUAGGI DECIDIBILI

I **problemi di decisione** sono problemi che hanno come soluzione una risposta SI o NO.

Esempi:

- PRIMO: Dato un numero x , x è primo?
- CONNESSO: Dato un grafo G , G è connesso? Si ricordi che un grafo si dice *connesso* se per ogni coppia di nodi esiste un cammino che li collega all'interno del grafo.
- ACCETTAZIONE DI UN DFA: Dato un DFA β e una stringa w , l'automa β accetta w ?

Se si vuole **risolvere un problema di decisione** utilizzando una MdT, in qualche modo **occorre trasformare il problema in un linguaggio**: questo perché le MdT sono essenzialmente accettatori di linguaggi.

Ricorda: l'input per una MdT è sempre una stringa. Se vogliamo dare in input altri oggetti, questi devono essere codificati come stringhe: ciò rappresenta il primo step per effettuare questa trasformazione. Qualsiasi oggetto, infatti, può essere codificato come stringa.

Rappresenteremo, quindi, i **problemi di decisione mediante linguaggi**.

Esempi:

- Il linguaggio che rappresenta il problema "PRIMO" è

$$P = \{\langle x \rangle \mid x \text{ è un numero primo}\},$$

dove $\langle x \rangle$ denota una rappresentazione sotto forma di stringa su un alfabeto Σ dell'oggetto x (*codifica*).

Utilizzeremo sempre la notazione $\langle x \rangle$ per indicare una tale rappresentazione. Quindi, se x è un intero, ed usiamo come alfabeto $\Sigma = \{0, 1\}$, denoteremo con $\langle x \rangle$ una codifica binaria che rappresenta tale intero.

Nota. $\langle x \rangle \in P$ se e solo se PRIMO ha risposta SI su input x .

- Il linguaggio che rappresenta il problema "CONNESSO" è

$$A = \{\langle G \rangle \mid G \text{ è un grafo connesso}\},$$

dove $\langle G \rangle$ denota una "ragionevole" codifica di G mediante una stringa su un alfabeto Σ .

Dunque, se G è connesso allora $\langle G \rangle \in A$ e la stringa viene accettata. Se G' non è connesso, allora $\langle G' \rangle \notin A$ e la stringa non viene accettata.

Per rappresentare un grafo, possiamo prendere $\Sigma = \{0, 1, \dots, 9, (,), \#\}$ e $\langle G \rangle = \{\{1, 2, 3\}, \{(1, 2), (2, 3), (3, 1)\}\}$, cioè una sequenza in cui il primo elemento è l'insieme dei vertici ed il secondo elemento è un insieme di *edge*, che sono tutti gli edge del grafo. Questa è la rappresentazione che useremo.

Un modo alternativo per rappresentare un grafo è prendere $\Sigma = \{0, 1, (,), \#\}$ e codificare G mediante una stringa $(1\#10\#11)\#((1\#10)\#(10\#11)\#(11\#1))$.

Sia $A = \{\langle G \rangle \mid G \text{ è un grafo connesso}\}$ un linguaggio di stringhe che rappresentano grafi connessi (non orientati). Si ha che $\langle G \rangle \in A$ se e solo se G è istanza SI per CONNESSO. Risolvere CONNESSO equivale a decidere il linguaggio A .

In questo modo esprimiamo un problema computazionale in termini di riconoscimento di un linguaggio (cioè, l'insieme delle codifiche di istanze SI per il problema).

Esempio:

Si ha in input un grafo G . Un modo per verificare se G è connesso è il seguente:

1. Seleziona un nodo di G e marcalo;
2. Ripeti finché si trovano nuovi nodi da marcare:
 - 2.1. Per ogni nodo v in G , se v è connesso ad un nodo marcato, allora marcalo;
3. Se tutti i nodi risultano marcati allora accetta, altrimenti reject.

Vogliamo convincerci che il risultato precedente è realizzabile mediante una MdT. Quindi, vogliamo una MdT che riconosce l'insieme $A = \{\langle G \rangle \mid G \text{ è un grafo connesso}\}$.

Il grafo può essere rappresentato mediante due liste: la lista dei *nodi* (numeri naturali) e la lista degli *edges* (coppie di numeri). Ad esempio, abbiamo il grafo $\langle G \rangle = \{\{A, B, C, D\}, \{(A, B), (A, D), (B, C), (C, D)\}\}$.

Nota. Non specifichiamo l'alfabeto (binario, decimale, ...). Ignoriamo i dettagli non importanti dell'implementazione.

Se facciamo questo, bisogna controllare prima di tutto che l'input sia:

- una lista di nodi (digit) senza ripetizioni;
- una lista di coppie di nodi (digit presenti nella lista precedente).

Un'implementazione può essere la seguente:

- 1) **Seleziona un nodo di G e marcalo:** Marca il primo nodo sulla lista (ad esempio, con un · sul digit più a sinistra);
- 2) **Ripeti finché i nuovi nodi sono marcati:**

- 2.1) *Per ogni nodo v in G , se v è connesso ad un nodo marcato, allora marcalo:*

- 2.1.1) Sottolinea il primo nodo n_1 senza · sulla lista (sottolineando il digit più a sinistra);
- 2.1.2) Sottolinea il primo nodo n_2 con · sulla lista;
- 2.1.3) Cerca se esiste edge (n_1, n_2) : se SI, allora marca n_1 con · e vai a 2); se NO, sia n_2 il successivo nodo con · sulla lista, sottolinealo e vai a 2.1.3)

- 3) **Se tutti i nodi risultano marcati allora accetta, altrimenti reject:** Scorri la lista dei nodi: se tutti i nodi hanno · allora accetta, altrimenti reject.

5.1 DECIDIBILITÀ

L'obiettivo è analizzare i *limiti* della risoluzione di problemi mediante algoritmi: ne studieremo il potere computazionale nella soluzione dei problemi, e proveremo che esistono problemi che possono essere risolti mediante algoritmi ed altri no.

Nell'esempio "PRIMO", abbiamo espresso un problema computazionale come un problema di *riconoscimento di un linguaggio* (cioè, l'insieme delle codifiche di istanze SI per il problema): dunque, risolvere "PRIMO" equivale a decidere un linguaggio $P = \{\langle x \rangle \mid x \text{ è un numero primo}\}$.

Sia data in input una stringa w ad una MdT M : vogliamo sapere rispondere alla domanda " M si arresta su input w ?". Il linguaggio corrispondente a questo problema è $\text{HALT}_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che si arresta su } w\}$.

Per ora, introduciamo degli strumenti con cui lavoreremo: **cardinalità di insiemi (infiniti)** e **diagonalizzazione (metodo introdotto da Cantor)**.

La **cardinalità** di un insieme è banalmente la sua taglia. Due insiemi hanno la stessa cardinalità se è possibile stabilire una corrispondenza tra i loro elementi.

Esempio: $A = \{1, 2, 3\}$, $B = \{4, 3, 5\} \Rightarrow 1 - 4, 2 - 3, 3 - 5$.

Paradosso di Hilbert:

Nel paese senza confini esiste il più grande di tutti gli alberghi, cioè un albergo con infinite stanze. Tuttavia, anche gli ospiti sono infiniti, e il proprietario ha esposto un cartello con la scritta "Completo". Ad un tratto, si presenta un viaggiatore che ha assolutamente bisogno di una camera per la notte. Egli non fa questione di prezzo e infine convince l'albergatore, il quale trova il modo di alloggiarlo. *Come fa?*

Sposta tutti i clienti nella camera successiva (l'ospite della 1 alla 2, l'ospite della 2 alla 3, ...); in questo modo, è possibile, essendo l'albergo infinito, sistemare (nella camera 1) il nuovo ospite anche se l'albergo è pieno.

[Passo completato]

Poco dopo, arriva una comitiva di *infiniti turisti*, anche in questo caso l'albergatore si lascia convincere (in fondo si tratta di un grosso affare), e trova posto ai *nuovi infiniti ospiti* con la stessa facilità con cui aveva alloggiato l'ospite in più. *Come fa* (senza ripetere infinite volte il passo visto prima)?

Sposta ogni ospite nella stanza con un numero doppio rispetto a quello attuale (dalla 1 alla 2, dalla 2 alla 4, ...), lasciando ai nuovi ospiti tutte le camere con i numeri dispari, che sono essi stessi infiniti, risolvendo dunque il problema. Questa operazione, infatti, fa sì che le sole stanze pari siano occupate, il che implica che le stanze dispari siano libere.

Gli ospiti sono tutti, dunque, sistemati, benché l'albergo fosse pieno.

[Passo completato]

Ancora più complesso: ci sono infiniti alberghi con infinite stanze, tutti al completo. Tutti gli alberghi chiudono, tranne uno. Tutti gli ospiti vogliono alloggiare nell'unico albergo rimasto aperto. *Come fa* (senza ripetere infinite volte il passo visto prima)?

Assegna ad ogni persona una coppia di numeri (n, m) in cui n indica l'albergo di provenienza, e m la relativa stanza. Gli ospiti sono quindi etichettati come segue:

(1, 1)	(1, 2)	(1, 3)	(1, 4)	...
(2, 1)	(2, 2)	(2, 3)	(2, 4)	...
(3, 1)	(3, 2)	(3, 3)	(3, 4)	...
(4, 1)	(4, 2)	(4, 3)	(4, 4)	...
...

A questo punto, basta assegnare le nuove stanze agli ospiti secondo un criterio ordinato, ad esempio per le diagonali, come indicato nella pagina seguente.

$$(1, 1) \rightarrow 1; (1, 2) \rightarrow 2; (2, 1) \rightarrow 3; (1, 3) \rightarrow 4; (2, 2) \rightarrow 5; (3, 1) \rightarrow 6; \dots$$

Possiamo, quindi, indicare una corrispondenza tra cardinalità di insiemi infiniti? Ad esempio, l'insieme dei numeri naturali è **INFINITO**, così come l'insieme dei numeri naturali pari e l'insieme dei numeri naturali dispari. Anche l'insieme dei numeri reali è **INFINITO**.

La quantità di numeri reali è la stessa di quella dei numeri naturali? Come si misura la cardinalità di insiemi infiniti?

5.2 INDECIDIBILITÀ

Metodo della Diagonalizzazione:

introdotto da Cantor nel 1973 mentre cercava di determinare come stabilire se, dati due insiemi infiniti, uno è più grande dell'altro. Cantor osservò che due insiemi finiti hanno la stessa cardinalità se gli elementi dell'uno possono essere messi in corrispondenza uno a uno con quelli dell'altro.

Successivamente, estese questo concetto agli insiemi infiniti.

Una funzione $f: X \rightarrow Y$ è una relazione input-output, dove X è l'insieme dei possibili input (**dominio**) e Y è l'insieme dei possibili output (**codominio**).

Per ogni input $x \in X$ esiste un solo output $y = f(x) \in Y$.

Una funzione $f: X \rightarrow Y$ è **iniettiva** se $\forall x, x' \in X, x \neq x' \Rightarrow f(x) \neq f(x')$.

Una funzione $f: X \rightarrow Y$ è **suriettiva** se $\forall y \in Y, y = f(x)$ per qualche $x \in X$.

Una funzione $f: X \rightarrow Y$ è una funzione **bijettiva** di X su Y (o una **biezione** tra X e Y) se f è iniettiva e suriettiva.

Una funzione biiettiva è una corrispondenza uno a uno tra gli elementi del dominio e gli elementi del codominio.

Esempio:

1. $f: \{1, 2, 5\} \rightarrow \{2, 4, 7\}$, con $1 \rightarrow 2, 2 \rightarrow 5, 5 \rightarrow 4$ è una funzione, ma non è né iniettiva né suriettiva.
2. $f: \{1, 2, 5\} \rightarrow \{2, 4, 7, 9\}$, con $1 \rightarrow 2, 2 \rightarrow 4, 5 \rightarrow 7$ è una funzione iniettiva ma non suriettiva.
3. $f: \{1, 2, 5\} \rightarrow \{2, 4\}$, con $1 \rightarrow 2, 2 \rightarrow 4, 5 \rightarrow 2$ è una funzione suriettiva ma non iniettiva.
4. $f: \{1, 2, 5\} \rightarrow \{2, 4, 7\}$, con $1 \rightarrow 2, 2 \rightarrow 4, 5 \rightarrow 7$ è una funzione biiettiva.
5. $f: \mathbb{N} \rightarrow \{2n \mid n \in \mathbb{N}\}$, con $n \rightarrow 2n$ è una funzione biiettiva.

Due insiemi X e Y hanno la stessa cardinalità se esiste una funzione biiettiva $f: X \rightarrow Y$ di X su Y .

Un insieme è **enumerabile** (o **numerabile**) se ha la stessa cardinalità di un sottoinsieme di \mathbb{N} .

Se A è numerabile, allora possiamo "numerare" gli elementi di A e scrivere una lista (a_1, a_2, \dots) ; cioè, per ogni numero naturale i , allora possiamo specificare l'elemento i -esimo della lista.

Esempio:

Per l'insieme dei numeri naturali pari, l'elemento i -esimo della lista corrisponde a $2i$.

Per quest'ultima proprietà, quindi, possiamo associare ad un insieme A (**finito**) un sottoinsieme dell'insieme \mathbb{N} (ovviamente, \mathbb{N} è **infinito**) attraverso una funzione biettiva $f: A \rightarrow \mathbb{N}$. Ciò vuol dire che avremo associato un unico elemento di A ad un unico elemento di \mathbb{N} .

L'insieme dei numeri razionali è numerabile. Mostriamo che possiamo formare una lista di tutti i numeri razionali. Formiamo un rettangolo infinito, come mostrato nella figura a destra. Questa tabella, con infinite righe e colonne, è corretta, e si noti che se scorre per le righe s'incrementa man mano il numeratore, mentre se si scorre per le colonne s'incrementa man mano il denominatore.

1/1	1/2	1/3	1/4	...
2/1	2/2	2/3	2/4	...
3/1	3/2	3/3	3/4	...
4/1	4/2	4/3	4/4	...
...

Per far vedere che questo insieme è numerabile, bisogna mostrare la biezione (o, equivalentemente, bisogna listare tutti i numeri razionali).

Sicuramente non possiamo procedere per righe, dato che se prendiamo tutta la prima linea non arriviamo mai alla seconda; allo stesso modo, non

possiamo procedere per colonne. Un'idea può essere procedere *in diagonale* (usando la secondaria, in questo caso). Dunque, prendiamo tutti gli elementi della prima diagonale (1/1) per poi prendere tutti gli elementi della seconda diagonale (2/1, 1/2), e così via.

Nota: Dovremmo eliminare i duplicati (ad esempio, 1/1 = 1 e 2/2 = 1), ma è solo una questione tecnica.

Costruita la lista (a_1, a_2, \dots), possiamo quindi numerarla. Ciò vuol dire che esiste la biezione tra l'insieme dei numeri razionali e l'insieme dei numeri reali, il che significa che l'insieme dei numeri razionali è numerabile.

L'insieme Σ^* è numerabile: per dimostrarlo, listiamo prima la stringa vuota, poi le stringhe (in ordine lessicografico) lunghe 1, poi 2, e così via. A questo punto, come nell'esempio precedente, possiamo numerare la lista (partendo da 1) e quindi *numerare* l'insieme Σ^* .

1/1	1/2	1/3	1/4	...
2/1	2/2	2/3	2/4	...
3/1	3/2	3/3	3/4	...
4/1	4/2	4/3	4/4	...
...

1/1, 2/1, 1/2, 3/1, 2/2, 1/3, ...

Esempio:

$$\Sigma = \{0, 1\} \Rightarrow w_0 = \epsilon, w_1 = 0, w_2 = 1, w_3 = 00, \dots$$

In questo caso, possiamo sapere anche la posizione in cui compare una determinata stringa nella lista. Presa 001, ad esempio, sappiamo che $|001| = 3$, e che essa compare in posizione $2^0 + 2^1 + 2^2 + 2$ (quest'ultimo in quanto è la *seconda* stringa in ordine lessicografico tra tutte le stringhe di lunghezza 3) = 9; quindi, la stringa 001 si trova in posizione 9 nella lista.

L'insieme delle descrizioni di MdT $\{\langle M \rangle \mid M \text{ è una MdT sull'alfabeto } \Sigma\}$ è numerabile: è possibile codificare una MdT M con una stringa su un alfabeto Σ .

In generale, *per determinare se un insieme è numerabile bisogna mostrare che esiste una biezione con \mathbb{N}* , cioè per ogni elemento possiamo far vedere che posizione occupa all'interno dell'insieme.

Teorema:

L'insieme dei numeri reali \mathbb{R} non è numerabile

Dimostrazione:

Sia per assurdo \mathbb{R} numerabile; allora possiamo costruire la lista $f(1), f(2), f(3), \dots$

Per ogni $i \geq 1$, scriviamo $f(i) = f_0(i)f_1(i)f_2(i)f_3(i)\dots$ Cioè, sappiamo che ogni $f(i)$ è un numero reale, ed essendo tale ha una parte intera ed una parte decimale. Nella rappresentazione precedente, $f_0(i)$ è la parte intera, separata da una virgola dalla parte decimale $f_1(i)f_2(i)f_3(i)\dots$ Ad esempio, se $f(1) = 4,256\dots$ allora $f_0(1) = 4$, $f_1(1) = 2$, $f_2(1) = 5$, $f_3(1) = 6$, ...

Organizziamoli in una matrice, posta a lato, in cui le colonne sono indicizzate con gli interi 1, 2, 3, ..., i e la riga i -esima è l'elemento $f(i)$ che compare nella lista.

Quindi, ad esempio, nella riga 1 compaiono le cifre decimali del primo elemento; stiamo ignorando la parte intera di ogni numero.

A questo punto, consideriamo la diagonale di questa matrice. Sia $x \in (0, 1)$ il numero

$x=0.x_1x_2x_3\dots x_i\dots$ ottenuto scegliendo $x_i \neq f_i(i)$ per ogni $i \geq 1$. Chiaramente, $x \in \mathbb{R}$.

A questo punto, risulta x nella lista? Se $x = f(j)$, allora il suo j -esimo digit soddisfa $x_j = f_j(j)$; ma $x_j \neq f_j(j)$ per definizione di x . Questa è una contraddizione.

Quindi, $x \in \mathbb{R}$ non può comparire nella lista e \mathbb{R} non è numerabile.

$i \setminus f(i)$	f_1	f_2	f_3
1	$f_1(1)$	$f_2(1)$	$f_3(1)$...	$f_1(1)$...
2	$f_1(2)$	$f_2(2)$	$f_3(2)$...	$f_1(2)$...
3	$f_1(3)$	$f_2(3)$	$f_3(3)$...	$f_1(3)$...
\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
i	$f_1(i)$	$f_2(i)$	$f_3(i)$...	$f_1(i)$...
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

Abbiamo detto che $\Sigma^* = \{w_1, w_2, \dots\}$ e l'insieme delle descrizioni di MdT su Σ (cioè $\{M_1, M_2, \dots\}$) sono numerabili. Anche in questo caso possiamo usare il Metodo della diagonalizzazione; quindi, costruiamo la tabella seguente, dove $x_{i,j} = 1$ se $w_j \in L(M_i)$, $x_{i,j} = 0$ altrimenti.

Possiamo sfruttare la diagonale principale ($x_{1,1}, x_{2,2}, \dots, x_{i,i}, \dots$) per costruire un nuovo linguaggio

$L = \{w_i \in \Sigma^* \mid w_i \notin L(M_i)\}$. Questo

linguaggio è il “complemento della diagonale”:

- se l'elemento (M_i, w_i) della diagonale $x_{i,i} = 1$, allora $w_i \notin L$;
- se l'elemento (M_i, w_i) della diagonale $x_{i,i} = 0$, allora $w_i \in L$.

