

Informatica Teorica

41169 - 06 CFU

   Disclaimer:   

questi appunti sono un mix, buona parte sono sbobinature delle lezioni del prof Malizia dell'A.A. 2025 ma sono stati integrati in parte anche gli appunti di Marco Calautti forniti dallo stesso prof. Malizia.

(agg. a Luglio 2025) Sono gratuitamente pubblicati su
<https://dynamik.vercel.app/informatica-teorica/appunti?from=informatica>

Se trovate errori fatemelo pure sapere scrivendo a

 alberto.zuccari@studio.unibo.it oppure su

 telegram [@b3rt0nes](https://t.me/b3rt0nes)

SOMMARIO

1	Problemi, Algoritmi e un assaggio di indecidibilità	5
1.1.1	Problemi di ricerca e di decisione -----	5
1.1.2	Esempio di problema incalcolabile-----	6
2	Alfabeti, stringhe, linguaggi e Macchine di Turing	8
2.1.1	Problemi di decisione come linguaggi-----	9
2.2	Introduzione alle Macchine di Turing.....	9
2.2.1	Programmazione delle Macchine di Turing -----	11
2.2.2	Macchine multitraccia -----	14
2.2.3	Macchine multinastro-----	15
2.3	Macchine di Turing non deterministiche.....	19
2.3.1	Tesi di Church-Turing-----	21
2.3.2	Riconoscimento di linguaggi tramite macchine non deterministiche -----	21
2.4	Macchine di Turing Universali	26
2.5	Problemi ricorsivi e ricorsivamente enumerabili	28
2.5.1	Proprietà sui linguaggi -----	29
3	Linguaggio universale e problema della fermata	30
4	Riduzioni	33
4.1	Post Correspondence Problem (PCP)	37
4.2	Tiling Problem.....	43
4.3	La classe CO-RE	48
5	Teorema di Rice	51
5.1.1	Esempi di applicazione-----	54
6	Complessità strutturale	56
6.1.1	Classe DTime -----	57
6.1.2	Classe NTime -----	58
6.2	Riduzioni polinomiali	61
6.3	Problemi NP-Completi	66
6.3.1	Independent Set (IS)-----	66
6.3.2	Vertex Cover (VC) -----	69
6.3.3	Clique-----	71
6.3.4	Dominating set (DS)-----	72
6.3.5	Definizione differente della classe NP-----	73
6.3.6	Colorabilità dei grafi (Col) -----	76
6.3.7	Exact cover (EC)-----	78
6.3.8	Knapsack (Kn) -----	80

6.4	Teorema di Cook.....	82
6.5	Complemento di NP	87
6.6	Exp e NExp	92
7	Complessità spaziale.....	94
7.1	Logspace L e NL.....	95
7.1.1	NL-completezza-----	97
7.1.2	Reach è NL-complete-----	98
7.2	Teorema di Savitch	100
7.2.1	Reach \in DSpace($\log^2 n$) -----	100
7.2.2	Proof Savitch -----	101
8	MdT con Oracoli	103
8.1.1	Classi ad oracolo-----	105
8.1.2	Gerarchia polinomiale-----	106
8.2	Problemi di ottimizzazione e classi funzionali	109
8.2.1	Travelling Salesman Problem (TSP) -----	110

1 PROBLEMI, ALGORITMI E UN ASSAGGIO DI INDECIDIBILITÀ

Durante gli studi abbiamo sempre analizzato gli algoritmi. Un algoritmo è una sequenza finita di istruzioni basilari (esprese in qualche linguaggio di programmazione), che *in un tempo finito*, risolve un certo *problema*, dati degli input. Solitamente ci viene già dato un problema e ci soffermiamo a ideare un algoritmo che lo risolva.

Esempi di problemi sono: “dato un array di interi ν e un intero x , trovare la posizione dell’array in cui compare x ” oppure “dato un grafo orientato $G = (V, E)$, con V un insieme di nodi connessi da un percorso presente in E , che parte dal nodo s e finisce al nodo t , controllare se esiste un percorso da s a t nel grafo G .

In questo corso sposteremo il focus dagli algoritmi sui problemi. In particolare, questo corso è diviso in due parti

| La prima è *teoria della computabilità*, che mira a rispondere a domande come:

“Questo problema è risolvibile da un algoritmo?

(in altre parole, esiste un algoritmo che *per ogni input*, fornisce la risposta corretta al problema?)”

Problemi come quelli di cui sopra vengono chiamati *calcolabili*.