Vogliamo stabilire se L può comparire nella lista, cioè se $L = L(M_h)$ per qualche h ; supponiamo che $L = L(M_h)$, allora:

- $w_h \in L \Rightarrow x_{h,h} = 0 \Rightarrow w_h \notin L(M_h) = L$, che è una contraddizione;
- $w_h \notin L \Rightarrow x_{h,h} = 1 \Rightarrow w_h \in L(M_h) = L$, che è una contraddizione.

	w_1	w_2	w_3	...	w_i	w_j
M_1	$x_{1,1}$
M_2	.	$x_{2,2}$
.	.	.	$x_{3,3}$.	.	.
.	.	.	.	$x_{4,4}$.	.
M_i	$x_{i,i}$	$x_{i,j}$
.

Da quest'ultimo risultato, otteniamo che $L \neq L(M_h)$ per ogni h , il che significa che L non può comparire nella lista. Da ciò segue che “esistono più linguaggi che Macchine di Turing”.

Corollario:

Esistono linguaggi che non sono Turing riconoscibili.

Macchina di Turing universale:

Una MdT **universale** U simula la computazione di una qualsiasi MdT M .

Allora U riceve in input una *rappresentazione* $\langle M, w \rangle$ di M e di un possibile input w di M .

N.B. Abbiamo visto che è possibile codificare una MdT M e una stringa w con una stringa su un alfabeto Σ .

Esempio:

$\langle M, w \rangle$ = “codifica di M ”#“codifica di w ”, cioè la concatenazione (#) tra le due codifiche.

Una tale macchina è chiamata “universale” perché la computazione di una qualsiasi MdT può essere simulata da U .

$$\langle M, w \rangle \rightarrow \text{MdT universale } U \rightarrow \begin{cases} \text{accetta} & \text{se } M \text{ accetta } w \\ \text{rifiuta} & \text{se } M \text{ rifiuta } w \end{cases}$$

Nota. U può anche andare in loop.

Dunque, abbiamo costruito una macchina universale U il cui linguaggio è $A_{TM} = \{\langle M, w \rangle \mid M \text{ accetta } w\}$.

Teorema:

Il linguaggio $A_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che accetta la parola } w\}$ è Turing riconoscibile

Dimostrazione:

Definiamo una MdT U che accetta A_{TM} : sull'input $\langle M, w \rangle$, dove M è una MdT e w è una stringa:

1. Simula M sull'input w ;
2. Se M accetta, allora accetta; se M rifiuta, allora rifiuta.

Abbiamo visto una MdT che simula un automa. Simulare una MdT M con un'altra MdT risulta molto simile:

1. Marca lo stato iniziale di M (stato corrente) e il primo simbolo sul nastro (posizione corrente della testina);
2. Cerca la prossima transizione (nella parte che descrive la funzione di transizione). Sia $(q, x) \rightarrow (q', x', D)$;
3. Esegui la transizione;
4. Aggiorna lo stato corrente (marca q') e la posizione corrente della testina (marca simbolo a D);
5. Se lo stato corrente risulta $q_{\text{accept}}/q_{\text{reject}}$ decidi di conseguenza, altrimenti ritorna al passo 2.

Nota: U è detta MdT universale.

Nota: U riconosce A_{TM} : accetta ogni coppia $\langle M, w \rangle \in A_{TM}$.

Nota: U cicla su $\langle M, w \rangle$ se (e solo se) M cicla su w . Quindi U non decide A_{TM} .

Teorema (INDECIDIBILITÀ DEL PROBLEMA DELLA FERMATA):

Il linguaggio $A_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che accetta la parola } w\}$ non è decidibile.

Notiamo che un tale linguaggio può essere trasformato in un problema di decidibilità/indecidibilità su input M, w e che risponde alla domanda “ M accetta w ?” Non è possibile, però, decidere se una data macchina si ferma su un dato input.

Paradosso di Bertrand Russel:

In un paese vive un solo barbiere, un uomo ben sbarbato, che rade tutti e soli gli uomini del villaggio che non si radono da soli. *Chi sbarba il barbiere?*

- Se il barbiere rade sé stesso, allora per definizione il barbiere non rade se stesso;

- se il barbiere non rade se stesso, allora, dato che il barbiere rade tutti quelli che non si radono da soli, il barbiere rade se stesso.

Si tratta di un'antinomia: compresenza di due affermazioni contraddittorie che possono essere entrambe dimostrate o giustificate.

In generale, Russel pose il problema dell'insieme di tutti gli insiemi che non contengono se stessi. L'autoreferenza può causare problemi.

Possiamo usare un'autoreferenzialità del genere per dimostrare l'indecidibilità di A_{TM} .

Dimostrazione:

Supponiamo per assurdo che A_{TM} sia decidibile, cioè che esiste una macchina di Turing H con due possibili risultati di computazione (accettazione, rifiuto) e tale che

$$H = \begin{cases} \text{accetta} & \text{se } M \text{ accetta } w \\ \text{rifiuta} & \text{se } M \text{ non accetta } w \end{cases}$$

Si noti che, a differenza di U , la macchina H è un decisore (o accetta o rifiuta, ma non va mai in loop).

Costruiamo una nuova MdT D che usa H come sottoprogramma D sull'input $\langle M \rangle$, dove M è una MdT:

1. Simula H sull'input $\langle M, \langle M \rangle \rangle$;
2. Fornisce come output l'opposto di H , cioè se H accetta, allora $rifiuta$ e se H rifiuta, allora $accetta$.

$$\langle M \rangle \rightarrow D \rightarrow \langle M, \langle M \rangle \rangle \rightarrow H \rightarrow \begin{cases} \text{accetta} & \text{se } M \text{ accetta } \langle M \rangle \\ \text{rifiuta} & \text{se } M \text{ non accetta } \langle M \rangle \end{cases} \rightarrow I \rightarrow \begin{cases} \text{rifiuta} & \text{se } D \text{ accetta } \langle M \rangle \\ \text{accetta} & \text{se } D \text{ non accetta } \langle M \rangle \end{cases}$$

Quindi:

$$D(\langle M \rangle) = \begin{cases} \text{rifiuta} & \text{se } M \text{ accetta } \langle M \rangle \\ \text{accetta} & \text{se } M \text{ non accetta } \langle M \rangle \end{cases}$$

Se ora diamo in input a D la sua stessa codifica $\langle D \rangle$ abbiamo:

$$\langle D \rangle \rightarrow D \rightarrow \langle D, \langle D \rangle \rangle \rightarrow H \rightarrow \begin{cases} \text{accetta} & \text{se } D \text{ accetta } \langle D \rangle \\ \text{rifiuta} & \text{se } D \text{ non accetta } \langle D \rangle \end{cases} \rightarrow I \rightarrow \begin{cases} \text{rifiuta} & \text{se } D \text{ accetta } \langle D \rangle \\ \text{accetta} & \text{se } D \text{ non accetta } \langle D \rangle \end{cases}$$

Cioè, D accetta $\langle D \rangle$ se e solo se D non accetta $\langle D \rangle$.

Questo è assurdo. Tutto è causato dall'assunzione che esiste H . Quindi, H non esiste. \square

Riepilogando la dimostrazione:

1. Definiamo $A_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che accetta la parola } w\}$;
2. Assumiamo che A_{TM} sia decibile; sia H una MdT che lo decide;
3. Usiamo H per costruire una MdT D che inverte le decisioni: $D(\langle M \rangle)$ accetta se M non accetta $\langle M \rangle$ e rifiuta se M accetta $\langle M \rangle$;
4. Diamo in input a D la sua codifica $\langle D \rangle$: $D(\langle D \rangle)$ accetta se e solo se D rifiuta $\langle D \rangle$. Contraddizione.

Anche se “nascosta”, la dimostrazione precedente utilizza la diagonalizzazione. Consideriamo la tabella a lato, dove le M_t sono tutte le MdT numerate, e $\langle M_t \rangle$ sono le loro descrizioni (stringhe): se nella cella (i, j) c'è “acc”, allora $\langle M_j \rangle \in L(M_i)$; se non c'è nulla, allora $\langle M_j \rangle \notin L(M_i)$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$...
M_1	acc		acc		...
M_2	acc	acc	acc	acc	...
M_3					...
M_4	acc	acc			...
:	:	:	:	:	:

Consideriamo H : la MdT H rifiuta anche se M_i va in loop (oltre a se M_i rifiuta). La sua tabella è posta a destra.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$...
M_1	acc	rej	acc	rej	...
M_2	acc	acc	acc	acc	...
M_3	rej	rej	rej	rej	...
M_4	acc	acc	rej	rej	...
:	:	:	:	:	:

Consideriamo, ora, D e $D(\langle D \rangle)$. Dobbiamo considerare la diagonale.

Dove sono posti i tre punti interrogativi (???) non eravamo in grado di mettere né accetta né rifiuta; infatti D , in corrispondenza della sua descrizione, non può né accettare né rifiutare.

Nella prova precedente, quindi, è stato usato il metodo della diagonalizzazione. In conclusione, A_{TM} è Turing riconoscibile ma è **indecidibile**.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$...	$\langle D \rangle$...
M_1	acc	rej	acc	rej
M_2	acc	acc	acc	acc
M_3	rej	rej	rej	rej
M_4	acc	acc	rej	rej
:	:	:	:	:	:	:	:
D	acc	acc	rej	rej	...	???	...
:	:	:	:	:	:	:	:

Che differenza c'è tra le due dimostrazioni? Cioè, che differenza c'è tra U e D ? Chiaramente, la differenza tra le due macchine è proprio nell'esistenza del loop.

Sappiamo che esistono linguaggi che non sono Turing riconoscibili. Vogliamo individuare uno specifico linguaggio non Turing riconoscibile ($\overline{A_{TM}}$, cioè il complemento di A_{TM}).

Diciamo che un linguaggio L è co-Turing riconoscibile se il suo complemento \overline{L} è Turing riconoscibile.

Teorema:

Un linguaggio L è decidibile se e solo se L è Turing riconoscibile e co-Turing riconoscibile.

Dimostrazione:

(\Rightarrow) Se L è decabile, allora esiste una macchina di Turing M con due possibili risultati di una computazione (accettazione, rifiuto) e tale che M accetta w se e solo se $w \in L$. Allora L è Turing riconoscibile. Inoltre, è facile costruire una MdT \overline{M} che accetta w se e solo se $w \notin L$:

$$w \rightarrow \boxed{M} \rightarrow \begin{cases} \text{accetta} & \text{se } w \in L \\ \text{rifiuta} & \text{se } w \notin L \end{cases} \rightarrow \begin{cases} \text{rifiuta} & \text{se } M \text{ accetta (cioè, } w \notin \overline{L}) \\ \text{accetta} & \text{se } M \text{ rifiuta (cioè, } w \in \overline{L}) \end{cases}.$$

(\Leftarrow) Supponiamo che L e il suo complemento siano entrambi Turing riconoscibili. Sia M_1 una MdT che riconosce L e sia M_2 una MdT che riconosce \overline{L} . Definiamo una MdT N a due nastri, la quale si limiterà a simulare in parallelo le macchine M_1 e M_2 .

La macchina N , su input x , funzionerà come segue:

1. Copia x sui nastri di M_1 e M_2 . Quindi, un nastro lo associamo alla macchina M_1 ed uno alla macchina M_2 , dopodiché ogni mossa sarà data dalla coppia di mosse che farebbero M_1 e M_2 ;
2. Simula M_1 e M_2 in parallelo (usa un nastro per M_1 , l'altro per M_2);
3. Se M_1 accetta, allora N accetta. Se M_2 accetta, allora N rifiuta.

$$x \rightarrow \boxed{\begin{matrix} M_1 & \rightarrow \text{accetta} \\ M_2 & \rightarrow \text{accetta} \end{matrix}} \rightarrow \begin{cases} \text{accetta} & \text{se } M_1 \text{ accetta} \\ \text{rifiuta} & \text{se } M_2 \text{ accetta} \end{cases}.$$

Segue che N decide L . Infatti, per ogni stringa x abbiamo due casi:

1. $x \in L$. Ma $x \in L$ se e solo se M_1 si arresta e accetta x . Quindi N accetta x ;
2. $x \notin L$. Ma $x \notin L$ se e solo se M_2 si arresta e accetta x . Quindi N rifiuta x .

Poiché una e solo una delle due MdT accetta x , allora N è una MdT con soli due possibili risultati di una computazione (accettazione, rifiuto) e tale che N accetta x se e solo se $x \in L$.

Teorema:

A_{TM} non è Turing riconoscibile

Dimostrazione:

Supponiamo per assurdo che $\overline{A_{TM}}$ sia Turing riconoscibile. Sappiamo che A_{TM} è Turing riconoscibile. Quindi, A_{TM} è Turing riconoscibile e co-Turing riconoscibile. Per il precedente teorema, A_{TM} è decabile. Questo è assurdo, in quanto abbiamo dimostrato che A_{TM} non è decabile.

□

È importante riconoscere che un problema P è indecidibile. Si può estendere la classe dei problemi indecidibili in due modi:

- 1) Supporre l'esistenza di una MdT che decide P e provare che questo conduce ad una contraddizione (come fatto con A_{TM});
- 2) Considerare un problema P' di cui sia nota l'indecidibilità (ad esempio, A_{TM}) e dimostrare che P' "non è più difficile" del problema in questione P .

Esempio:

Sia $\Sigma = \{0, 1\}$. Consideriamo i problemi:

- $EVEN = \{w \in \Sigma^* \mid w \text{ è la rappresentazione binaria di } n \in \mathbb{N} \text{ pari}\};$
- $ODD = \{w \in \Sigma^* \mid w \text{ è la rappresentazione binaria di } n \in \mathbb{N} \text{ dispari}\}.$

Sia $w \in L$ e sia n il corrispondente decimale di w . È facile costruire la MdT $INCR$ che incrementa il valore di n :

$$w \rightarrow \boxed{INCR} \rightarrow w' (= \text{rappresentazione binaria di } n + 1).$$

Possiamo dire che $EVEN$ "non è più difficile" di ODD : se esiste una MdT R che decide ODD , allora la MdT S decide $EVEN$.

$$S : w \rightarrow \boxed{INCR} \rightarrow w' \rightarrow \boxed{R} (\rightarrow \text{accetta } w' \text{ se } n + 1 \text{ è dispari} \leftrightarrow n \text{ è pari}).$$

Viceversa, se $EVEN$ è indecidibile proviamo così che anche ODD lo è: se per assurdo esistesse una MdT R che decide ODD , allora la MdT S deciderebbe $EVEN$.

6. RIDUCIBILITÀ

Una **riduzione** è un modo di convertire un problema in un altro problema in modo tale che una soluzione al secondo problema può essere usata per risolvere il primo problema.

La riducibilità coinvolge sempre due problemi, chiamati A e B. Se A si riduce a B, possiamo usare una soluzione per B per risolvere A.

Quando A è riducibile a B, trovare la soluzione di A non può essere più difficile di risolvere B perché una soluzione per B offre una soluzione A. In termini di teoria della computabilità, se A riducibile a B e B è decidibile, allora anche A è decidibile.

Equivalentemente, se A è indecidibile e riducibile a B, B è indecidibile.

Il "vero" problema della fermata:

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che si arresta su } w\}$$

Abbiamo $A_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT che accetta } w\}$. L'unico caso che fa sì che $\langle M, w \rangle$ non sia un'istanza Sí è il caso in cui la macchina va in loop su w .

Vogliamo far vedere che questo $HALT_{TM}$ è anch'esso indecidibile, utilizzando una **riduzione** da un problema indecidibile. Se $HALT_{TM}$ fosse decidibile, allora potremmo decidere anche A_{TM} (cioè, se avessimo un decisore per $HALT_{TM}$ allora potremmo costruire un decisore per A_{TM} , il che ci porterà ad una contraddizione per l'indecidibilità di A_{TM}).

Sia R una MdT che decide $HALT_{TM}$ (per assurdo). Costruiamo quindi S (che può decidere A_{TM}) che sull'input $\langle M, w \rangle$, dove M è una MdT e w è una stringa, simula R su $\langle M, w \rangle$:

- se R rifiuta, allora S rifiuta (poiché M va in loop quando $w \notin L(M)$);
- se R accetta (cioè, M si ferma su w), allora simula M finché M si arresta su w.

Se M ha accettato, allora S accetta l'input $\langle M, w \rangle$ ($w \in L(M)$); se M ha rifiutato, allora S rifiuta l'input $\langle M, w \rangle$ ($w \notin L(M)$). In definitiva, S accetta $\langle M, w \rangle$ se e solo se $\langle M, w \rangle \in A_{TM}$.

Se esistesse R che decide $HALT_{TM}$, allora otterremmo S che decide A_{TM} . Poiché sappiamo che A_{TM} è indecidibile, allora R non può esistere e $HALT_{TM}$ deve essere indecidibile (contraddizione).

In generale, lo **schema di riduzione** dal problema A al problema B si utilizza come segue:

1. Sappiamo che il problema noto A risulta indecidibile;
2. Vogliamo provare che B è indecidibile;
3. Assumiamo (per assurdo) B decidibile ed usiamo questa assunzione per provare A decidibile;
4. La contraddizione ci fa concludere che B è indecidibile.

Di seguito un riepilogo per l'esempio della prova dell'indecidibilità di $HALT_{TM}$:

1. *Sappiamo che il problema noto A risulta indecidibile.* Abbiamo posto A = A_{TM} ;
2. *Vogliamo provare che B è indecidibile.* $HALT_{TM}$ gioca il ruolo di B;
3. *Assumiamo (per assurdo) B decidibile ed usiamo questa assunzione per provare A decidibile.* Proviamo che se $HALT_{TM}$ è decidibile allora A_{TM} è decidibile;
4. *La contraddizione ci fa concludere che B è indecidibile.* Giungiamo alla **contraddizione**.

Una funzione $f: \Sigma^* \rightarrow \Sigma^*$ è calcolabile se esiste una MdT M tale che, su ogni input w, M si arresta con $f(w)$ (e solo con $f(w)$) sul suo nastro.
Cioè, una funzione è calcolabile se esiste una Macchina di Turing che la calcola.

Esempio:

Le seguenti funzioni aritmetiche sono calcolabili (dove $n, m \in \mathbb{N}$):

- $incr(n) = n + 1$;
- $dec(n) = \begin{cases} n - 1 & \text{se } n > 0 \\ 0 & \text{se } n = 0 \end{cases}$;
- $(m, n) \rightarrow m + n$;
- $(m, n) \rightarrow m - n$;
- $(m, n) \rightarrow m \cdot n$.

Un linguaggio A è riducibile a un linguaggio B ($A \leq_m B$) se esiste una funzione calcolabile $f: \Sigma^* \rightarrow \Sigma^*$ tale che $\forall w, w \in A \Leftrightarrow f(w) \in B$.

Una riduzione fornisce un modo per convertire problemi di appartenenza ad A in problemi di appartenenza a B. Se un problema A è riducibile a B, e sappiamo risolvere B, allora sappiamo risolvere A: ciò implica che A "non è più difficile" di B.

Teorema:

Se $A \leq_m B$ e B è decidibile, allora A è decidibile.

Dimostrazione:

Siano M il decider per B ed f la riduzione da A a B. Costruiamo un decider N per A che, su input w:

- calcola f(w);
- "utilizza" M su f(w) e dà lo stesso output.

A questo punto: $w \in A \Leftrightarrow f(w) \in B$ (perché f è una riduzione da A a B) $\Leftrightarrow M \text{ accetta } f(w)$. Quindi, N decide A.

Dunque: $w \in A \Rightarrow M \text{ accetta } f(w) \Rightarrow N \text{ accetta } w$.

D'altronde: $w \notin A \Rightarrow f(w) \notin B \Rightarrow \begin{cases} M \text{ non accetta } w \\ N \text{ non accetta } w \end{cases}$.

Teorema:

Se $A \leq_m B$ e B è Turing riconoscibile, allora A è Turing riconoscibile.

Dimostrazione:

Siano R_A un riconoscitore per A e R_B un riconoscitore per B; allora:

$$R_A: w \rightarrow [f] \rightarrow f(w) \rightarrow [R_B].$$

Corollario:

Se $A \leq_m B$ e A è indecidibile, allora B è indecidibile.

Dimostrazione:

Se B fosse decidibile lo sarebbe anche A, in virtù del teorema precedente.

Corollario:

Se $A \leq_m B$ e A non è Turing riconoscibile, allora B non è Turing riconoscibile.

Dimostrazione:

Se B fosse Turing riconoscibile lo sarebbe anche A , in virtù del teorema precedente.

Esempio 1:

Siano definiti i seguenti linguaggi:

$$\begin{aligned} A_{TM} &= \{\langle M, w \rangle \mid M \text{ è una MdT e } w \in L(M)\}, \\ E_{TM} &= \{\langle M \rangle \mid M \text{ è una MdT e } L(M) = \emptyset\}. \end{aligned}$$

Un esempio di riduzione è $A_{TM} \leq_m \overline{E_{TM}}$. Il linguaggio E_{TM} prende in input una MdT M e si pone come domanda “ $L(M) = \emptyset$?”. Questa riduzione ci dirà che decidere se il linguaggio di una MdT è vuoto è un problema indecidibile. Dobbiamo far vedere che esiste la funzione di riduzione, ossia una funzione che mappa stringhe del primo linguaggio in stringhe del secondo linguaggio e, per ogni stringa che non appartiene al primo linguaggio, il risultato sarà una stringa che non appartiene al secondo linguaggio.

Consideriamo $f: \Sigma^* \rightarrow \Sigma^*$ tale che $f(\langle M, w \rangle) = \langle M_1 \rangle$, dove M_1 su input x :

1. Se $x \neq w$, allora M_1 si ferma e rifiuta x ;
2. Se $x = w$, allora M_1 simula M su w e accetta x se M accetta w .

f è una riduzione di A_{TM} a $\overline{E_{TM}}$?

Abbiamo che:

$$\langle M, w \rangle \in A_{TM} \Rightarrow w \in L(M) \Rightarrow \begin{cases} x \notin L(M_1) & \text{se } x \neq w \\ w \in L(M_1) & \text{se } x = w \end{cases} \Rightarrow \begin{cases} L(M_1) \neq \emptyset \\ \langle M_1 \rangle \in E_{TM} \end{cases}.$$

In questo punto, si ha che la funzione definita mappa una stringa del primo linguaggio (A_{TM}) in una stringa $f(\langle M, w \rangle) = \langle M_1 \rangle$ che appartiene al secondo linguaggio; in definitiva, abbiamo mostrato che

$$\langle M, w \rangle \in A_{TM} \Rightarrow f(\langle M, w \rangle) = \langle M_1 \rangle \in \overline{E_{TM}}.$$

Ci resta da mostrare il caso in cui la stringa non appartiene ad A_{TM} :

$$\langle M, w \rangle \notin A_{TM} \Rightarrow w \notin L(M) \Rightarrow \begin{cases} x \in L(M_1) & \text{se } x \neq w \\ x \notin L(M_1) & \text{se } x = w \end{cases} \Rightarrow \begin{cases} L(M_1) = \emptyset \\ f(\langle M, w \rangle) = \langle M_1 \rangle \notin \overline{E_{TM}} \end{cases}.$$

In sintesi, la funzione f è calcolabile, e M accetta w (cioè, $\langle M, w \rangle \in A_{TM}$) se e solo se $L(M_1) \neq \emptyset$ (cioè, se e solo se $\langle M_1 \rangle \in \overline{E_{TM}}$). \square

Per completezza, in base ad uno dei corollari definiti in precedenza, abbiamo che

$$A_{TM} \leq_m \overline{E_{TM}} \text{ e } A_{TM} \text{ indecidibile} \Rightarrow \overline{E_{TM}} \text{ indecidibile.}$$

Quindi, anche E_{TM} è indecidibile (*la decidibilità non risente dalla complementazione*).

Nota: Non si conosce una riduzione da A_{TM} a E_{TM} .

Esempio 2:

Siano definiti i linguaggi:

$$\begin{aligned} A_{TM} &= \{\langle M, w \rangle \mid M \text{ è una MdT e } w \in L(M)\}, \\ REGULAR_{TM} &= \{\langle M \rangle \mid M \text{ è una MdT e } L(M) \text{ è regolare}\}. \end{aligned}$$

Un esempio di riduzione è $A_{TM} \leq_m REGULAR_{TM}$. Il linguaggio $REGULAR_{TM}$ contiene tutte le descrizioni di MdT il cui linguaggio è regolare (cioè, si ricordi, se il linguaggio è accettato da qualche automa finito). Questo linguaggio prende in input una MdT M e si pone come domanda “ $L(M)$ è regolare?”. Questa riduzione ci dirà che decidere se il linguaggio di una MdT è regolare è un problema indecidibile.

Non abbiamo alcuna restrizione su come costruire la funzione f . L'unica cosa importante è che questa funzione sia calcolabile.

Consideriamo la funzione $f: \langle M, w \rangle \rightarrow \langle R \rangle$ come riduzione da A_{TM} a $REGULAR_{TM}$, dove R su un input x :

1. Se $x \in \{0^n 1^n \mid n \in \mathbb{N}\}$, allora R si ferma e accetta x ;
2. Se $x \notin \{0^n 1^n \mid n \in \mathbb{N}\}$, allora R simula M su w e accetta x se M accetta w .

Posto $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ per brevità, abbiamo che:

$$\langle M, w \rangle \in A_{TM} \Rightarrow M \text{ accetta } w \Rightarrow \begin{cases} \text{accettata da } R \text{ (per def.)} & \text{se } x \in L \\ \text{accettata da } R & \text{se } x \notin L \end{cases} \Rightarrow L(R) = \Sigma^*,$$

posto $\Sigma = \{0, 1\}$. Sapendo che $\Sigma^* (= L(R))$ è regolare, allora $f(\langle M, w \rangle) = \langle R \rangle \in REGULAR_{TM}$.

Ci resta da mostrare l'implicazione inversa:

$$\langle M, w \rangle \notin A_{TM} \Rightarrow M \text{ non accetta } w \Rightarrow \begin{cases} \text{accettata da } R \text{ (per def.)} & \text{se } x \in L \\ \text{non accettata da } R & \text{se } x \notin L \end{cases} \Rightarrow L(R) = L,$$

il quale non è regolare (mostrato in passato con il Pumping Lemma).

Questo fatto implica che $f(\langle M, w \rangle) = \langle R \rangle \notin REGULAR_{TM}$. A questo punto, abbiamo dimostrato le due implicazioni.

In sintesi, la funzione f è calcolabile, e:

- $L(R) = \Sigma^*$ (regolare) se M accetta w ;
- $L(R) = \{0^n 1^n \mid n \in \mathbb{N}\}$ (non regolare) altrimenti.

Esempio 3:

Siano definiti i linguaggi:

$$\begin{aligned} E_{TM} &= \{\langle M \rangle \mid M \text{ è una MdT e } L(M) = \emptyset\}, \\ EQ_{TM} &= \{\langle M_1, M_2 \rangle \mid M_1, M_2 \text{ sono MdT e } L(M_1) = L(M_2)\}. \end{aligned}$$

Un esempio di riduzione è $E_{TM} \leq_m EQ_{TM}$. Il linguaggio EQ_{TM} prende in input due MdT M_1 e M_2 e si pone come domanda “ $L(M_1) = L(M_2)$?”. Questa riduzione ci dirà che decidere se il linguaggio di due MdT sono uguali è un problema indecidibile.

Consideriamo la funzione $f: \langle M \rangle \rightarrow \langle M_1, M_2 \rangle$ come riduzione da E_{TM} a EQ_{TM} , dove $\langle M \rangle \in E_{TM}$ se e solo se $f(\langle M \rangle) \in EQ_{TM}$; occorre dimostrare che la funzione è calcolabile, e che vale il sse precedente.

Sia M_1 una MdT tale che $L(M_1) = \emptyset$. La funzione sarà $f: \langle M \rangle \rightarrow \langle M, M_1 \rangle$, la quale è calcolabile, e bisogna far vedere che effettivamente questa è una riduzione di E_{TM} a EQ_{TM} .

Mostriamo le implicazioni:

$$\begin{aligned} \langle M \rangle \in E_{TM} &\Rightarrow L(M) = \emptyset \Rightarrow L(M) = L(M_1) \Rightarrow \langle M, M_1 \rangle = f(\langle M \rangle) \in EQ_{TM}, \\ \langle M \rangle \notin E_{TM} &\Rightarrow L(M) \neq \emptyset \Rightarrow L(M_1) \neq \emptyset \Rightarrow \langle M, M_1 \rangle = f(\langle M \rangle) \notin EQ_{TM}. \end{aligned}$$

Esempio 4:

Siano definiti i linguaggi:

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ è una MdT e } w \in L(M)\},$$

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2 \text{ sono MdT e } L(M_1) = L(M_2)\}.$$

Un esempio di riduzione è $A_{TM} \leq_m EQ_{TM}$. Procediamo come nell'esempio precedente.

Consideriamo la funzione $f: \langle M, w \rangle \rightarrow \langle M_1, M_2 \rangle$ come riduzione da A_{TM} a EQ_{TM} , dove $\langle M, w \rangle \in A_{TM}$ se e solo se $f(\langle M, w \rangle) \in EQ_{TM}$; occorre dimostrare che la funzione è calcolabile, e che vale il sse precedente.

L'idea è la seguente. Data $\langle M, w \rangle$, consideriamo le MdT M_1 e M_2 tali che:

- M_1 accetta x (qualsiasi input) $\Rightarrow L(M_1) = \Sigma^*$;
- M_2 simula M su w . Se M accetta w , allora M_2 accetta $x \Rightarrow L(M_2) = M$.

La funzione è calcolabile, in quanto costruire una macchina che accetta tutte le stringhe è semplice, così come la seconda parte di f è M stessa, quindi è l'input (basta concatenare la descrizione di M_1 a M stessa, ed otteniamo il risultato della funzione). Ora, mostriamo che la funzione f è una riduzione.

Mostriamo le implicazioni:

$$\langle M, w \rangle \in A_{TM} \Rightarrow w \in L(M) \Rightarrow \begin{cases} L(M_1) = \Sigma^* \\ L(M_2) = \Sigma^* \end{cases} \Rightarrow L(M_1) = L(M_2) \Rightarrow \langle M_1, M_2 \rangle = f(\langle M, w \rangle) \in EQ_{TM},$$

$$\langle M, w \rangle \notin A_{TM} \Rightarrow w \notin L(M) \Rightarrow \begin{cases} L(M_1) = \Sigma^* \\ L(M_2) = \emptyset \end{cases} \Rightarrow L(M_1) \neq L(M_2) \Rightarrow \langle M_1, M_2 \rangle = f(\langle M, w \rangle) \notin EQ_{TM}.$$

In sintesi, $f: \langle M, w \rangle \rightarrow \langle M_1, M_2 \rangle$ è una riduzione da A_{TM} a EQ_{TM} .

Teorema:

EQ_{TM} non è né Turing riconoscibile né co-Turing riconoscibile.

Dimostrazione:

Supponiamo per assurdo che EQ_{TM} sia Turing riconoscibile. Allora

$$A_{TM} \leq_m \overline{EQ_{TM}} \Rightarrow \overline{A_{TM}} \leq_m \overline{EQ_{TM}}.$$

Quindi $\overline{A_{TM}}$ sarebbe Turing riconoscibile: questo è assurdo.

Supponiamo per assurdo che EQ_{TM} sia co-Turing riconoscibile, cioè che $\overline{EQ_{TM}}$ sia Turing riconoscibile. Allora

$$A_{TM} \leq_m EQ_{TM} \Rightarrow \overline{A_{TM}} \leq_m \overline{EQ_{TM}}.$$

Quindi $\overline{A_{TM}}$ sarebbe Turing riconoscibile: questo è assurdo.

6.1 TEOREMA DI RICE

Questo teorema è uno strumento generale che permette di stabilire l'indecidibilità di una vasta classe di problemi.

Afferma che, per ogni proprietà non banale delle funzioni calcolabili, il problema di decidere quali funzioni soddisfano tale proprietà e quali no, è indecidibile.

Proprietà banale:

Proprietà che non effettua alcuna discriminazione tra le funzioni calcolabili, cioè che vale o per tutte o per nessuna.

Formalizzato in termini di linguaggio, e quindi decidibilità di linguaggi:

Teorema di Rice:

Sia $L_P = \{\langle M \rangle \mid M \text{ è una MdT che verifica la proprietà } P\}$ un linguaggio che soddisfa le seguenti due condizioni:

1. L'appartenenza di M a L_P dipende solo da $L(M)$, cioè:
 $\forall M_1, M_2 \text{ MdT tali che } L(M_1) = L(M_2), \langle M_1 \rangle \in L_P \leftrightarrow \langle M_2 \rangle \in L_P$
2. L_P è un problema non banale, cioè:
 $\exists M_1, M_2 \text{ MdT tali che } \langle M_1 \rangle \in L_P, \langle M_2 \rangle \notin L_P$
allora L_P è indecidibile.

Quindi, ogni proprietà non banale del linguaggio di una MdT è indecidibile.

Informalmente, il punto 1) significa che la proprietà P è una proprietà del linguaggio della MdT in considerazione, significa che se 2 MdT hanno lo stesso linguaggio, per ogni coppia M_1 e M_2 MdT tali che $L(M_1) = L(M_2)$, allora entrambe appartengono al linguaggio L_P o nessuna delle due appartiene. Il punto 2) significa che (non banale significa che non vale né per tutte le macchine né per nessuna macchina) deve esistere almeno una MdT che gode della proprietà P ed almeno una MdT che non gode della proprietà P . In termini del linguaggio L_P devono esistere 2 MdT M_1 ed M_2 dove la descrizione di M_1 appartiene al linguaggio e la descrizione di M_2 non appartiene al linguaggio.

Ricapitolando:

Se abbiamo un insieme di descrizioni di MdT che soddisfano la data proprietà, e quest'ultima è non banale che dipende solo dal linguaggio della macchina e non dalla macchina stessa, allora si ha che il linguaggio L_P è indecidibile. Il che significa che ogni proprietà non banale del linguaggio di una MdT, la verifica di P è un problema indecidibile.

Nota: La differenza tra una proprietà di $L(M)$ e una proprietà di M è che:

Proprietà del linguaggio significa che dipende dal linguaggio, $L(M)$, quindi dalle stringhe accettate dalla macchina e non dalla macchina M .

Esempio:

- $L(M) = \emptyset$ è una proprietà del linguaggio;
- “ M ha almeno 1000 stati” è una proprietà della MdT;

Quindi, “ $L(M) = \emptyset$ ” è indecidibile, mentre “ M ha almeno 1000 stati” è facilmente decidibile, basta guardare alla codifica di M e contare.

Dimostrazione Teorema di Rice:

Vogliamo dimostrare che se abbiamo una proprietà P e consideriamo il linguaggio di tutte le descrizioni di MdT che verificano la P , allora il linguaggio $L_P = \{\langle M \rangle \mid M \text{ è una MdT che verifica la proprietà } P\}$ gode delle due proprietà del teorema, L_P è indecidibile.

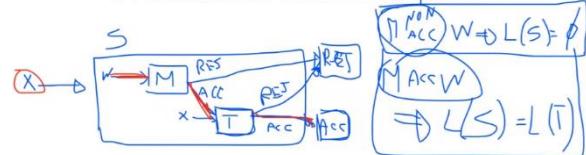
Mostriamo $A_{TM} \leq_m L_P$, e mostriamo una funzione calcolabile $\forall x \in A_{TM}$ sse $f(x) \in L_P$, quindi andiamo a trasformare la nostra riduzione in una funzione calcolabile che ci permette di trasformare elementi di A_{TM} in elementi di L_P , mentre se invece abbiamo una stringa che non appartiene a A_{TM} allora non otteniamo un elemento di L_P . In A_{TM} abbiamo coppie $\langle M, w \rangle$ e in L_P abbiamo delle MdT M' , quindi la funzione deve trasformare una coppia $\langle M, w \rangle$ in una stringa che è una descrizione di una macchina M' , dove $\langle M, w \rangle \in A_{TM}$ sse $M' \in L_P$. Formalmente: $f: \langle M, w \rangle \rightarrow \langle M' \rangle$.

Andiamo ad indicare con T_0 una MdT tale che $L(T_0) = \emptyset$, a questo punto possiamo assumere che $\langle T_0 \rangle \notin L_P$, altrimenti si potrebbe procedere con $\overline{L_P}$.

Poiché L_P è non banale, per ipotesi del teorema di Rice, esiste una MdT T tale che $\langle T \rangle \in L_P$ (ed esiste una MdT tale che $\langle T \rangle \notin L_P$).

A questo punto, la funzione f sarà: $f(\langle M, w \rangle) = \langle S \rangle$, dove S è una MdT su input x :

- Simula M con input w :
 - Se M si ferma e rifiuta, allora S rifiuta x ;
 - Se M accetta, allora S simula T su input x :
se T accetta x allora S accetta, se T rifiuta x allora S rifiuta.



Quindi se M rifiuta w il risultato sarà reject, mentre se M accetta w la macchina chiede cosa fa T su input x , quello che abbiamo è che se M non accetta w allora il linguaggio di S sarà \emptyset , $L(S) = \emptyset$. Ma se M accetta w il linguaggio accettato da S sarà il linguaggio accettato da T , $L(S) = L(T)$.

Abbiamo quindi un doppio funzionamento, andando a creare due macchine con due linguaggi ben precisi che dipendono dal fatto che se la coppia $\langle M, w \rangle$ appartiene o meno ad A_{TM} .

Mostriamo che la funzione **f è una riduzione**:

- f è calcolabile, perché abbiamo costruito da M e w la macchina S , come se S fosse un programma che utilizza due sottoprogrammi, uno M e uno T .
- $\langle M, w \rangle \in A_{TM} \iff \langle S \rangle \in L_P$, è vero perché:
 $(\Rightarrow) \langle M, w \rangle \in A_{TM} \rightarrow w \in L(M) \rightarrow L(S) = L(T)$, ma $T \in L_P$ e sfruttando la *proprietà 1 del teorema di Rice* $\rightarrow S \in L_P$,
 $(\Leftarrow) \langle M, w \rangle \notin A_{TM} \rightarrow M \text{ non accetta } w \rightarrow L(S) = \emptyset = L(T_0) \rightarrow \text{ma sappiamo per ipotesi } \langle T_0 \rangle \notin L_P \text{ e proprietà 1 del teorema di Rice} \rightarrow S \notin L_P$,

Poiché sappiamo che A_{TM} è indecidibile allora L_P è indecidibile.

Conseguenze del Teorema di Rice:

Si può dimostrare col teorema di Rice che non possiamo decidere se una MdT:

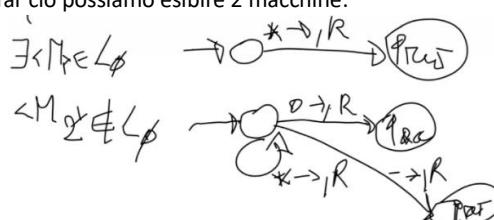
- Accetta \emptyset :

Esempio:

$L_\emptyset = \{\langle M \rangle \mid M \text{ è TM tale che } L(M) = \emptyset\}$, quindi la proprietà P dal teorema di Rice sarà $P: L(M) = \emptyset$ (non banale perché dipende solo dal linguaggio).

Verifichiamo adesso questa proprietà tramite i 2 punti del teorema di Rice:

1. $\forall M_1, M_2 \text{ MdT tali che } L(M_1) = L(M_2), \langle M_1 \rangle \in P \leftrightarrow \langle M_2 \rangle \in P$, se queste due macchine hanno lo stesso linguaggio entrambe sono il $L(M) = \emptyset$ o nessuna:
Io vediamo perché se $L(M_1) = L(M_2) = \emptyset$ e quindi $\langle M_1 \rangle, \langle M_2 \rangle \in L_\emptyset$, mentre se $L(M_1) = L(M_2) \neq \emptyset$ e quindi $\langle M_1 \rangle, \langle M_2 \rangle \notin L_\emptyset$.
2. $\exists M_1, M_2 \text{ MdT tali che } \langle M_1 \rangle \in P, \langle M_2 \rangle \notin P$, per far ciò possiamo esibire 2 macchine:



E quindi le condizioni 1 e 2 ci dicono che L_\emptyset è indecidibile, senza necessità della funzione calcolabile.