Un altro tipo di domande che la teoria della computabilità mira a risolvere è:

“Se due problemi non sono computabili, possiamo lo stesso concludere
che uno è più difficile dell’altro, e qual è quello difficile?”

La prima domanda può sembrare strana, essendo abituati a risolvere tutto con un programma, un’app, ..., ed è strano credere che esistano problemi che non possono essere risolti da un algoritmo.

Come andremo a formalizzare più tardi, questo tipo di problemi esistono.

| La seconda è *teoria della complessità*: assumiamo di sapere che un problema è effettivamente computabile. Chiaramente, il fatto che esista un algoritmo che *per ogni input* fornisce la risposta giusta non dice molto su quanto *difficile* sia il problema, per esempio, che risorse utilizza in termini di tempo e memoria per risolvere il problema?

Una delle domande principali sulla teoria della complessità chiede:

“Cosa rende un problema più difficile di altri in termini di utilizzo delle risorse?”

1.1.1 Problemi di ricerca e di decisione

Abbiamo due principali tipologie di problemi:

- Problemi di ricerca: quei problemi che chiedono di trovare/costruire qualche soluzione/output per un input dato. Per esempio, trova/costruisci la posizione (int) per cui un numero x appare in ν .
- Problemi di decisione: questi sono problemi le cui risposte sono solamente “sì” o “no”. Per esempio, dato un grafo G e i nodi s e t , stabilire se esiste un percorso da s a t in G .

L’input di un problema è anche detto *istanza* del problema.

Spesso i problemi di ricerca hanno un problema di decisione corrispondente, e viceversa. Per esempio, dato un vettore c e un numero x potremmo chiedere soltanto “ x appare in qualche posizione di ν ? ”.

Chiaramente un problema di ricerca è difficile almeno quanto la sua versione di decisione, perché possiamo risolvere il problema di decisione trovando una soluzione per quello di ricerca. Per esempio, se troviamo in che posizione x appare in ν , allora la risposta è “sì”, altrimenti, se nessuna posizione viene trovata, “no”.

Quindi tipicamente, per semplificare la discussione ci si concentra sui problemi di decisione, dato che, se non si riesce a risolvere un problema di decisione allora non si riuscirà a risolvere la sua versione di ricerca. Quindi ci concentreremo perlopiù su problemi di decisione.

1.1.2 Esempio di problema incalcolabile

Andremo ora a vedere un esempio di un problema incalcolabile. In particolare, nel concentrarci su di un problema di decisione utilizzeremo la parola *decidibile* al posto di computabile, e *indecidibile* invece che incomputabile, ma essenzialmente sono la stessa cosa.

Immaginiamo di star lavorando in Python (o C++/Java, non importa), e stiamo scrivendo tutti i nostri algoritmi in Python. Consideriamo questa la semplice funzione che conta quanti spazi ci sono in una stringa:

```
1. def countSpaces(myString):
2.     count = 0
3.     i = 0
4.     while i < len(myString):
5.         if myString[i] == ' ':
6.             count = count + 1
7.             i = i + 1
8.     return count
```

Vedi il problema con questa funzione? L'autore ha scritto `i = i + 1` dentro l'`if` piuttosto che fuori. Ciò significa che se la stringa in input è “abc d”, la funzione piuttosto che fermarsi e ritornare quanti spazi non si fermerà, e quindi l'indice `i` non verrà mai incrementato.

Sarebbe bello se avessimo un algoritmo che sia capace di trovare questo tipo di bug, anche per funzioni semplici come questa che prende in input una sola stringa.

Quindi chiamiamo funzioni Python che prendono solo una stringa *one-string functions*. Vorremmo avere un algoritmo che sia capace di risolvere il seguente problema:

PROBLEMA:	HALTING
INPTU:	Una stringa P che rappresenti il codice di una one-string function e.g. $P = \text{"def someFunc(someString)..."}$, e un'altra stringa I
DOMANDA:	La funzione P si ferma quando eseguita su I come input?

Chiaramente si può provare a sviluppare un algoritmo che risolva il problema e sicuramente per problemi basilari come `countSpaces` non è difficile scriverlo.

Comunque quello che andremo a dimostrare è che non importa quanto ti ci spendi e quanto arguto il tuo algoritmo sia, ci sarà sempre qualche input P e I per il quale il tuo algoritmo non sarà in grado di restituire la risposta corretta al problema della fermata.