- Accetta un linguaggio finito:
- Accetta un linguaggio regolare
- ...

7. TEORIA DELLA COMPLESSITÀ

Con lo studio precedente, ovvero quello della **calcolabilità**, si è visto che nell'insieme di tutti i problemi vi sono alcuni che possiamo risolvere col calcolatore ed alcuni non risolvibili in questo modo, in più questo studio non considera la **dificoltà** dei problemi, distingue solo ciò che è risolubile da ciò che non lo è.

Anche quando un problema è decidibile, e quindi computazionalmente risolvibile, può non essere risolvibile praticamente, se la soluzione richiede una quantità eccessiva di tempo o di memoria.

Lo studio della **complessità** considera **solo problemi solubili**, allo scopo di fornire una caratterizzazione dal punto di vista della **quantità di risorse di calcolo necessario a risolverli** (chiamata **dificoltà di risoluzione**), qualunque algoritmo ha bisogno di una quantità minima di risorse.

Le **risorse** di cui si tiene principalmente conto quando si scrivono o si utilizzano programmi sono relative al **tempo** e allo **spazio** (ma non solo le uniche risorse usate durante il calcolo).

Ci limiteremo a considerare **solo il tempo** utilizzato per la soluzione di un problema e considereremo **problem di decisione**, quelli che hanno come soluzione una risposta sì o no, e quelli che sono **decidibili**.

Ricorda:

Dato un problema decidibile lo possiamo esprimere come un linguaggio decidibile, formalmente $P \Leftrightarrow L_P$, e quello che dobbiamo considerare per andare a discutere di tempo di esecuzione è la dimensione dell'input, ad esempio se dobbiamo ordinare una lista di 10 elementi avremo bisogno di un determinato tempo, ma se gli elementi fossero milioni avremmo bisogno di un tempo diverso, quindi il tempo necessario non è un tempo assoluto ma dipende dalla dimensione dell'input.

Quando andiamo a considerare un linguaggio, la dimensione in questo caso è la lunghezza della stringa che rappresenta l'input, ovvero $|x|$.

Esempio:

Se prendiamo un problema che prende in input un grafo e chiede se G è un grafo connesso sappiamo che il linguaggio associato a questo problema è l'insieme di tutte le stringhe che rappresentano un grafo G dove questo è connesso. La dimensione del problema per un grafo normalmente è il numero dei nodi, e quindi nella rappresentazione di G , la stringa che rappresenta il grafo avrà una lunghezza proporzionale al numero di nodi e quindi quello che ci interessa è la lunghezza della stringa che rappresenta il grafo in input.

G è un grafo connesso $\Leftrightarrow \{(G) | G \text{ è un grafo connesso}\}$

Dimensione di G (numero nodi) $\Leftrightarrow |(G)|$

Complessità temporale:

Andremo a considerare la quantità di tempo necessaria all'esecuzione di un programma per risolvere un certo problema, tecnicamente è chiamato **complessità temporale**.

Il numero di passi che utilizza un algoritmo su un particolare input può dipendere da diversi parametri, per semplicità, si calcola il tempo di esecuzione di un algoritmo semplicemente in funzione della lunghezza della stringa che rappresenta l'input e non si considerano eventuali altri parametri.

Nell'analisi del **caso peggiore**, che noi considereremo ovvero **O-grande**, valuta il tempo di esecuzione massimo tra tutti gli input di una determinata lunghezza. Per utilizzare questa analisi asintotica, lo facciamo prendendo in considerazione solo il termine di ordine maggiore dell'espressione del tempo di esecuzione dell'algoritmo, trascurando sia il coefficiente di tale termine, che tutti i termini di ordine inferiore, perché il termine di ordine più alto domina gli altri termini quando l'input è grande.

Possiamo formalizzare tutto ciò nella seguente definizione:

Sia R^+ l'insieme dei numeri reali non negativi e siano f e g funzioni $f, g: N \rightarrow R^+$.

Si dice che $f(n)=O(g(n))$ se esistono interi positivi c e n_0 tali che per ogni $n \geq n_0$, risulti $f(n) \leq cg(n)$.

Quando $f(n)=O(g(n))$, diciamo che $g(n)$ è un **limite superiore** per $f(n)$, o più precisamente che $g(n)$ è un **limite superiore asintotico** per $f(n)$, per sottolineare che stiamo ignorando le costanti.

Esempio:

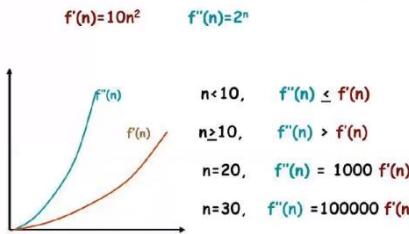
La funzione $f(n) = 6n^3 + 2n^2 + 20n + 45$ ha quattro termini e quello di ordine maggiore è $6n^3$, trascuriamo il coefficiente 6 e diciamo che f è asintoticamente al più n^3 . In generale, se si ha un polinomio vale $a_c n^c + \dots + a_1 n + a_0 = O(n^c)$.

Quindi la notazione asintotica (**O-grande**) per descrivere questo rapporto è $f(n)=O(n^3)$.

Altro esempio può essere: $3n \log_2 n + 5n \log_2 \log_2 n + 2 = O(n \log n)$.

La nostra divisione fondamentale sarà tra **polinomiale** (indipendentemente dall'esponente) ed **esponenziale** (indipendentemente dalla base). È importante fare queste due distinzioni perché la variazione di n può essere molto significativa nei due casi, ad esempio:

Confronto di complessità Dato un problema consideriamo 2 algoritmi A e B con compl. $f'(n)$ e $f''(n)$



Espressione O	Nome informale
$O(1)$	costante
$O(\log n)$	logaritmico (base?)
$O(n)$	lineare
$O(n^2)$	quadratico
$O(n^3)$	cubico
$O(n^k)$, $k \geq 1$	polinomiale
$O(d^n)$, $d \geq 2$	esponenziale

Espressione O	Nome informale
$O(n^k)$, $k \geq 1$	polinomiale
$O(d^n)$, $d \geq 2$	esponenziale

7.1 COMPLESSITÀ TEMPORALE DI UNA MACCHINA DI TURING

Sia $M = (Q, \Sigma, \Gamma, f, q_0, q_{\text{accept}}, q_{\text{reject}})$ una MdT deterministica, che si arresta su ogni input. La complessità temporale di M è la funzione $f: N \rightarrow N$ dove $f(n)$ è il massimo numero di passi eseguiti da M su un qualsiasi input di lunghezza n , per ogni $n \in N$.

Cioè $f(n) = \text{massimo numero di passi in } q_0 w \rightarrow^* uqv, q \in \{q_{\text{accept}}, q_{\text{reject}}\}$, al variare di w in Σ^n .

Se M ha complessità temporale $f(n)$, diremo che M decide $L(M)$ in tempo $f(n)$.

La complessità temporale dipende dalla codifica utilizzata, codificare un numero intero n in base 2 richiede $\lceil \log n + 1 \rceil$ (= più piccolo intero $\geq n + 1$) cifre binarie, mentre codificarlo in base unaria richiede n cifre unarie. Essendo l'input più lungo, la macchina ha più tempo a disposizione.

Avere una complessità temporale $O(n)$ rispetto alla codifica unaria, può voler dire che la complessità temporale rispetto alla codifica binaria sia $O(2^n)$.

Esempio:

PRIMO: Dato un numero intero x , x è primo?

Algoritmo semplice: dividì x per tutti gli interi $i < x$. Se tutti i resti di tali divisioni sono diversi da zero, x è primo.

Richiede: ponendo $n = |\langle x \rangle|$

- $O(n)$ passi se x è rappresentato in unario,
- $O(2^n)$ passi se x è rappresentato in base 2.

Quindi se ho un intero 1000, non vale mille ma lo rappresento in binario nella macchina e quindi ho bisogno non di mille bit ma della lunghezza della stringa che me lo rappresenta è lunga dieci, e quindi quando vado a considerare il numero di passi, se faccio 10 passi ho una complessità lineare, se invece faccio mille passi ho una complessità 2^{10} e quindi è esponenziale. Quindi per gli interi consideriamo una codifica in base 2 o una qualsiasi altra pur che si maggiore o uguale a 2. Per i grafi, che dobbiamo rappresentare nodi e archi, in generale possiamo rappresentarli mediante liste delle adiacenze o matrici di adiacenze. La rappresentazione per insiemi, relazioni e funzioni mediante enumerazione delle codifiche dei relativi elementi. Codifiche "ragionevoli" dei dati sono **polinomialmente correlate**: è possibile passare da una di esse a una qualunque altra codifica "ragionevole" delle istanze dello stesso problema in un tempo polinomiale rispetto alla rappresentazione originale.

Le varianti di macchine di Turing deterministiche possono simularsi tra di loro con un sovraccarico computazionale **polinomiale**.

Anche gli altri modelli di calcolo possono simularsi con un sovraccarico computazionale polinomiale, ad eccezione di quelle **non deterministiche**.

Esaminiamo come la scelta del modello di calcolo può influire sulla complessità di tempo dei linguaggi.

Theorema:

Sia $t(n)$ una funzione tale che $t(n) \geq n$. Per ogni MdT multinastro M con complessità temporale $t(n)$ esiste una MdT a nastro singolo M' con complessità temporale $O(t^2(n))$.

Analizziamo i passi delle due macchine per determinare la quantità di tempo supplementare richiesta.

Dimostrazione:

Sia M una TM a k -nastri avente tempo di esecuzione $t(n)$, costruiamo una TM S a singolo nastro che ha tempo di esecuzione $O(t^2(n))$. La macchina S opera simulando M . Nel rivedere tale simulazione, ricordiamo che S utilizza il suo unico nastro per rappresentare il contenuto di tutti i k nastri di M . Inizialmente, S mette il nastro nel formato che rappresenta tutti i nastri di M e poi simula i passi di M . Per simulare un passo, S scorre tutte le informazioni memorizzate sul suo nastro per determinare i simboli presenti sotto le testine di M . Poi S esegue un'altra scansione del suo nastro per aggiornare il contenuto dei nastri e le posizioni delle testine. Se una testina di M si sposta a destra su una parte non letta del nastro, S deve aumentare la quantità di spazio allocato per questo nastro. Lo fa spostando una parte del suo nastro di una cella a destra.

Da quanto detto, per ogni passo di M , la macchina S fa due passi sulla parte attiva del suo nastro. Il primo ottiene le informazioni necessarie per determinare la prossima mossa e il secondo la esegue. La lunghezza della parte attiva del nastro di S determina il tempo che S impiega per eseguire la scansione, quindi dobbiamo determinare un limite superiore per questa lunghezza. Per farlo prendiamo la somma delle lunghezze delle parti attive dei k nastri di M . Ciascuna di queste parti attive ha lunghezza al più $t(n)$ poiché M utilizza $t(n)$ celle del nastro in $t(n)$ passi, se la testina si sposta verso destra ad ogni passo e anche meno se vi sono spostamenti di qualche testina a sinistra. Quindi una scansione della parte di nastro attiva di S impiega $O(t(n))$ passi. Per simulare ciascuna delle fasi di M , S esegue due scansioni ed eventualmente fino a k spostamenti a destra. Ognuno impiega un tempo $O(t(n))$, per cui il tempo totale per S per simulare un passo di M è $O(t(n))$. Ora possiamo limitare il tempo totale impiegato dalla simulazione. La fase iniziale, in cui S mette il nastro nel formato corretto, usa $O(n)$ passi. In seguito, S simula ciascuno dei $t(n)$ passi di M , utilizzando $O(t(n))$ passi, per cui questa parte della simulazione utilizza $t(n) \times O(t(n)) = O(t^2(n))$ passi. Quindi l'intera simulazione di M utilizza $O(n) + O(t^2(n))$ passi.

Abbiamo assunto che $t(n) \geq n$ (un'ipotesi ragionevole perché M non potrebbe nemmeno leggere l'intero input in meno tempo). Pertanto, il tempo di esecuzione di S è $O(t^2(n))$, quindi **MdT multinastro e a singolo nastro non fa variare il tipo di complessità**.

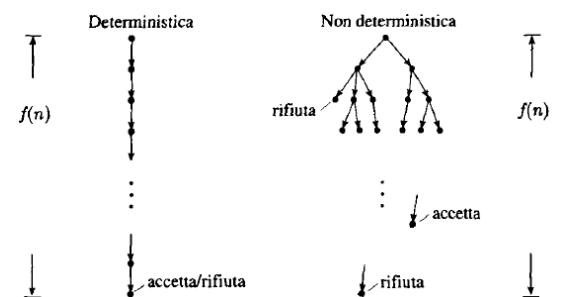
Theorema:

Sia $t(n)$ una funzione tale che $t(n) \geq n$. Per ogni MdT a nastro singolo, non deterministica N e con complessità temporale $t(n)$ esiste una MdT a nastro singolo, deterministica e con complessità temporale $2^{O(t(n))}$.

Dimostrazione:

Sia N una TM non deterministica avente tempo di esecuzione $t(n)$. Costruiamo una TM deterministica D che simula N effettuando una ricerca sull'albero delle computazioni di N .

Su un input di lunghezza n , ogni ramificazione dell'albero delle computazioni di N ha lunghezza al più $t(n)$. Ogni nodo dell'albero può avere al più b figli, dove b è il massimo numero di scelte possibili in accordo alla funzione di transizione di N . Così il numero totale di foglie nell'albero è al massimo $b^{t(n)}$. La simulazione procede esplorando l'albero prima in ampiezza. In altre parole, si visitano tutti i nodi a profondità d prima di passare ad uno qualsiasi dei nodi a profondità $d+1$. Inizia inefficientemente dalla radice e si sposta in basso verso un nodo ogni volta che visita il nodo stesso. Tuttavia, l'eliminazione di tale inefficienza non altera l'enunciato del Teorema, quindi la lasciamo in questa forma. Il numero totale di nodi dell'albero inferiore al doppio del numero di foglie, quindi è limitato da $O(b^{t(n)})$. Il tempo per partire dalla radice e raggiungere un nodo è $O(t(n))$. Pertanto, il tempo di esecuzione di D è $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. Come descritto dal teorema che afferma che per una MdT non deterministica esiste una MdT deterministica equivalente, la MdT D ha tre nastri. Convertirla in una a singolo nastro al più fa sì che si elevi al quadrato il tempo di esecuzione, per il teorema precedente. Quindi il tempo di esecuzione di quella a singolo nastro è $(2^{O(t(n)})^2 = 2^{O(2t(n))} = 2^{O(t(n))})^2$, quindi **non manteniamo la complessità**.



7.2 CLASSE P

Vogliamo definire classi chiuse rispetto al cambio del modello di calcolo utilizzato e al cambio di rappresentazione dei dati.

La classe P è l'insieme dei linguaggi L per i quali esiste una MdT M con un solo nastro che decide L e per cui $t_M(n) = O(n^k)$ per qualche $k \geq 1$.

Ricordando la tesi di Church-Turing il quale afferma che tutto quello che risulta computabile può essere computato da una MdT deterministica.

La versione forte di questa tesi afferma la **correlazione polinomiale** nel tempo tra algoritmi e computazione di una MdT deterministica.

Quindi, possiamo definire P come l'insieme dei problemi computazionali che ammettono un **algoritmo polinomiale**, i così detti problemi "**trattabili**".

I problemi che sono risolvibili in teoria, ma non possono essere risolti nella pratica, sono chiamati "**intrattabili**", ad esempio al suo interno troviamo diversi problemi che richiedono almeno $n^{1000000}$ operazioni, ma contiene anche problemi naturali come determinare se un numero è primo.

8. COMPLESSITÀ DEGLI ALGORITMI

La **complessità** ha lo scopo di dividere i problemi in grandi classi che dipendono dal fatto che esista o meno un algoritmo polinomiale per la soluzione.

Le classi che abbiamo visto e che vedremo sono:

- **Indecidibilità**, ovvero che non esiste algoritmo che risolvi un dato problema;
- **Classe P**, insieme di problemi per cui esiste un algoritmo polinomiale che li risolve, la cui complessità di tempo è $O(n^k)$;
- **NP-Completezza**, non si conosce un algoritmo polinomiale e non si riesce a dire che l'algoritmo esiste o meno per un dato problema (*spoiler!!!*).

I problemi che possiamo risolvere, *in pratica*, sono quelli che ammettono algoritmi aventi **tempo polinomiale**.

Se abbiamo problemi con tempo polinomiale, gli algoritmi utilizzati possono essere usati anche per risolvere istanze grandi di un altro problema. Da questo abbiamo definito la classe P come classe di problemi che hanno algoritmi efficienti (ovvero con *tempo polinomiale*), per quanto riguarda gli altri problemi non appartenenti a P, non sono noti algoritmi che risolvono tali problemi in tempo polinomiale, forse questi ammettono algoritmi polinomiali che si basano su principi non ancora noti o semplicemente non possono essere risolti in tale modo.

8.1 RIDUZIONI POLINOMIALI

Supponiamo che possiamo risolvere il problema X in tempo polinomiale, se questo problema X si riduce **efficientemente** (ovvero in tempo polinomiale) al problema Y, una soluzione efficiente per Y può essere usata per risolvere X efficientemente.

Questa riduzione la si dimostra col concetto di "**riduzione polinomiale**", simile alla riduzione fatta in precedenza ma riguarda la complessità di tempo.

In altre parole, il problema X **si riduce in tempo polinomiale** al problema Y se arbitrarie istanze di X possono essere risolte usando un numero polinomiale di passi di computazione, più un numero polinomiale di chiamate ad un oracolo (algoritmo qualsiasi "*black-box*" che risolve istanze di Y in un solo passo) che risolve Y, tutto ciò si denota $X \leq_p Y$.

Nota: le istanze di Y devono avere dimensione polinomiale, in quanto gli oracoli di Y richiedono tempo per elaborare sia l'input che l'output.

Lo **scopo** della riduzione polinomiale è classificare i problemi in accordo alla difficoltà **relativa**.

Se $X \leq_p Y$ e Y ammette soluzione in tempo polinomiale, allora anche X ammette soluzione in tempo polinomiale.

Al contrario, se $X \leq_p Y$ e X non ammette soluzione in tempo polinomiale, allora anche Y non ammette soluzione in tempo polinomiale.

In alcuni casi è possibile stabilire un'uguaglianza tra riduzioni, se $X \leq_p Y$ e $Y \leq_p X$ allora $X \equiv_p Y$.

RIDUZIONE MEDIANTE EQUIVALENZA SEMPLICE:

Per provare tale riduzione introduciamo due problemi NP noti:

INDEPENDENT-SET:

Dato un grafo $G = (V, E)$, diciamo che un insieme di nodi $S \subseteq V$ è indipendente se non ci sono due nodi in S uniti da un arco. Il problema del Independent Set è il seguente: Dato G , trova un Independent Set che sia il più grande possibile. Ad esempio, la dimensione massima di un Independent Set nella Figura in alto a destra è quattro, ottenuta dal Independent Set a quattro nodi $\{1, 4, 5, 6\}$.

Domanda: dato un grafo $G = (V, E)$ e un intero k , esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \geq k$, e per ogni edge, **almeno** uno dei suoi estremi è in S ?

Es: Esiste un insieme indipendente di dimensione ≥ 6 , nella figura in basso a destra? Si.

Es: Esiste un insieme indipendente di dimensione ≥ 7 , nella figura in basso a destra? No.

VERTEX-COVER:

Dato un grafo $G = (V, E)$, diciamo che un sottoinsieme di nodi $S \subseteq V$ è una Vertex Cover se ogni bordo $e \in E$ ha almeno un'estremità (dell'arco) in S .

Si noti che in una Vertex Cover, i vertici fanno la "copertura", e gli archi sono gli oggetti che sono "coperti". Ora, è facile trovare grandi Vertex Cover in un grafo, la parte difficile è trovare quelle piccole.

Ad esempio, nella figura in alto a destra, l'insieme di nodi $\{1, 2, 6, 7\}$ è una Vertex Cover di dimensione 4, mentre l'insieme $\{2, 3, 7\}$ è una Vertex Cover di dimensione 3.

Domanda: dato un grafo $G = (V, E)$ e un intero k , esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \leq k$, e per ogni edge, **al più** uno dei suoi estremi è in S ?

Es: Esiste un vertex-cover di dimensione ≤ 4 , nella figura in basso a destra? Si.

Es: Esiste un vertex-cover di dimensione ≤ 3 , nella figura in basso a destra? No.

Non sappiamo come risolvere questi due problemi in tempo polinomiale, ma cosa possiamo dire della loro relativa difficoltà?

Mostriamo ora che sono equivalentemente difficili, ovvero **VERTEX-COVER \equiv_p INDEPENDENT-SET**.

Dimostrazione:

Mostriamo che S è un Independent-Set sse il suo complemento $V-S$ è un Vertex-Cover. Mostrandolo nelle due direzioni:

(\Rightarrow) INDEPENDENT-SET \leq_p VERTEX-COVER

In primo luogo, supponiamo che S sia un Independent Set. Consideriamo un arco arbitrario $e = (u, v)$, poiché S contiene vertici indipendenti, non è possibile che sia u e sia v siano in S quindi uno di essi deve essere per forza in $V-S$.

Ne consegue che ogni arco ha almeno un'estremità in $V-S$, e quindi $V-S$ è una Vertex Cover.

Riassumendo:

Sia S un qualsiasi independent set, consideriamo un arbitrario edge (u, v) .

S indipendente $\rightarrow u \notin S$ o $v \notin S \rightarrow u \in V-S$ o $v \in V-S$.

Quindi, $V-S$ copre (u, v) .

(\Leftarrow) VERTEX-COVER \leq_p INDEPENDENT-SET

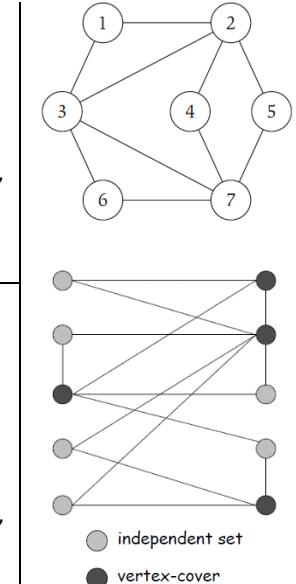
Al contrario, supponiamo che $V-S$ sia una Vertex Cover. Consideriamo due nodi u e v in S , se fossero uniti da un arco e , allora nessuna delle estremità di e risiederebbe in $V-S$, contraddicendo la nostra ipotesi che $V-S$ sia una Vertex Cover.

Ne consegue che due nodi in S non sono uniti da un arco e , quindi S è un Independent Set.

Riassumendo:

Sia $V-S$ un qualsiasi vertex-cover, consideriamo due nodi $u \in S$ e $v \in S$. Notiamo che $(u, v) \notin E$ poiché $V-S$ è un vertex-cover.

Quindi, non esistono due nodi in S connessi da un edge $\rightarrow S$ independent set.



VERTEX-COVER \leq_p INDEPENDENT-SET

Se abbiamo un oracolo per risolvere l'Independent Set, allora possiamo decidere se G ha una Vertex Cover di dimensioni al massimo k , chiedendo all'oracolo se G ha un Independent Set di dimensioni almeno $|v| - k$.

INDEPENDENT-SET \leq_p VERTEX-COVER

Se abbiamo un oracolo per risolvere la Vertex Cover, allora possiamo decidere se G ha un Independent Set di dimensioni almeno k , chiedendo all'oracolo se G ha una Vertex Cover di dimensioni al massimo $|v| - k$.

Per riassumere, questo tipo di analisi illustra il nostro piano in generale: anche se non sappiamo come risolvere in modo efficiente né l'Independent Set né la Vertex Cover, le due dimostrazioni ci dicono come potremmo risolvere uno di questi trovando una soluzione efficiente all'altro problema.

RIDUZIONE DA CASO SPECIALE A CASO GENERALE:

L'Independent Set può essere visto come un problema di imballaggio: l'obiettivo è quello di "impacchettare" il maggior numero possibile di vertici, senza a "conflitti" (senza archi che li collegano).

Il Vertex Cover, d'altra parte, può essere visto come un problema di copertura: l'obiettivo è "coprire" tutti gli archi del grafo usando il minor numero possibile di vertici.

Il Vertex Cover è un problema di copertura espresso specificamente nel linguaggio dei grafi, ma c'è un problema di copertura più generale, ovvero il **SET-COVER**, in cui si cerca di coprire un insieme arbitrario di oggetti usando una raccolta di insiemi più piccoli, ed è definito come:

Dato un insieme di elementi $\{1, 2, \dots, n\}$ (chiamato universo U) e una raccolta S di m sottoinsiemi (S_1, S_2, \dots, S_m) la cui unione è uguale ad U , il **Set Cover Problem** è identificare le più piccole sotto-raccolte di S la cui unione è uguale all'universo U .

Domanda: Dato un insieme U di elementi, una collezione S_1, S_2, \dots, S_m di sottoinsiemi di U , e un intero k , esiste una collezione $\leq k$ di questi sottoinsiemi la cui unione è uguale ad U ?

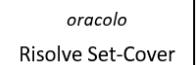
Nell'esempio a destra, se $k=2$ il nostro obiettivo è trovare al più 2 sottoinsiemi la quale la loro unione forma U , in questo caso $S_2 \cup S_6 = U$.

$U = \{1, 2, 3, 4, 5, 6, 7\}$	
$k = 2$	
$S_1 = \{3, 7\}$	$S_4 = \{2, 4\}$
$S_2 = \{3, 4, 5, 6\}$	$S_5 = \{5\}$
$S_3 = \{1\}$	$S_6 = \{1, 2, 6, 7\}$

Intuitivamente, sembra che Vertex Cover sia un caso speciale di Set Cover: in quest'ultimo caso, stiamo cercando di coprire un insieme arbitrario di vertici usando sottoinsiemi arbitrari di vertici stessi, mentre nel primo caso, stiamo specificamente cercando di coprire gli archi di un grafo usando l'insieme degli archi incidente ai vertici, possiamo provare che **VERTEX-COVER \leq_p SET-COVER**.

Dimostrazione:

Supponiamo di avere accesso a un oracolo in grado di risolvere Set Cover e prendere in considerazione un'istanza arbitraria di Vertex Cover, specificata da un grafo $G = (V, E)$ e un numero k .



Il nostro obiettivo è quello di coprire gli archi in E , quindi formuliamo un'istanza di Set Cover in cui l'insieme U è uguale ad E .

Ogni volta che selezioniamo un vertice nel Vertex Cover Problem, copriamo tutti gli archi incidenti ad esso; quindi, per ogni vertice $v \in V$, aggiungiamo un insieme $S_v \subseteq U$ alla nostra istanza Set Cover, costituito da tutti gli archi in G incidenti a v .

Ora ricordiamo che U può essere coperto con al massimo k degli insiemi S_1, \dots, S_n se e solo se G ha una Vertex Cover di dimensioni al massimo k .

(\Rightarrow) Questo può essere dimostrato molto facilmente, ovvero se S_{v_1}, \dots, S_{v_l} sono $l \leq k$ insiemi che coprono U , allora ogni arco in G è incidente a uno dei vertici v_1, \dots, v_l , e quindi l'insieme $\{v_1, \dots, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$.

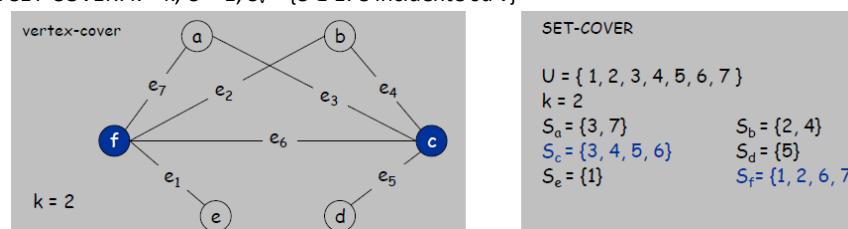
(\Leftarrow) Al contrario, se $\{v_1, \dots, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$, allora gli insiemi S_{v_1}, \dots, S_{v_l} coprono U .

Pertanto, data la nostra istanza di Vertex Cover, formuliamo l'istanza di Set Cover descritta sopra e la passiamo al nostro oracolo.

Rispondiamo sì se e solo se l'oracolo risponde sì.

Ricapitolando:

Creiamo un'istanza di SET-COVER: $k = k$, $U = E$, $S_v = \{e \in E: e \text{ incidente su } v\}$



Set-cover è di dimensione $\leq k$ sse vertex-cover è di dimensione $\leq k$ (dimostrata sopra).

RIDUZIONE VIA “GADGETS”:

Prima di parlare di questa riduzione introduciamo il **problema della soddisfacibilità**, uno dei problemi noti NP-Completo (*classe descritta più avanti*).

Si ricordi che le variabili booleane assumono solo valori vero o falso (1 o 0), con queste si possono creare operazioni booleane tramite AND, OR e NOT.

Una **formula booleana** è un'espressione che coinvolge variabili e operazioni booleane, per esempio $\varphi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$, ed è **soddisfacibile** se qualche assegnamento di 0 e 1 alle variabili fa sì che la formula valga 1, nella formula precedente lo si fa con $x=0, y=1$ e $z=0$, quindi diciamo che l'assegnamento **soddisfa** φ . Da tutto ciò, diciamo che il **problema della soddisfacibilità** consiste nel verificare se una formula booleana è soddisfacibile, denotata **SAT**.

Utilizzeremo un caso speciale di SAT, ovvero 3SAT, in cui tutte le formule booleane sono in una forma speciale. Prima di parlare di questa forma speciale introduciamo altri due concetti che serviranno, ovvero **letterale** che è una variabile booleana o il suo negato, e **clausola C_j** che consiste in diversi letterali connessi tramite operatore **OR (V)**, ad esempio $C_1 = (\bar{x} \vee y \vee z)$.

Una formula booleana è in **forma normale congiunta**, ed è detta una **forma cnf**, se comprende diverse clausole connesse tramite operatore **AND (\wedge)** ($\varphi = C_1 \wedge C_2 \wedge C_3$), come ad esempio la seguente formula: $\varphi = (\bar{x} \vee y \vee z) \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee y \vee \bar{z})$.

Essa è una **formula 3cnf** se tutte le clausole della formula hanno esattamente tre letterali, come: $(\bar{x} \vee y \vee z) \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee y \vee \bar{z})$.

Questo problema di verificare se una formula booleana è in 3cnf, viene definito **3-SAT = { $\{\varphi\} | \varphi$ è una formula 3cnf soddisfacibile}**. Domanda:

Dato un insieme di clausole C_1, \dots, C_k , ciascuno di lunghezza 3, su un insieme di variabili $X = \{x_1, \dots, x_n\}$, esiste un assegnamento di verità soddisfacente?

Mettiamo in relazione la difficoltà computazionale tra **3-SAT**, che riguarda l'impostazione delle variabili booleane in presenza di vincoli, e **Independent-Set**, che riguarda la selezione dei vertici indipendenti in un grafo.

Per risolvere un'istanza di 3-SAT, usando un oracolo che risolve Independent-Set, abbiamo bisogno di una codifica di tutti i vincoli booleani in nodi e archi di un grafo, in modo che la soddisfabilità corrisponda all'esistenza di trovare un insieme indipendente, dimostriamo **3-SAT \leq_p Independent-Set**.

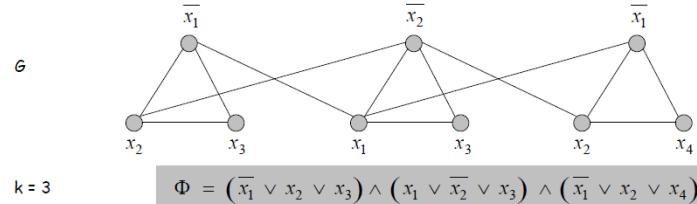
Dimostrazione:

Abbiamo un oracolo per Independent-Set e vogliamo risolvere un'istanza di 3-SAT composta da variabili $X = \{x_1, \dots, x_n\}$ e clausole C_1, \dots, C_k .

Da quanto detto, dobbiamo trovare un modo di prendere la formula booleana da risolvere e convertirla in un grafo, affinché l'oracolo che già abbiamo risolvi il nostro problema. Diciamo che due termini sono in "conflitto" se una variabile x_i ha in comune un arco col suo negato \bar{x}_i .

Un modo di risolvere un'istanza di 3-SAT è il seguente: scegliere un termine da ciascuna clausola, non quelli in "conflitto", e uguagliarli ad 1, in modo da soddisfare tutte le clausole.

Iniziamo a costruire il grafo G che contiene 3 vertici per ogni clausola, uno per ogni letterale, connettiamo i 3 letterali in una clausola (che forma un triangolo), ed infine connettiamo ogni letterale ad ogni suo negato, risultando come segue:



Prima di procedere, considera come appaiono gli insiemi indipendenti di dimensione k in questo grafo: Non è possibile selezionare due vertici dallo stesso triangolo, ma questo è proprio il nostro obiettivo, ovvero quello di scegliere un termine in ciascuna clausola, che poi verrà assegnato a 1, soddisfacendo la nostra formula booleana.

Da tutto questo si enuncia che: **G contiene un insieme indipendente di dimensione $k = |\varphi|$ sse φ è soddisfacibile**, in altre parole supponiamo che l'istanza 3-SAT originale sia soddisfacibile se e solo se il grafo G , che abbiamo costruito, ha un insieme indipendente di dimensioni **almeno k** .

Dimostriamolo nei due versi:

(\Leftarrow) In primo luogo, supponiamo che il nostro grafo G abbia un insieme indipendente S di dimensioni almeno k . Quindi la dimensione di S è esattamente k , e deve consistere in un nodo per ogni triangolo. Ora, sosteniamo che esiste un'assegnazione di verità v per le variabili nell'istanza 3-SAT con la proprietà che le etichette di tutti i nodi in S valgono 1. Ecco come possiamo costruire tale assegnazione v . Se un nodo in S fosse etichettato \bar{x}_i e un altro fosse etichettato x_i , allora ci sarebbe un arco tra questi due nodi, contraddicendo la nostra ipotesi, ovvero che S sia un insieme indipendente. Pertanto, se x_i appare come un'etichetta di un nodo in S , impostiamo $v(x_i)=1$, altrimenti impostiamo $v(x_i)=0$. Costruendo v in questo modo, tutte le etichette dei nodi in S valgono 1.

Riassumendo:

Sia S l'insieme indipendente di dimensione k :

- S deve contenere esattamente un vertice di ogni triangolo
- Non può contenere un letterale ed il suo negato

Poniamo questi letterali di S a 1, l'assegnazione di verità è consistente e tutte le clausole sono soddisfatte.

(\Rightarrow) Al contrario, se l'istanza 3-SAT è soddisfacibile, allora ogni triangolo nel grafo contiene almeno un nodo che ha valore 1. Sia S un insieme costituito da uno di questi nodi da ciascun triangolo. Sosteniamo che S è indipendente, poiché se ci fosse un arco tra due nodi $u, v \in S$, allora le etichette di u e v dovrebbero essere in conflitto (uno 0 e l'altro 1), ma questo non è possibile, poiché entrambi i vertici valgono 1.

Riassumendo:

Data un'assegnazione soddisfacente, selezioniamo un letterale che vale 1 da ogni triangolo.

Questo è un insieme indipendente di dimensione k .

Per concludere, poiché G ha un Independent-Set di dimensioni almeno k se e solo se l'istanza 3-SAT originale è soddisfacente, la riduzione è completa.

TRANSITIVITÀ DELLE RIDUZIONI:

Se $X \leq_p Y$ e $Y \leq_p Z$, allora $X \leq_p Z$

Dimostrazione:

Dato un oracolo per Z , mostriamo come risolvere un'istanza di X , essenzialmente componiamo solo i due algoritmi implicati da $X \leq_p Y$ e $Y \leq_p Z$.

Eseguiamo l'algoritmo per X usando un oracolo per Y ; ma ogni volta che viene chiamato l'oracolo per Y , lo simuliamo in un numero polinomiale di passaggi usando l'algoritmo che risolve istanze di Y , ma in tal caso stiamo usando un oracolo per Z .

Ad esempio, poiché abbiamo dimostrato che **3-SAT \leq_p Independent Set \leq_p Vertex Cover \leq_p Set Cover**, possiamo concludere che **3-SAT \leq_p Set Cover**.

8.2 CERTIFICAZIONE EFFICIENTE E DEFINIZIONE DI NP

Quando abbiamo parlato di Independent-Set Problem abbiamo detto che un grafo contiene un insieme indipendente di dimensioni almeno k , per provarlo si può benissimo esibire un esempio di grafo in modo tale che contiene k insiemi indipendenti. Allo stesso modo, se un'istanza 3-SAT è soddisfacibile, basta prendere una formula booleana in forma 3cnf e assegnare i vari valori alle variabili in modo che tale formula risulti 1.

Il problema è il contrasto tra la **ricerca** di una soluzione e il **controllo** di una soluzione proposta.

Per Independent-Set o 3-SAT, non conosciamo un algoritmo a tempo polinomiale per trovare tali soluzioni, ma il controllo di una soluzione proposta a questi problemi può essere facilmente eseguito in tempi polinomiali.

Da ciò possiamo dire che esistono due diversi problemi, un **problema di decisione**, ovvero controllare se esiste una soluzione al problema, e un **problema di ricerca**, ovvero trovare l'effettiva soluzione al problema, noi ci occuperemo di problemi relativi al primo tipo.

Questo ci porta al **Self-Reducibility**, il quale afferma che un **problema di ricerca \leq_p problema di decisione**, ciò si applica a tutti i problemi NP-Completi che vedremo più avanti, giustificando la focalizzazione sui problemi di decisione.

PROBLEMI E ALGORITMI:

L'input per un problema computazionale verrà codificato come una stringa binaria finita s , ed indichiamo la lunghezza con $|s|$. Identificheremo un problema decisionale X con l'insieme di stringhe su cui la risposta è "sì". Un algoritmo A per un problema di decisione riceve una stringa di input se restituisce il valore "sì" o "no": indicheremo questo valore restituito con $A(s)$.

Diciamo che l'algoritmo A risolve il problema X se per tutte le stringhe s , abbiamo $A(s) = \text{sì}$, se e solo se $s \in X$.

Ricapitolando: Un **problema di decisione** è caratterizzato da:

- X insieme di stringhe provenienti dal nostro problema decisionale
- Un'istanza di X è una stringa s
- Algoritmo A risolve il problema X : $A(s) = \text{sì}$ sse $s \in X$.

Come sempre, diciamo che A ha un tempo di esecuzione polinomiale se esiste una **funzione polinomiale** $p(\cdot)$ in modo che per ogni stringa di input s , l'algoritmo A termina su s al più in $p(|s|)$ passi.

Ricapitolando: **Tempo polinomiale**:

Algoritmo A ha tempo polinomiale se per ogni stringa s , $A(s)$ termina in al più $p(|s|)$ "passi", dove $p()$ è qualche polinomio.

CERTIFICAZIONE EFFICIENTE:

Un "**algoritmo di controllo**", chiamato "**certificatore**", per un problema X ha una struttura diversa da un algoritmo che cerca effettivamente di risolvere il problema. Per molti problemi non è noto se esiste un algoritmo polinomiale di soluzione.

Un certificatore non serve a risolvere il problema, ma permette di verificare se l'istanza di un problema è una istanza "sì" con l'aiuto di un altro elemento, ovvero il **certificato** denotato con t .