Quindi, essenzialmente, andremo a provare che:

HALTING è indecidibile.

(in altre parole, non esiste un algoritmo che *per ogni input P e I , dia la risposta corretta al problema della fermata*)

Dimostrazione informale. La dimostrazione che andremo a dare è per assurdo. Assumiamo per assurdo che abbiamo un algoritmo e chiamiamolo HaltChecker, che dato un programma *qualunque* P di una one-string function, e una stringa input I restituisca “sì” se la funzione dato I in input si ferma, e “no” altrimenti.

Se abbiamo questa procedura, possiamo chiamarla da un'altra funzione, tipo:

```
1. def reverse (P):
2.     halts = haltChecker(P; P)
3.     if (halts):
4.         while (true) // ciclo infinito
5.     else:
6.         pass          // termina
```

Questa procedura la chiamo codeReverse;

Se poi eseguiamo il comando:

```
1. reverse(codeReverse);
```

Che cosa otterremo? Due casi:

- > STOP, la procedura si ferma ⇒ vuol dire che halts = false
- > LOOP, la procedura continua all'infinito ⇒ vuol dire che halts = true

Abbiamo quindi un assurdo, e quindi abbiamo dimostrato che una funzione haltChecker valida per ogni input non può esistere.



2 ALFABETI, STRINGHE, LINGUAGGI E MACCHINE DI TURING

Per studiare i problemi e le loro proprietà in maniera rigorosa dobbiamo definire formalmente l'idea di problema. Intuitivamente quando pensiamo a un problema pensiamo a qualche tipo di input dato e qualche processo da effettuare col dato input. Questo processo può produrre un output (problema di ricerca) o produrre una risposta "sì"/"no".

In maniera più astratta, possiamo pensare agli input dei nostri problemi come *stringhe* di qualche alfabeto codificando l'input al problema.

Per esempio, consideriamo il problema, *dato un array ν di interi e un intero x , che ci chiede di trovare la posizione di x in ν* . L'input di questo problema può essere rappresentato con una stringa tipo:

$$\langle [1,4,7,1,2], 2 \rangle$$

Dove $\nu = [1,4,7,1,2]$ e $x = 2$. Similmente l'output del problema (che sarà un intero) può essere rappresentato con una stringa sull'alfabeto $\{0,1,\dots,9\}$. Quindi l'alfabeto generale per le stringhe in input e output del nostro problema sarà l'insieme di simboli:

$$\Sigma = \{[,], (,), ', 0, 1, \dots, 9\}.$$

Riprendiamo una notazione che sarà utile:

Un **alfabeto** è un insieme finito di simboli e di solito è denotato con Σ . Possiamo usare Σ^n per indicare tutte le stringhe di lunghezza $n \geq 0$ che utilizzano simboli dall'alfabeto Σ . Per esempio, se $\Sigma = \{a, b, c\}$, allora:

$$\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

Utilizziamo Σ^* per indicare l'insieme di tutte le stringhe possibili (di qualunque lunghezza finita, compresa la stringa vuota), utilizzando simboli dall'alfabeto Σ . Ovvero:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$$

Poiché la stringa vuota è vuota, quando vogliamo usarla esplicitamente nelle formule la indichiamo con ε . Quindi, se per esempio $\Sigma = \{0,1\}$, allora:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots\}$$

Ora possiamo definire formalmente cos'è un problema

DEF (Problema)

Un problema su un alfabeto Σ è una funzione della forma

$$f: \Sigma^* \rightarrow \Sigma^*$$

Quindi un problema viene definito come una funzione che mappa ogni input, codificato attraverso qualche stringa in Σ^* , all'output corrispondente (codificato in una stringa in Σ^*).

Notiamo che è possibile prendere qualunque stringa da Σ^* come input, ma chiaramente esistono stringhe che non rappresentano input validi per il problema che la funzione rappresenta.

Per risolvere questo problema dovremmo essere molto più precisi e specifici e dire che il dominio del problema non dev'essere Σ^* ma piuttosto un insieme $D \subseteq \Sigma^*$ che contenga input validi per il problema.

Ma ciò va a complicare la nostra analisi poiché non potremmo più trattare i problemi in maniera uniforme. Certamente avere problemi che prendono stringhe arbitrarie di Σ^* non ci da troppo fastidio, basta che se la codifica della stringa non va bene il programma dia un output di default "don't care".