Per "**controllare**" una soluzione, abbiamo bisogno della stringa di input s , nonché di una stringa di "**certificato**" separata t che contenga la prova che s è un'istanza "sì" di X .

Tecnicamente ciò significa che esiste un algoritmo di verifica $C(s, t)$ che, usando t , controlla in tempo polinomiale l'esistenza di una soluzione per s avente la proprietà richiesta.

Quindi diciamo che C è un **certificatore efficiente** per un problema X se valgono le seguenti proprietà:

- C è un algoritmo a tempo polinomiale che accetta due argomenti di input s e t .
- Esiste una funzione polinomiale p in modo che per ogni stringa s , abbiamo $s \in X$ sse esiste una stringa t tale che $|t| \leq p(|s|)$ e $C(s, t) = \text{sì}$.

Algoritmo $C(s, t)$ è un certificatore per il problema X se per ogni stringa s , $s \in X$ sse esiste una stringa t (certificato) tale che $C(s, t) = \text{sì}$.

Nota: Il **certificatore** non determina se $s \in X$, semplicemente controlla se una data dimostrazione t che $s \in X$, quindi **vede se il certificato è corretto**.

Esempio: Un certificatore efficiente C può essere utilizzato come componente principale di un algoritmo "bruteforce" per un problema X : su un input s , prova tutte le stringhe t di lunghezza $\leq p(|s|)$ e vede se $C(s, t) = \text{sì}$ per una di queste stringhe.

Ma l'esistenza di C non ci fornisce alcun modo chiaro per progettare un algoritmo efficiente che risolva effettivamente X ; dopo tutto, spetta ancora a noi trovare una stringa t che farà sì che $C(s, t)$ dica "sì" e ci sono molte possibilità esponenziali per t .

CLASSE DEI PROBLEMI - NP:

Definiamo NP come l'insieme di tutti i problemi per i quali esiste un **certificatore efficiente**. L'atto di cercare una stringa t che farà sì che un certificatore efficiente accetti gli input s , viene spesso visto come una ricerca non deterministica nello spazio delle possibili dimostrazioni t ; per questo motivo, NP è stato nominato acronimo di "**Non-deterministic Polynomial-time**".

Adesso possiamo ridefinire alcune classi di problemi già viste:

- **P**: Problema di decisione per cui esiste un **algoritmo polinomiale**.
- **EXP**: Problema di decisione per cui esiste un **algoritmo esponenziale**.
- **NP**: Problema di decisione per cui esiste **certificatore polinomiale**.

Da quanto appena detto possiamo osservare che la classe **P ⊆ NP**.

Dimostrazione:

Consideriamo un problema $X \in P$, questo significa che esiste un algoritmo in tempo polinomiale A che risolve X . Per dimostrare che $X \in NP$, dobbiamo dimostrare che esiste un certificatore efficiente C per X , progettiamo C come segue:

Quando C viene eseguito con una coppia di input (s, t) , il certificatore C restituisce semplicemente il valore di $A(s)$. C è un certificatore efficiente per X perché ha un tempo di esecuzione polinomiale, poiché esegue A (che ha un tempo polinomiale). Quindi, se una stringa $s \in X$, allora per ogni t di lunghezza al massimo $p(|s|)$, abbiamo $C(s, t) = \text{sì}$. D'altra parte, se $s \notin X$, allora per ogni t di lunghezza al massimo $p(|s|)$, abbiamo $C(s, t) = \text{no}$.

Riassumendo:

Consideriamo un qualsiasi problema X in P

Per Definizione, esiste un algoritmo polinomiale $A(s)$ che risolve X .

Certificato: $t = \epsilon$, mentre il certificatore: $C(s, t) = A(s)$.

Adesso possiamo verificare che alcuni problemi introdotti precedentemente appartengono ad NP, si tratta di determinare in che modo un certificatore efficiente, per ciascuno di essi, utilizzerà una stringa t "certificato":

Ricordiamo che un numero è composto se è il prodotto di due numeri maggiori di 1, ovvero è composto se non è primo.

Il primo problema polinomialmente verificabile è il **COMPOSITES**, ovvero dato un intero s , questo verifica se è composto. Abbiamo:

Il **certificato**:

Un fattore non banale t di s . Nota che tale certificato esiste sse s è composto. Inoltre $|t| \leq |s|$.

Ed il **certificatore**:

```
boolean C(s, t) {
    se (t ≤ 1 or t ≥ s)
        Restituisci false
    else se (s è un multiplo di t)
        Restituisci true
    else
        Restituisci false
}
```

Una prova di tale problema può essere:

Istanza $s=437,669 \rightarrow$ Certificato $t=541$ o 809 ($437,669 = 541 \times 809$)

In conclusione, **COMPOSITES** è in NP.

Il secondo problema polinomialmente verificabile è il **SAT/3-SAT**, ovvero data una formula CNF φ , verifica se esiste un'assegnazione che la soddisfa.

Il **certificato** sarà un assegnamento di valori di verità alle n variabili booleane.

Il **certificatore** invece controlla che ogni clausola in φ ha almeno un letterale ad 1.

Una prova di tale problema può essere:

$$\begin{array}{c} (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4) \\ \text{istanza } s \\ x_1=1, x_2=1, x_3=0, x_4=1 \\ \text{certificato } t \end{array}$$

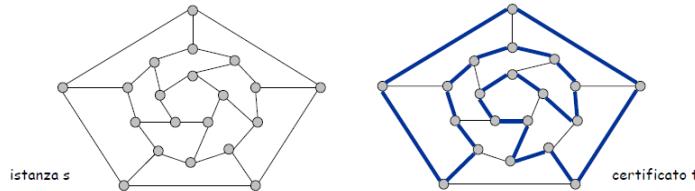
In conclusione, **SAT/3-SAT** è in NP.

Il terzo problema polinomialmente verificabile è **HAM-CYCLE**, ovvero dato un grafo non orientato $G=(V,E)$, verifica se esiste un ciclo semplice C che visita ogni nodo esattamente una volta.

Il **certificato** è una permutazione di n nodi.

Il **certificatore** controlla che la permutazione contiene ogni nodo in V esattamente una volta, e che esiste un arco tra ogni coppia di nodi adiacenti nella permutazione.

Una prova di tale problema può essere:



In conclusione, **HAM-CYCLE** è in NP.

Tra l'altro, non sappiamo se NP è contenuta in una classe di complessità deterministica più piccola, il miglior metodo deterministico attuale noto per decidere se un linguaggio è in NP, è vedere se utilizza un tempo esponenziale. In altre parole, possiamo dimostrare che **NP ⊆ EXP**.

Dimostrazione:

Consideriamo un qualsiasi problema X in NP.

Per definizione, esiste un certificatore polinomiale $C(s, t)$ per X .

Per risolvere X su input s , usiamo $C(s, t)$ su tutte le stringhe t con $|t| \leq p(|s|)$.

Restituisci s , se $C(s, t)$ restituisce s per una qualsiasi stringa.

LA QUESTIONE P=NP?

La domanda se $P=NP$ è uno dei maggiori problemi irrisolti dell'informatica teorica, se queste classi fossero uguali, qualsiasi problema polinomialmente verificabile sarebbe anche polinomialmente decidibile. La maggior parte dei ricercatori ritengono che non siano uguali, perché molte persone hanno investito, senza successo, sforzi per trovare algoritmi aventi tempo polinomiale per problemi in NP.

Tuttavia, non possiamo dimostrare che nessuno di questi problemi richieda più del tempo polinomiale per risolverlo. In effetti, non possiamo dimostrare che in NP vi siano problemi che non appartengono a P. Quindi al posto di un teorema concreto, possiamo solo fare una domanda:

C'è un problema in NP che non appartiene a P? $P = NP?$



8.3 NP-COMPLETITÀ

In assenza di progressi sulla domanda $P = NP$, le persone si sono rivolte a una domanda correlata ma più accessibile: quali sono i problemi più difficili in NP? Le riduzioni polinomiali ci offrono un modo per affrontare questa domanda e ottenere informazioni sulla struttura di NP.

Approfondimento (Tesi di Cook-Levin):

Un problema L è NP completo se sta in NP e se $\forall L'$ (appartenente anch'esso a NP), $L' \leq L$, ovvero se L è il più difficile problema in NP.

In altre parole, possiamo dire che un linguaggio L è NP-completo se i seguenti enunciati sono veri:

1. L è in NP
2. Per ogni Linguaggio L' in NP esiste una riduzione polinomiale di L' a L

si può dire sia un caso particolare della *Cook-completezza* che definisce un problema NP-completo se, dato un *oracolo* per il problema P , ovvero un meccanismo in grado di rispondere ad una qualsiasi domanda sull'appartenenza di una stringa a P in un'unità di tempo, è possibile riconoscere in tempo polinomiale un qualsiasi linguaggio in NP. La definizione di *Cook-completezza* risulta essere più generale tanto da includere i complementi dei problemi NP-completi nella classe dei problemi NP-completi.

"Riducibile" quindi significa che per ogni problema L , c'è una *riduzione polinomiale*, ovvero un algoritmo deterministico che trasforma istanze $I \in L$ in istanze $C \in C$, così che la risposta a C è sì se e solo se la risposta a I è sì. Per provare che un problema NP A è infatti un problema NP-completo è sufficiente mostrare che un problema NP-completo già conosciuto si riduce a A .

Una conseguenza di questa definizione è che se avessimo un algoritmo di tempo polinomiale per C , potremmo risolvere tutti i problemi in NP in tempo polinomiale, questa definizione è stata data da Stephen Cook.

Un problema che soddisfa la condizione 2 ma non necessariamente la condizione 1 è detto NP-hard. Informalmente, un problema NP-hard è "almeno difficile come" qualsiasi problema NP-completo, e forse anche più difficile.

Il problema X si riduce in modo polinomiale (Cook) al problema Y se istanze arbitrarie del problema X possono essere risolte usando:

Un numero polinomiale di passi di computazione standard, più un numero polinomiale di chiamate ad oracolo che risolve il problema Y.

Il problema X si trasforma in modo polinomiale (Karp) al problema Y se dato un qualsiasi input $x \in X$, possiamo costruire un input y tale che x è istanza "si" di X se y è istanza "si" di Y.

Nota: Le trasformazioni polinomiali sono riduzioni polinomiali con una sola chiamata all'oracolo per Y, esattamente alla fine dell'algoritmo per X.

Quasi tutte le precedenti riduzioni erano di questa forma.

DEFINIZIONE NP-COMPLETO: Probabilmente il modo più naturale per definire un problema "più difficile" X, è attraverso le seguenti due proprietà:

- (i) $X \in NP$;
- (ii) per tutti $Y \in NP$, $Y \leq_P X$.

In altre parole, richiediamo che ogni problema in NP possa essere ridotto a X. Chiameremo tale X un problema NP-completo.

Supponiamo che X sia un problema NP-completo, allora X è risolvibile in tempo polinomiale se e solo se $P=NP$.

Dimostrazione:

(\Leftarrow) Chiaramente, se $P = NP$, allora X può essere risolto in tempo polinomiale poiché appartiene a NP.

(\Rightarrow) Al contrario, supponiamo che X possa essere risolto in tempo polinomiale.

Se Y è un altro problema in NP, allora $Y \leq_P X$, e quindi Y può essere risolto in tempo polinomiale.

Ciò implica $NP \subseteq P$, ma già sappiamo che $P \subseteq NP$, e quindi $P=NP$.

Una conseguenza di questa dimostrazione è: Se c'è un problema in NP che non può essere risolto in tempo polinomiale, allora nessun problema NP completo può essere risolto in tempo polinomiale.

CIRCUIT SATISFIABILITY, il primo problema NP-Completo:

Il primo problema NP-Completo è il **CIRCUIT-SAT**, chiamato anche "soddisfabilità dei circuiti".

Per dimostrare che un problema è NP-completo, si deve mostrare come quest'ultimo potrebbe codificare qualsiasi problema in NP. Questa è una questione molto più complicata di quanto riscontrato nelle riduzioni polinomiali precedenti, in cui abbiamo cercato di codificare specifici problemi individuali in termini di altri problemi già noti.

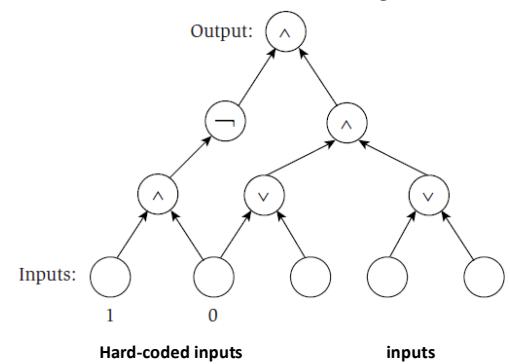
Nel 1971, Cook e Levin mostrarono indipendentemente come farlo per problemi molto naturali in NP. Forse la scelta del problema più naturale per un primo problema NP-completo è il seguente **Circuit Satisfiability Problem**.

Per specificare questo problema, dobbiamo precisare cosa intendiamo per circuito. Consideriamo gli operatori booleani standard che abbiamo usato per definire il problema di soddisfazione: \wedge (AND), \vee (OR) e \neg (NOT). La nostra definizione di circuito è progettata per rappresentare un circuito fisico costruito da nodi che implementano questi operatori. Quindi definiamo un circuito K come un grafo aciclico marcato e diretto come il seguente:

- Le foglie in K sono etichettate con una delle costanti 0 o 1 o con il nome di una variabile distinta. I nodi di quest'ultimo tipo saranno indicati come gli input al circuito.
- Ogni altro nodo è etichettato con uno degli operatori booleani \wedge , \vee o \neg ; i nodi etichettati con \wedge o \vee avranno due archi in entrata e i nodi etichettati con \neg avranno un arco in entrata.
- Esiste un singolo nodo senza archi in uscita che rappresenterà l'output: il risultato che viene calcolato dal circuito, ovvero la radice dell'albero.

Alle due foglie a sinistra sono assegnati i valori 1 e 0 e le tre foglie successive costituiscono gli input.

Ad esempio, se in input abbiamo 101, otteniamo nel penultimo livello 011 per i rispettivi nodi di quel, per i nodi del secondo livello otteniamo 11 ed infine otterremo 1 sulla radice, la quale rappresenta il nostro output (che ha soddisfatto la formula iniziale essendo =1).



Ora, il problema di soddisfazione del circuito è il seguente: Ci viene dato un circuito e dobbiamo decidere se esiste un'assegnazione di valori in input che fa sì che l'output restituiscia il valore 1 (in tal caso, diremo che il circuito dato è soddisfatto).

Nel nostro esempio, abbiamo appena visto, tramite l'assegnazione 101 in input, che il circuito è soddisfatto ed è quindi istanza "si".

Possiamo vedere il teorema di Cook e Levin nel modo seguente: **CIRCUIT-SAT è NP-Completo**.

Come già discusso, la dimostrazione di questo teorema richiede che consideriamo un problema arbitrario X in NP e mostrare che $X \leq_P CIRCUIT-SAT$.

Non descriveremo la dimostrazione in dettaglio, ma in realtà non è poi così difficile basta seguire l'idea di base:

Un qualsiasi algoritmo che prende in input un numero fissato n di bits e produce una risposta si/no può essere rappresentato con tale circuito inoltre, se l'algoritmo prende tempo polinomiale, allora il circuito è di dimensione polinomiale.

Questo circuito è equivalente ad un algoritmo, nel senso che il suo output è esattamente 1 sugli input per i quali l'algoritmo produce sì.

(Si noti che un algoritmo non ha problemi a gestire input di lunghezze variabili, ma un circuito è strutturalmente codificato con una dimensione fissa).

Dimostrazione (idea):

Consideriamo un problema X in NP, questo ha un certificatore polinomiale $C(s, t)$ per definizione.

Per determinare se $s \in X$, dobbiamo sapere se esiste un certificato t di lunghezza $p(|s|)$ tale che $C(s, t) = \text{si}$. Consideriamo $C(s, t)$ come un algoritmo su $|s| + p(|s|)$ bits (input s , certificato t) e convertiamolo in circuito di dimensione polinomiale K con $|s| + p(|s|)$ foglie.

- i primi $|s|$ bits (foglie) sono **hard-coded** con s (bits fissi)
- restanti $p(|s|)$ bits (foglie) rappresentano i bits di t

Quindi, il circuito K è soddisfacibile se e solo se $C(s, t) = \text{si}$, ovvero se il circuito restituisce 1.

Ora osserviamo semplicemente che $s \in X$ se e solo se esiste un modo per impostare i bit di ingresso su K in modo che il circuito produca 1.

Ciò stabilisce che $X \leq_P CIRCUIT-SAT$.

Esempio:

Supponiamo di avere il seguente problema: Dato un grafico G, contiene un indipendent-set (problema in NP) a due nodi?

Vediamo come un'istanza di questo problema può essere risolta costruendo un'istanza equivalente di CIRCUIT-SAT.

Costruiamo un circuito equivalente K:

Supponiamo che siamo interessati a decidere la risposta a questo problema per un grafo G sui tre nodi u, v, w , in cui v è unito sia a u che a w (grafo rappresentato dalla figura a destra a sinistra). Ciò significa che ci occupiamo di un input di lunghezza $n = 3$. La Figura a destra mostra un circuito equivalente a un certificatore efficiente per il nostro problema (il lato destro del circuito verifica che siano stati selezionati almeno due nodi e il lato sinistro verifica che non abbiamo selezionato nodi collegati da un arco).

Codifichiamo gli archi di G come costanti nelle prime tre foglie e lasciamo le restanti tre foglie (che rappresentano la scelta dei nodi da inserire nell'insieme indipendente) come variabili.

Ora osserviamo che questa istanza di CIRCUIT-SAT è soddisfacibile con l'assegnazione $u=1, v=0$ e $w=1$ come input.

Ciò corrisponde alla scelta dei nodi uw , che in effetti formano un insieme indipendente a due nodi nel nostro grafo G .

STABILIRE NP-COMPLETEZZA:

Una volta che conosciamo il primo problema NP-completo, possiamo scoprirne altri attraverso la seguente osservazione:

Se X è un problema NP-completo e Y è un problema in NP con la proprietà che $X \leq_p Y$, allora Y è NP-completo.

Dimostrazione:

Sia W un qualsiasi problema in NP, abbiamo $W \leq_p X$, per definizione di NP-completo, e $X \leq_p Y$ per ipotesi, tramite transitività ne consegue che $W \leq_p Y$. Quindi Y è NP-Completo.

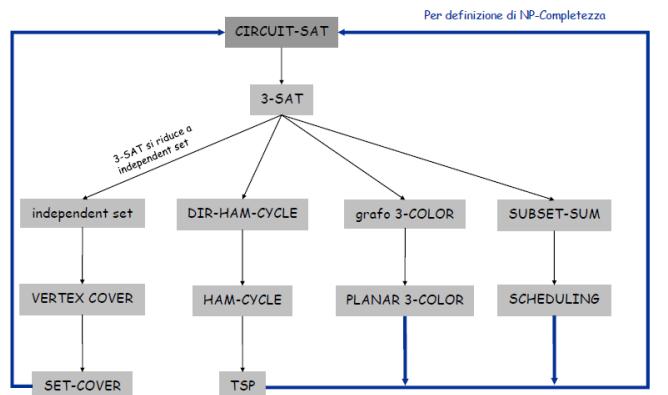
Riassumendo: **Passi per stabilire NP-Completezza** di problema Y :

- passo 1. Mostrare che Y è in NP.
- passo 2. Scegliere un problema NP-Completo X .
- passo 3. Provare che $X \leq_p Y$.

In precedenza, abbiamo riscontrato una serie di riduzioni tra alcuni problemi di base, come Indipendet-Set, Ham-Cycle, etc....

Per stabilire la loro NP-completezza, basterebbe collegare CIRCUIT-SAT a quest'ultimi, ma risulterebbe molto lungo e difficile.

Il modo più semplice per farlo è mettere in relazione CIRCUIT-SAT con 3-SAT, più precisamente riduciamo tutti questi problemi di base a 3-SAT, ed infine riduciamo quest'ultimo a CIRCUIT-SAT, così per la transitività e come se riducessimo tutti questi problemi di base, a CIRCUIT-SAT.



Osservazione: Tutti i problemi sono NP-Completi e ammettono riduzione polinomiale da uno all'altro!

Pratica: La maggior parte dei problemi NP sono noti essere in P oppure essere NP-Completi.

Adesso mostriamo che **3-SAT è NP-Completo**.

Dimostrazione:

Chiaramente 3-SAT è in NP, poiché possiamo verificare in tempo polinomiale che un'assegnazione di verità soddisfa l'insieme di clausole dato.

Dimostreremo che è NP-completo tramite la riduzione **CIRCUIT-SAT** \leq_p **3-SAT**.

Data un'istanza arbitraria di CIRCUIT-SAT, costruiremo un'istanza equivalente di 3-SAT in cui ogni clausola contiene al massimo tre variabili.

Quindi consideriamo un circuito arbitrario K ed associamo delle variabili 3-SAT x_v a ciascun nodo v del circuito. Ora dobbiamo codificare i nodi etichettati con operatori booleani, in modo da soddisfare le clausole. Ci saranno tre casi a seconda della tipologia di operazione:

- Se il nodo x_2 è etichettato con \neg , e il suo solo arco di input proviene dal nodo x_3 , allora dobbiamo avere $x_2 = \bar{x}_3$.
Trasformiamo ciò aggiungendo 2 clausole ($x_2 \vee x_3$) e ($\bar{x}_2 \vee \bar{x}_3$).
- Se il nodo x_1 è etichettato con V e i suoi due archi di input provengono dai nodi x_4 e x_5 , dobbiamo avere $x_1 = x_4 \vee x_5$.
Trasformiamo ciò aggiungendo le 3 clausole: ($x_1 \vee \bar{x}_4$), ($x_1 \vee \bar{x}_5$) e ($\bar{x}_1 \vee x_4 \vee x_5$).
- Se il nodo x_0 è etichettato con \wedge e i suoi due archi di input provengono dai nodi x_1 e x_2 , dobbiamo avere $x_0 = x_1 \wedge x_2$.
Trasformiamo ciò aggiungendo le 3 clausole: ($\bar{x}_0 \vee x_1$), ($\bar{x}_0 \vee x_2$) e ($x_0 \vee \bar{x}_1 \vee \bar{x}_2$).

Hard-coded input e output:

Dobbiamo garantire che le costanti (foglie a sinistra dell'albero) abbiano i loro valori e che l'output sia uguale a 1.

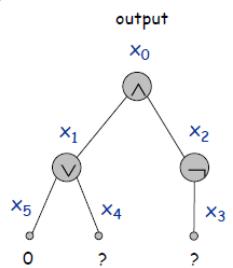
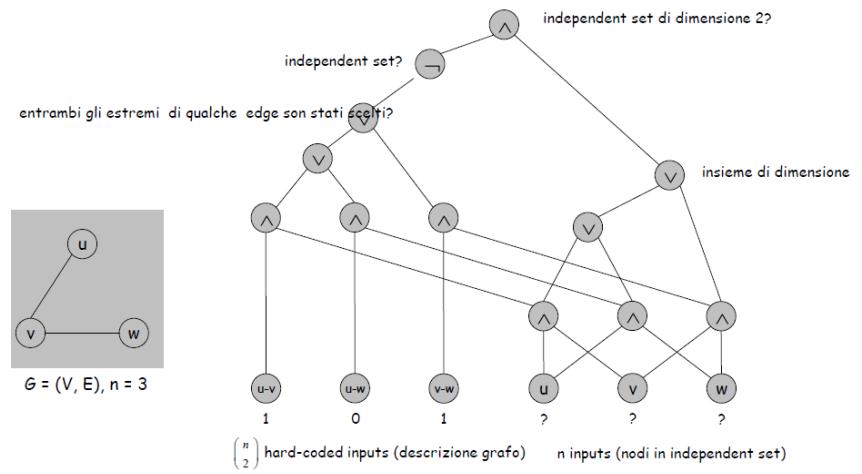
Pertanto, per il nodo x_5 che è stato etichettato come una costante, aggiungiamo una clausola con la variabile x_5 o \bar{x}_5 , in modo da forzare x_5 ad assumere il valore designato, in questo caso aggiungiamo la clausola con la singola variabile \bar{x}_5 .

Per il nodo x_0 che rappresenta l'output, aggiungiamo la clausola con la singola variabile x_0 , la quale richiede di valere 1, concludendo la costruzione.

Passo finale:

Finora abbiamo creato un'istanza SAT, ma il nostro obiettivo era quello di creare un'istanza di 3-SAT, quindi dobbiamo convertire questa istanza di SAT in una istanza equivalente in cui ogni clausola ha esattamente tre variabili.

Per far ciò, bisogna inserire altre variabili nelle clausole con lunghezza <3, e assegnare tali variabili con valori in modo che il risultato non cambi.



8.4 CO-NP

Abbiamo visto che l'idea di un certificatore efficiente non suggerisce un algoritmo concreto per risolvere effettivamente il problema.

La definizione di certificazione efficiente, e quindi in NP, è fondamentalmente asimmetrica:

Una stringa di input s è un'istanza "si" se e solo se esiste una t in modo che $C(s, t) = \text{"si"}$, negando questa affermazione, vediamo che una stringa di input s è un'istanza "no" se e solo se per tutte le t , abbiamo che $C(s, t) = \text{"no"}$.

Per ogni problema X , esiste un problema naturale **complementare** \bar{X} : per tutte le stringhe di input s , diciamo $s \in \bar{X}$ se e solo se $s \notin X$. Nota che se $X \in$ classe P, allora $\bar{X} \in P$, poiché da un algoritmo A che risolve X , possiamo produrre un algoritmo \bar{A} che esegue A e poi inverte la sua risposta.

In altre parole, dato un problema di decisione X , il **complemento** \bar{X} è lo stesso problema con risposte si e no invertite.

Ma se $X \in NP$, dovrebbe essere $\bar{X} \in NP$. Il problema \bar{X} , invece, ha una proprietà diversa in NP: per tutti s , abbiamo $s \in \bar{X}$ se e solo se per ogni t di lunghezza al massimo $p(|s|)$, abbiamo $C(s, t) = \text{no}$.

In termini concreti: data una serie insoddisfacente di clausole, come posiamo provare rapidamente che non esiste un'assegnazione soddisfacente? Questa è una definizione fondamentalmente diversa e non può essere aggirata semplicemente "invertendo" l'output del certificatore efficiente C, per produrre \bar{C} . Il problema è che l'"esiste t " nella definizione di NP è diventata un "per ogni t ", e questo è un problema.

Esiste una classe di problemi parallela a NP progettata per modellare questo specifico problema, chiamato **co-NP**.

Un problema X appartiene a co-NP se e solo se il problema complementare \bar{X} appartiene a NP.

Non sappiamo con certezza che NP e co-NP siano diversi; possiamo solo chiederci: **NP=co-NP?**

Dimostrare $NP = co-NP$ sarebbe un passo ancora più grande di provare $P = NP$, per il seguente motivo:

Se $NP \neq co-NP$, allora $P \neq NP$

Dimostrazione:

Dimostriamo l'affermazione contrapposta, ovvero: se $P = NP$ allora $NP = co-NP$.

Il punto è che P è chiusa per il complemento, quindi se $P = NP$, allora anche NP è chiusa per il complemento. Partendo dal presupposto $P=NP$, abbiamo:

$$X \in NP \Rightarrow X \in P \Rightarrow \bar{X} \in P \Rightarrow \bar{X} \in NP \Rightarrow X \in co-NP \quad \text{e} \quad X \in co-NP \Rightarrow \bar{X} \in NP \Rightarrow \bar{X} \in P \Rightarrow X \in P \Rightarrow X \in NP.$$

Quindi ne conseguirebbe che $NP \subseteq co-NP$ e $co-NP \subseteq NP$, da cui $NP = co-NP$.

CARATTERIZZAZIONE $NP \cap co-NP$:

Se il problema X è sia in NP e co-NP, allora se istanza è "si" esiste un certificato succinto, mentre se l'istanza è "no" esiste un "disqualifier" succinto.

Pertanto, si dice che i problemi che appartengono a questa intersezione $NP \cap co-NP$ abbiano una **buona caratterizzazione**, poiché esiste sempre un buon certificato per la soluzione.

Esempio:

Dato un grafo bipartito, esso ammette un perfect matching, e questo tipo di grafo è in $NP \cap co-NP$:

- se "si", possiamo fornire un perfect matching.
- se "no", possiamo fornire un insieme di nodi S tale che $|N(S)| < |S|$ (fornendo una metodologia o teorema per provarlo).

Nota: un grafo ha un perfect matching sse per ogni S risulta $|N(S)| \geq |S|$ (N sta per neighborhood, ovvero il numero di nodi vicini).

Naturalmente, si vorrebbe sapere se esiste un problema con una buona caratterizzazione ma nessun algoritmo del tempo polinomiale. Ma anche questa è una domanda aperta: $P = NP \cap co-NP$? Al momento possiamo solo osservare che $P \subseteq NP \cap co-NP$.

L'opinione generale sembra alquanto contrastata su questa domanda. In parte, questo è perché ci sono molti casi in cui è stato riscontrato che un problema ha una buona caratterizzazione non banale; e poi (molti anni dopo) si scoprì anche che aveva un algoritmo polinomiale.

Adesso vediamo 2 problemi appartenenti a $NP \cap co-NP$:

Teorema. PRIMES è in $NP \cap co-NP$.

Dim. Già sappiamo che PRIMES è in co-NP. Basta provare che PRIMES è in NP.

Teorema. Un intero dispari s è primo sse esiste intero $1 < t < s$ t.c.

$$\begin{aligned} t^{s-1} &\equiv 1 \pmod{s} \\ t^{(s-1)/p} &\not\equiv 1 \pmod{s} \\ \text{per tutti i divisori primi } p \text{ di } s-1 \end{aligned}$$

Input. $s = 437,677$
Certificato. $t = 17, 2^2 \times 3 \times 36,473$

↑
Fattorizzazione di $s-1$: richiede certificati per assicurare che 2, 3 e 36,473 sono primi

Il certificatore.

- Controlliamo $s-1 = 2 \times 2 \times 3 \times 36,473$.
- Controlliamo $17^{s-1} \equiv 1 \pmod{s}$.
- Controlliamo $17^{(s-1)/2} \equiv 437,676 \pmod{s}$.
- Controlliamo $17^{(s-1)/3} \equiv 329,415 \pmod{s}$.
- Controlliamo $17^{(s-1)/36,473} \equiv 305,452 \pmod{s}$.

FACTORIZE. Dato intero x , trova fattorizzazione.

FACTOR. Dati due interi x e y , x ha un fattore minore di y ?

Teorema. FACTOR è in $NP \cap co-NP$.

Dim.

- Certificato: un fattore p di x minore di y .
- Disqualifier: fattorizzazione di x (senza fattore minore di y), e certificato che ogni fattore è primo

8.5 PROBLEMI DI SEQUENCING

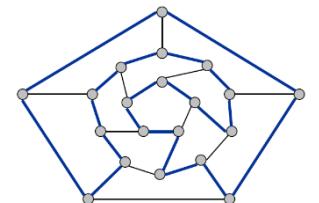
Con questi problemi troviamo se esiste, una **sequenza** di elementi che risolve il problema dato.

HAMILTONIAN CYCLE PROBLEM:

Dato un grafo $G=(V, E)$, diciamo che un ciclo C in G è un ciclo Hamiltoniano se visita ogni vertice esattamente una volta. In altre parole, costituisce un "giro" di tutti i vertici, senza ripetizioni.

Il problema del ciclo hamiltoniano è:

Dato un grafo *non orientato* $G=(V, E)$, esiste un ciclo semplice Γ che contiene ogni nodo in V ?



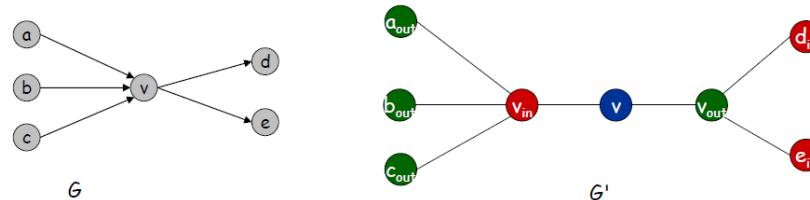
DIR-HAM-CYCLE:

Consideriamo una versione orientata di HAM-CYCLE: Dato un grafo *orientato* $G=(V, E)$, esiste un ciclo orientato semplice Γ contenente ogni nodo in V ?

Vogliamo far vedere che **DIR-HAM-CYCLE** \leq_p **HAM-CYCLE**.

Dimostrazione:

Dato un grafo orientato $G = (V, E)$, costruiamo un grafo non orientato G' .



Dobbiamo mostrare che **G ha un ciclo Hamiltoniano se e solo se G' lo ha anche esso.**

\Rightarrow Ciascun nodo u di G , eccetto s e t , viene sostituito da una tripla di nodi u_{in} , u , u_{out} in G' , collegati tra loro tramite archi, mentre i nodi s e t verranno sostituiti rispettivamente con s_{out} e t_{in} .

I nodi u e v in G , che sono collegati da un arco, in G' saranno collegati da u_{out} e v_{in} come in figura sopra, ciò completa la costruzione in G .

Per dimostrare che ciò funziona, facciamo vedere che un cammino Hamiltoniano in G , ad esempio $s-u-v-t$, in G' sarà $s_{out}-u_{in}-u-u_{out}-v_{in}-v-v_{out}-t_{in}$.

\Leftarrow Affermiamo che qualsiasi cammino Hamiltoniano in G' da s_{out} a t_{in} deve andare da una tripla di nodi ad un'altra tripla, eccetto per l'inizio e la fine, perché qualsiasi cammino di questo tipo ha un cammino Hamiltoniano corrispondente in G .

Per confermare ciò, se si inizia il cammino da s_{out} e lo si segue, non è possibile andare al di fuori delle varie triple, fino a t_{in} .

Mostriamo ora che **3-SAT** \leq_p **DIR-HAM-CYCLE**, e proviamo che quest'ultimo è NP-Completo.

Innanzitutto, mostriamo che DIR-HAM-CYCLE è in NP. Dato un grafo diretto $G=(V, E)$ e un certificato, il quale mostra che esiste una soluzione, ovvero l'elenco ordinato dei vertici su un ciclo hamiltoniano. Potremmo verificare, in tempo polinomiale, che l'elenco dei vertici contenga ogni vertice esattamente una volta e che ogni coppia consecutiva nell'ordinamento sia unita da un arco, ciò stabilirebbe che definisce un ciclo Hamiltoniano.

Dimostrazione:

Consideriamo un'istanza arbitraria di 3-SAT, con le variabili x_1, \dots, x_n e clausole C_1, \dots, C_k , essenzialmente, possiamo impostare i valori delle variabili come vogliamo e ci vengono date tre possibilità per soddisfare ogni clausola.

Per ciascuna formula φ , facciamo vedere come costruire un grafo diretto G con due nodi, s e t , in cui esiste un cammino Hamiltoniano tra s e t se e solo se φ è soddisfacibile.

Iniziamo la costruzione con una formula 3cnf φ contenente k clausole: $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k)$, dove ciascun a, b, c è un letterale x_i o \bar{x}_i e siano x_1, \dots, x_l le l variabili di φ .

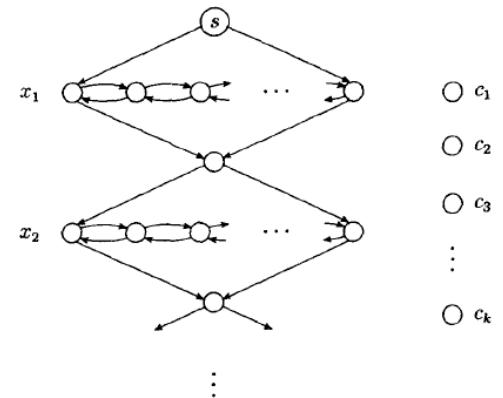
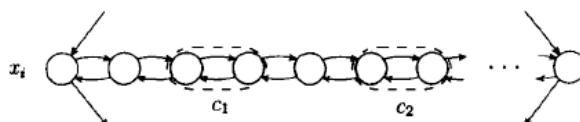
Rappresentiamo ciascuna variabile x_i , con una struttura di forma romboidale che contiene una riga orizzontale di nodi.

Rappresentiamo ciascuna clausola di φ con un singolo nodo $\odot c_j$.

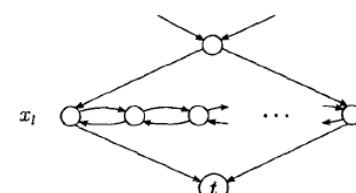
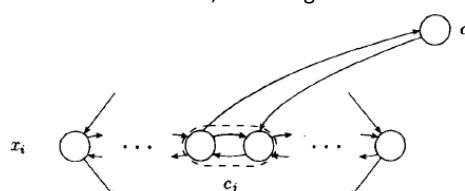
La figura a destra riporta la struttura globale di G , essa mostra tutti gli elementi di G e le loro relazioni, eccetto le relazioni delle variabili con le loro clausole.

Ciascuna struttura romboidale contiene una riga orizzontale di nodi collegati tramite archi orientati in entrambe le direzioni, questa riga contiene $3k+1$ nodi in aggiunta ai due nodi alle estremità del rombo.

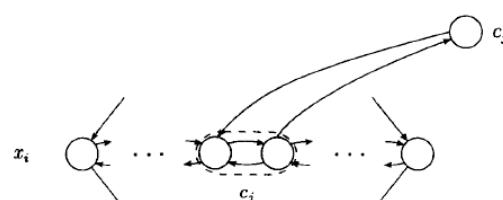
Questi nodi sono raggruppati in coppie adiacenti, una per ciascuna clausola, con nodi separati aggiuntivi tra le coppie, come segue:



Se la variabile x_i è presente nella clausola c_j , aggiungiamo due archi dalla coppia j -esima nell' i -esimo rombo al j -esimo nodo della clausola, come segue:

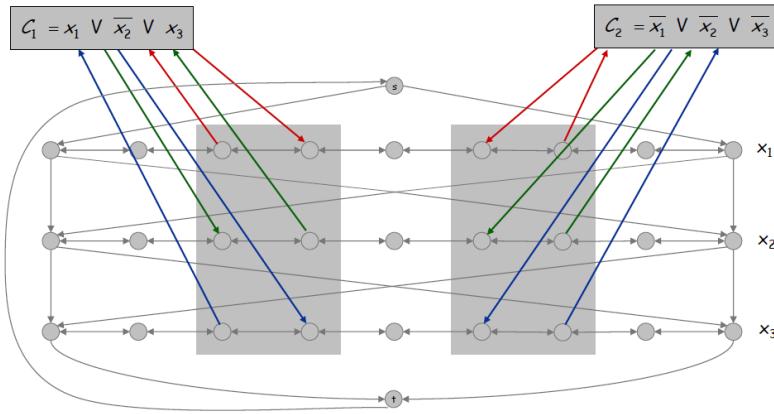


Se \bar{x}_i è presente nella clausola c_j , aggiungiamo due archi dalla coppia j -esima nell' i -esimo rombo al j -esimo nodo della clausola, come segue:



Dopo aver aggiunto tutti gli archi alle rispettive occorrenze, la costruzione di G è completa. Per far vedere che funziona tale costruzione, dimostriamo che: Data un'istanza di 3-SAT, costruiamo un'istanza di **DIR-HAM-CYCLE** ammette un circuito Hamiltoniano se e solo se φ è soddisfacibile.