DEF (Problema di decisione)

Un problema di decisione su un alfabeto Σ è una funzione della forma

$$f: \Sigma^* \rightarrow \{"si", "no"\}$$

esempio

PROBLEMA: RICERCABILITÀ

INPUT: Un grafo $G = (V, E)$ e due nodi $s, t \in V$

DOMANDA: Esiste un percorso da s a t ?

$$\text{RICERCABILITÀ}(w) = \begin{cases} 1 & \text{se esiste un percorso da } s \text{ a } t \text{ in } G \\ 0 & \text{altrimenti} \end{cases}$$

2.1.1 Problemi di decisione come linguaggi

Notiamo che poiché i problemi di decisione hanno come output solo “sì” o “no”, possiamo definire in modo equivalente un problema di decisione come l’insieme delle stringhe in Σ^* per le quali la risposta è “sì”.

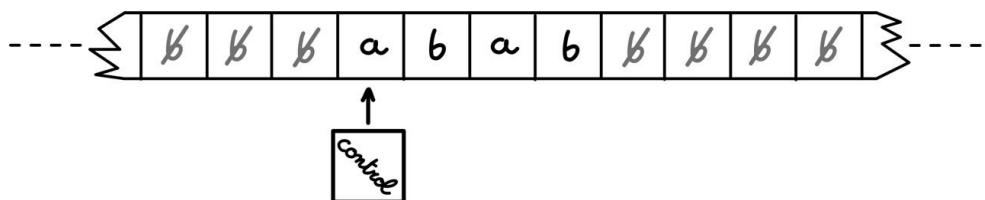
Per esempio, RICERCABILITÀ è l’insieme di tutte le stringhe in Σ^* della forma G, s, t dove G è un grafo orientato e s, t sono due nodi tali per cui esiste un percorso da s a t in G .

In altre parole, i problemi di decisione possono essere definiti come *linguaggi*, ovvero sottoinsiemi dell’intero insieme di tutte le possibili stringhe sull’alfabeto.

Come già detto prima, per semplicità, ci concentreremo solo su problemi di decisione, quindi sui linguaggi.

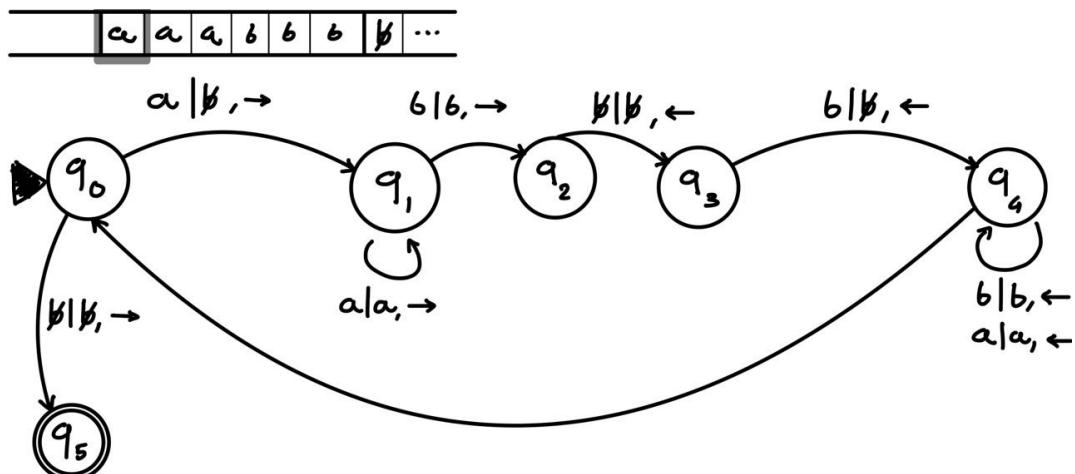
2.2 INTRODUZIONE ALLE MACCHINE DI TURING

Una macchina di Turing è una macchina astratta per modellare il calcolo. È diversa da un automa a stati finiti, perché la Macchina di Turing ha un nastro infinito in entrambe le direzioni; questo nastro è diviso in celle con i simboli da leggere all’interno, abbiamo poi una testina che legge i simboli e ci scrive sopra



Quando la macchina parte con l’esecuzione il nastro contiene la stringa in input e a destra e alla sinistra sua il nastro è riempito con un simbolo speciale \emptyset chiamato *Blank*. All’inizio dell’esecuzione la testina è posizionata all’inizio della stringa in input. Consideriamo il seguente problema: decidere questo linguaggio:

$$L = \{ a^m b^m \mid m \geq 0 \}$$



DEF (Macchina di Turing)