Mostriamo prima un esempio complessivo per includere tutto ciò che è stato detto:



- (\Leftarrow) Supponiamo che φ sia soddisfacibile, per mostrare un cammino Hamiltoniano da s a t , ignoriamo per il momento i nodi clausola. Il cammino inizia da s , attraversa ciascun rombo in successione, e termina in t . Per raggiungere i nodi orizzontali in un rombo, il cammino può procedere in due direzioni, ovvero da sinistra a destra e viceversa, l'assegnamento che soddisfa φ determina quale scegliere dei due versi. Se $x_i = 1$ allora il cammino procede da sinistra a destra, altrimenti se $x_i = 0$ va da destra a sinistra. Possiamo dire quindi che per ogni clausola C_j , esiste una riga i -esima che attraversa nella direzione "corretta" (modalità descritta alla riga precedente) per includere il nodo C_j nel circuito.
- (\Rightarrow) Se G ha un ciclo Hamiltoniano da s a t , facciamo vedere un assegnamento che soddisfa φ . Se il cammino Hamiltoniano passa attraverso i rombi in ordine da quello più in alto a quello più in basso e muovendosi da sinistra a destra o viceversa, possiamo determinare quali valori avranno le variabili booleane, ovvero se il cammino procede da sinistra a destra allora $x_i = 1$, altrimenti se va da destra a sinistra $x_i = 0$. Nel caso dell'esempio sopra possiamo dire che le due clausole, per essere soddisfatte, x_2 dovrà essere uguale a 0, mentre le altre qualsiasi valore.

TRAVELING SALESMAN PROBLEM (Problema del commesso viaggiatore):

Probabilmente il sequencing problem più famoso è il "**Problema del commesso viaggiatore**".

Consideriamo un venditore che deve visitare n città etichettate v_1, v_2, \dots, v_n . Il venditore inizia nella città v_1 , la sua casa, e vuole trovare un **tour**, ovvero un ordine in cui visitare tutte le altre città e tornare a casa. Il suo obiettivo è trovare un tour che gli faccia percorrere la minor distanza totale possibile. Più formalmente, per ogni coppia ordinata di città (v_i, v_j) , specificheremo un numero non negativo $d(v_i, v_j)$ come distanza da v_i a v_j .

La ragione di ciò è rendere la nostra formulazione il più generale possibile. In effetti, il commesso viaggiatore si presenta naturalmente in molte applicazioni in cui i punti non sono città e il viaggiatore non è un commesso. Pertanto, dato l'insieme delle distanze, bisogna ordinare le città in un tour v_1, \dots, v_n , in modo da ridurre al minimo la distanza totale.

Da quanto detto ci chiediamo: **Dato un insieme di n città e una funzione di distanza $d(u, v)$, esiste un tour di lunghezza $\leq D$?**

Per dimostrare quanto detto usiamo HAM-CYCLE, e mostriamo che **Traveling Salesman è NP-Completo**.

Dimostrazione:

È facile vedere che il commesso viaggiatore è in NP: il certificato è una permutazione delle città e un certificatore verifica che la durata del tour corrispondente sia al massimo il limite indicato D .

Mostriamo ora che **Hamiltonian Cycle \leq Traveling Salesman**.

Dato un grafo diretto $G = (V, E)$, definiamo la seguente istanza di Traveling Salesman: Abbiamo una città v per ogni nodo u del grafo G .

Definiamo $d(u, v) = 1$ se c'è un arco (u, v) in G , altrimenti sarà uguale a 2, più formalmente:

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$$

Ora affermiamo che **G ha un ciclo Hamiltoniano se e solo se c'è un tour di lunghezza al massimo n nella nostra istanza di Traveling Salesman**.

(\Rightarrow) Se G ha un ciclo Hamiltoniano, allora questo ordinamento delle città corrispondenti definisce un giro di lunghezza n .

(\Leftarrow) Al contrario, supponiamo che ci sia un tour di lunghezza al massimo n , ad esempio A-B-C-D-(e poi torno ad A).

La durata di questo tour è la somma degli n termini, ognuno dei quali è almeno 1; quindi deve essere il caso che tutti i termini sugli archi siano uguali a 1. Quindi ogni coppia di nodi in G , che corrispondono a città consecutive nel tour, deve essere collegata da un arco; ne consegue che l'ordinamento di questi nodi corrispondenti deve formare un ciclo Hamiltoniano.

8.6 PROBLEMI DI PARTITIONING

Adesso consideriamo dei problemi fondamentali di partizionamento, in cui cerchiamo dei modi per dividere una raccolta di oggetti in sottoinsiemi.

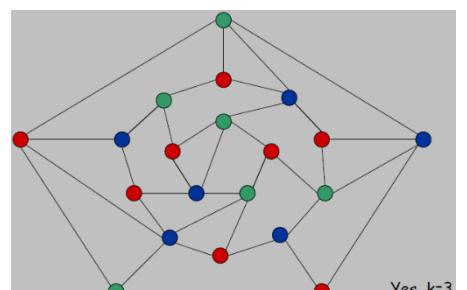
Un problema di questo tipo è quello di trovare un grafo bipartito, ovvero un grafo dove l'insieme dei suoi vertici si può partizionare in due sottoinsiemi tali che ogni vertice di una di queste due parti è collegato solo a vertici dell'altro insieme, questo è risolvibile in tempo polinomiale.

Ma le cose diventano più complicate quando passiamo da **coppie** ordinate a **triple** ordinate.

Per semplificare la comprensione del problema di trovare triple ordinate in un dato grafo, aiutiamoci con i colori.

La colorazione del grafo si riferisce allo stesso processo su un grafo G non orientato, coi nodi che svolgono il ruolo delle regioni da colorare e gli archi che rappresentano le coppie vicine. Cerchiamo di assegnare un colore a ciascun nodo di G in modo che se (u, v) è un arco, u e v saranno colorati in modo diverso.

Più formalmente, una k colorazione di G è una funzione $f: V \rightarrow \{1, 2, \dots, k\}$ in modo che per ogni arco (u, v) , abbiamo $f(u) \neq f(v)$ (i colori qui sono chiamati 1, 2, ..., k , e la funzione f rappresenta la scelta di un colore per ogni nodo). Questo introduce un problema noto, ovvero **k -COLOR**:



Dato un grafo non orientato G esiste un modo di colorare i nodi usando k colori in modo che nodi adiacenti NON hanno lo stesso colore?

Se consideriamo $k=2$, questo è il problema di determinare se un grafo G è bipartito, ma se passiamo a $k=3$, le cose diventano molto più difficili.

In effetti, il caso del grafo tricolore è già un problema molto difficile e possiamo dimostrare che **3-COLOR è NP-Completo**.

Dimostrazione:

È facile capire perché il problema si trova in NP. Dati G e k , un certificato che afferma che la risposta è "sì" è una colorazione k : si può verificare in tempo polinomiale che al massimo vengono usati i k colori e che nessuna coppia di nodi uniti da un arco riceve lo stesso colore.

3-COLOR è un problema difficile da mettere in relazione con altri problemi NP-completi che abbiamo visto.

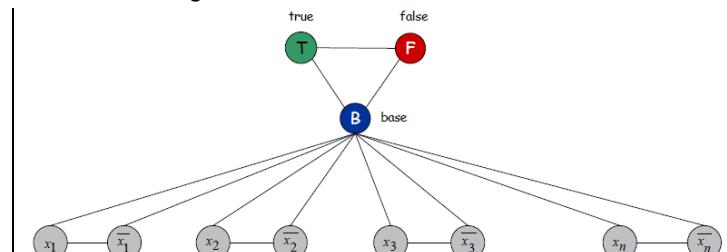
Quindi, ancora una volta, torneremo su 3-SAT, mostrando che **3-SAT \leq_p 3-COLOR**.

Data un'istanza arbitraria di 3-SAT, con variabili x_1, \dots, x_n e clausole C_1, \dots, C_k , lo risolveremo usando un oracolo per 3-COLOR.

Definiamo tre "nodi speciali" T , F e B , che chiamiamo Vero, Falso e Base. Adesso costruiamo il grafo:

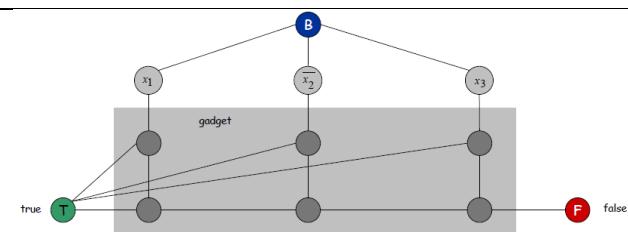
1. Creiamo un nodo per ogni letterale.
2. Creiamo 3 nuovi nodi T , F , B , assicurandoci che questi 3 abbiano colori diversi, connettiamoli in un triangolo, e colleghiamo ogni letterale negato e non, al nodo B .
3. Connessioni ogni letterale alla sua negazione, quest'ultimo dovrà avere un colore diverso rispetto al nodo B e al nodo letterale corrispondente non negato.

Ogni nodo letterale ha valore T o F.



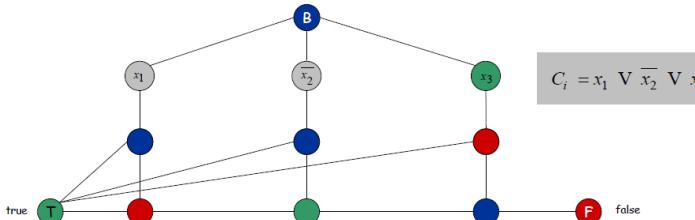
Tramite la riduzione via "Gadget", aggiungiamo un gadget di 6 nodi e 13 archi, come in figura a destra →

Il gadget assicura che almeno un letterale in ogni clausola è T



Consideriamo una clausola $C = (x_1 \vee \bar{x}_2 \vee x_3)$ e supponiamo che tutte le variabili siano tutte False, avremmo la figura a destra →

Quindi troviamo un'istanza "no", ma se almeno uno dei 3 fosse uguale a True avremmo risolto, e se proviamo a colorare troveremmo la soluzione, ovvero:



Possiamo completare la costruzione: iniziamo con il grafo G definito per primo, e per ogni clausola nell'istanza 3-SAT, alleghiamo un sottografo a sei nodi, come in seconda figura. Dimostriamo ora che **l'istanza 3-SAT è soddisfacente se e solo se G ha una 3-COLOR**.

(⇒) In primo luogo, supponiamo che ci sia un'assegnazione soddisfacente per l'istanza 3-SAT. Definiamo una colorazione di G colorando prima **Base**, **Vero** e **Falso** arbitrariamente con i tre colori, quindi, per ogni i , assegnando a v_i il colore **Vero** se $x_i = 1$ e il colore **Falso** se $x_i = 0$.

Infine, è ora possibile estendere questa 3-COLOR in ciascun sottografo della clausola a sei nodi, risultando una 3-COLOR in tutto G .

(⇐) Al contrario, supponiamo che G abbia un 3-COLOR. In questa colorazione, a ciascun nodo v_i viene assegnato il colore **True** o **False**; impostiamo la variabile x_i di conseguenza. Ora affermiamo che in ciascuna clausola dell'istanza 3-SAT, almeno uno dei termini nella clausola ha il valore di verità 1. In caso contrario, tutti e tre i nodi corrispondenti hanno il colore **Falso** nella 3-COLOR di G e, come abbiamo visto sopra, non vi è alcuna 3-COLOR del sottografo della clausola corrispondente, ed abbiamo quindi una **contraddizione**.

8.7 PROBLEMI NURERICI

Consideriamo ora alcuni problemi computazionalmente difficili che coinvolgono operazioni aritmetiche sui numeri.

SUBSET-SUM:

Il nostro problema di base in questo caso sarà la somma dei sottoinsiemi:

Dati dei numeri naturali w_1, \dots, w_n e intero W , esiste un sottoinsieme la cui somma è esattamente W ?

Esempio:

$$\{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}, W = 3754 \quad \rightarrow \quad \text{si: } 1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754$$

Poiché i numeri interi vengono forniti in rappresentazione binaria, o in base 10, la quantità W è realmente esponenziale nella dimensione dell'input; Quindi una domanda di base è: il SUBSET SUM può essere risolta da un algoritmo polinomiale?

Il seguente risultato suggerisce che non è probabile che ciò avvenga, e possiamo dimostrare che **SUBSET SUM è NP-Completo**.

Dimostrazione:

Mostriamo innanzitutto che SUBSET-SUM è in NP. Dati i numeri naturali w_1, \dots, w_n e un obiettivo W , un certificato attestante che esiste una soluzione sarebbe il sottoinsieme w_1, \dots, w_k che si suppone che la somma è W . Nel tempo polinomiale, possiamo calcolare la somma di questi numeri e verificare che sia uguale a W . Una volta dimostrato che SUBSET-SUM è in NP, mostriamo che **3-SAT \leq_p SUBSET-SUM**.

Sia un'istanza ϕ di 3-SAT con variabili x_1, \dots, x_n e clausole c_1, \dots, c_k , dimostriamo che **SUBSET-SUM ha una soluzione se e solo se ϕ è soddisfacibile**.

(⇐) Data un'istanza ϕ con n variabili e k clausole, formiamo $2n+2k$ interi, ognuno di $n+k$ cifre, dobbiamo mostrare che ϕ è soddisfacibile se presi dei sottoinsiemi la somma di questi risulta W .

Supponiamo che abbiamo queste clausole:

$$\begin{aligned}C_1 &= \bar{x} \vee y \vee z \\C_2 &= x \vee \bar{y} \vee z \\C_3 &= \bar{x} \vee \bar{y} \vee \bar{z}\end{aligned}$$

Possiamo osservare che abbiamo $n=k=3$ e quindi $2(3)+2(3)=12$ interi (quindi 12 righe della tabella). Adesso costruiamo la tabella a destra, ha una colonna per ogni variabile e clausole date, ed una riga per ogni letterale ed infine abbiamo altre 6 (in realtà sono $2k$) righe non indicizzate.

Al suo interno metteremo il valore 1 se abbiamo ad esempio la colonna x in corrispondenza con le righe x e $\neg x$, in più un altro valore 1 in corrispondenza alle clausole di appartenenza delle variabili, ad esempio x appare nella clausola C_2 .

Per quanto riguarda la parte non indicizzata della tabella, le colonne corrispondenti alle variabili saranno 0, mentre quelle corrispondenti alle clausole avranno dei numeri fissi, le prime due righe della seconda parte della tabella, solo la colonna corrispondente alla clausola C_1 avrà valori, ovvero 1 con 2 sottostante, e così via. Per finire, l'ultima riga, ovvero W , ci saranno degli 1 in corrispondenza delle variabili e 4 in corrispondenza delle clausole.

Da questa tabella ricaviamo dei numeri interpretando le righe (numeri posti al lato destro della tb).

Supponiamo che la formula è soddisfacibile prendendo $y=z=\text{true}$ e $x=\text{false}$ (e quindi $\neg x=\text{true}$) e prendiamo i numeri corrispondenti in tabella.

Adesso consideriamo i numeri posti a destra della tabella corrispondenti al nostro assegnamento di verità (righe in verde) e sommiamoli provando ad ottenere l'ultimo numero ovvero 111,444, se non riusciamo ad ottenere quest'ultimo soltanto sommando i numeri presi dalle variabili dell'assegnamento di verità, possiamo prendere i numeri della seconda parte della tabella, per arrivare al numero obiettivo.

- (\Rightarrow) Viceversa, se ho dei sottoinsiemi di interi, e quindi una soluzione a SUBSET-SUM, considero le righe corrispondenti e vediamo a che letterale corrisponde. Il numero 4 nell'ultima riga, ovvero W , assicura che deve esserci almeno un 1 tra le righe selezionate, ma questo mi dice che c'è almeno un letterale posto a true e quindi la clausola corrispondente è soddisfatta.

SCHEDULE-RELEASE-TIMES:

La difficoltà di SUBSET SUM può essere utilizzata per stabilire la difficoltà di SCHEDULE-RELEASE-TIMES.

Supponiamo di avere una serie di n jobs che devono essere eseguiti su una singola macchina.

Ogni lavoro i ha un *release time* r_i quando è disponibile per la prima elaborazione, una *deadline* d_i entro la quale deve essere completato, e un *processing time* t_i . Supponiamo che tutti questi parametri siano numeri naturali.

Per essere completato, il lavoro i deve essere assegnato ad uno slot contiguo di unità di tempo t_i da qualche parte nell'intervallo $[r_i, d_i]$. La domanda è: **possiamo pianificare tutti i lavori in modo tale che ciascuno venga completato entro la sua scadenza?**

Dimostriamo che SCHEDULE-RELEASE-TIMES è NP-Completo.

Dimostrazione:

Data un'istanza del problema, un certificato risolvibile sarà una specifica *release time* per ciascun lavoro. Potremmo quindi verificare che ogni lavoro venga eseguito per un intervallo di tempo distinto, tra la sua *release time* e la sua *deadline*. Quindi il problema è in NP.

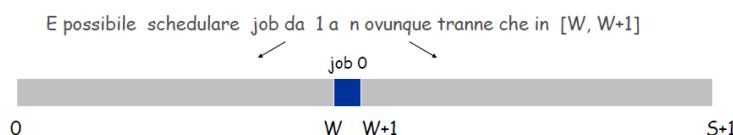
Mostriamo ora che $\text{SUBSET-SUM} \leq_p \text{SCHEDULE-RELEASE-TIMES}$.

Pertanto, si consideri un'istanza di SUBSET-SUM con numeri w_1, \dots, w_n e un numero W .

Innanzitutto, creiamo n jobs con *processing time* $t_i=w_i$, *release time* $r_i=0$ (tutti i jobs arrivano allo stesso tempo all'inizio quindi sono tutti immediatamente disponibili), e nessuna deadline, ma poniamo $d_j = 1 + \sum_{j=1}^n w_j$.

Da quanto detto potremmo mettere tutti i jobs come vogliamo, non incontrando conflitti durante la schedulazione.

A questo punto, creiamo un job 0 con $t_0=1$, *release time* $r_0=W$ e *deadline* $d_0=W+1$, ma questo è problematico in quanto ha $r_0=W$ e $d_0=W+1$, quindi il job 0 lo si deve schedulare quando arriva al tempo W ed eseguirlo immediatamente perché deve finire in tempo $W+1$. Ma adesso è come se il nostro intervallo fosse diviso in due, la parte sinistra da 0 a W , quindi ho W unità di tempo, e a destra c'è il resto, dovendo terminare entro $1 + \sum_{j=1}^n w_j$ (che indichiamo come $S+1$ per semplicità). Seguendo questa impostazione, abbiamo che le nostre unità di tempo adesso sono S , ovvero pari al tempo che abbiamo bisogno per andare a eseguire tutti i job, ma lo possiamo fare solo se riusciamo a distribuire i job metà prima del job 0 e l'altra metà dopo.



Mostriamo che SUBSET-SUM ha una soluzione se e solo se troviamo uno SCHEDULE-RELEASE-TIMES fattibile.

- (\Leftarrow) Quindi dobbiamo dividere gli interi w_1, \dots, w_n in qualche modo in un sottoinsieme, in modo tale che quest'ultimo sia eseguibile prima che arrivi il job 0 ed il resto sarà eseguito dopo che job 0 è finito, ma questa è proprio ciò che si deve fare per risolvere SUBSET-SUM su un insieme di interi.
- (\Rightarrow) Quindi se l'istanza di SUBSET-SUM è un'istanza "si" allora posso dividere w_1, \dots, w_n in due parti, di cui una somma W , che corrisponde a tutti i jobs eseguiti prima del job 0, mentre i restanti sono eseguiti dopo il job 0, d'altra parte se abbiamo che i jobs possono essere eseguiti e quindi schedulati prima di job 0 ed il loro processing time totale è W , vuol dire che esiste un sottoinsieme dei w_i la cui somma è W .

	x	y	z	C_1	C_2	C_3	
x	1	0	0	0	1	0	100,010
$\neg x$	1	0	0	1	0	1	100,101
y	0	1	0	1	0	0	10,100
$\neg y$	0	1	0	0	1	1	10,011
z	0	0	1	1	1	0	1,110
$\neg z$	0	0	1	0	0	1	1,001
	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
W	1	1	1	4	4	4	111,444