Una macchina di Turing è una tupla

$$M = \langle \Sigma, \Gamma, \delta, Q, q_0, F \rangle$$

1. Σ è l'alfabeto in input
2. Γ è l'alfabeto di nastro $\Rightarrow \Gamma \supseteq \Sigma$
3. $\delta \in \Gamma$
4. Q è l'insieme finito degli stati
5. q_0 è lo stato iniziale
6. $F \subseteq Q$ è l'insieme degli stati finali
7. δ è la funzione di transizione e $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$

DEF (Configurazione)

Una configurazione è una notazione per rappresentare lo stato di un automa

$$\boxed{aq_1bb} \quad \text{oppure} \quad \boxed{abbq_1\delta}$$

DEF (legal successor)

Date due configurazioni c_1 e c_2 per M diciamo che c_2 è il *legal successor* di c_1 rispetto a M

$$c_1 \vdash c_2$$

Se c_2 è una configurazione che M raggiunge partendo da c_1 e facendo un solo passo secondo δ

$$\begin{array}{ccc} \boxed{q_0aab} & \vdash & \boxed{q_aabb} \\ c_1 & & c_2 \end{array}$$

DEF (configurazione iniziale)

La *configurazione iniziale* di M su una semantica $w = w_1 \dots w_n$ è

$$\boxed{q_0w_1 \dots w_n}$$

DEF (configurazione finale)

Una *configurazione finale* per M è una configurazione che non ammette un legal successor secondo δ

DEF (configurazione accettante)

Una *configurazione accettante* è una configurazione finale il cui stato $\in F$

DEF (computazione parziale)

Una *computazione parziale* di M è una sequenza di configurazioni c_1, \dots, c_m tale che

$$c_1 \vdash_M c_2 \vdash_M \dots \vdash_M c_m$$

DEF (computazione completa)

Una *computazione completa* di M su w è una computazione parziale di M , $c_1 \dots c_m$ tale che c_1 è configurazione iniziale di M su w e c_m è una configurazione finale.

DEF (computazione accettante)

Una computazione di M è *accettante* $\Leftrightarrow c_m$ è una configurazione accettante

DEF (linguaggio di una Macchina di Turing)

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid M(w) = 1\}$$

DEF (M che decide)

Una M decide il linguaggio $\mathcal{L} \Leftrightarrow$

$$\forall w \in \Sigma^* \begin{cases} \text{se } w \in \mathcal{L} \Rightarrow M(w) = 1 \\ \text{se } w \notin \mathcal{L} \Rightarrow M(w) = 0 \end{cases}$$

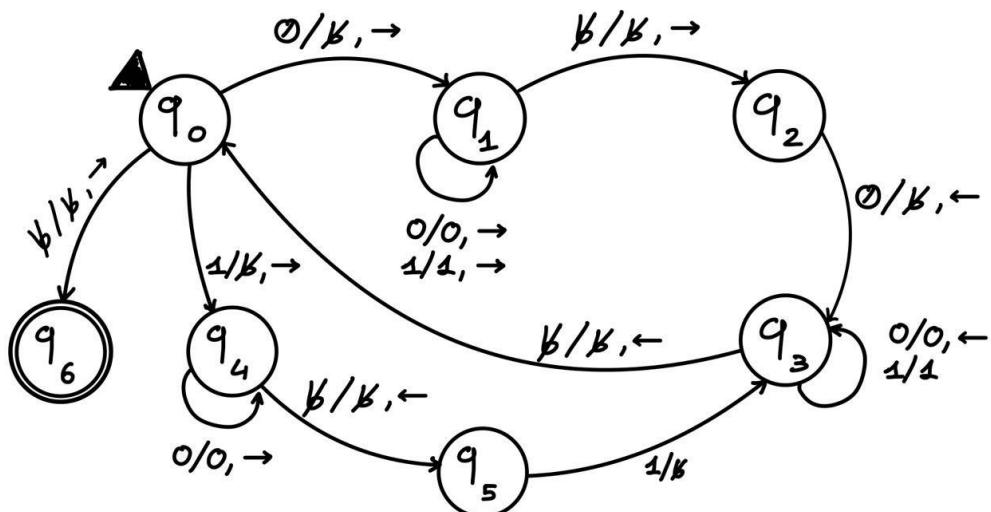
DEF (M che accetta)

Una M decide il linguaggio $\mathcal{L} \Leftrightarrow$

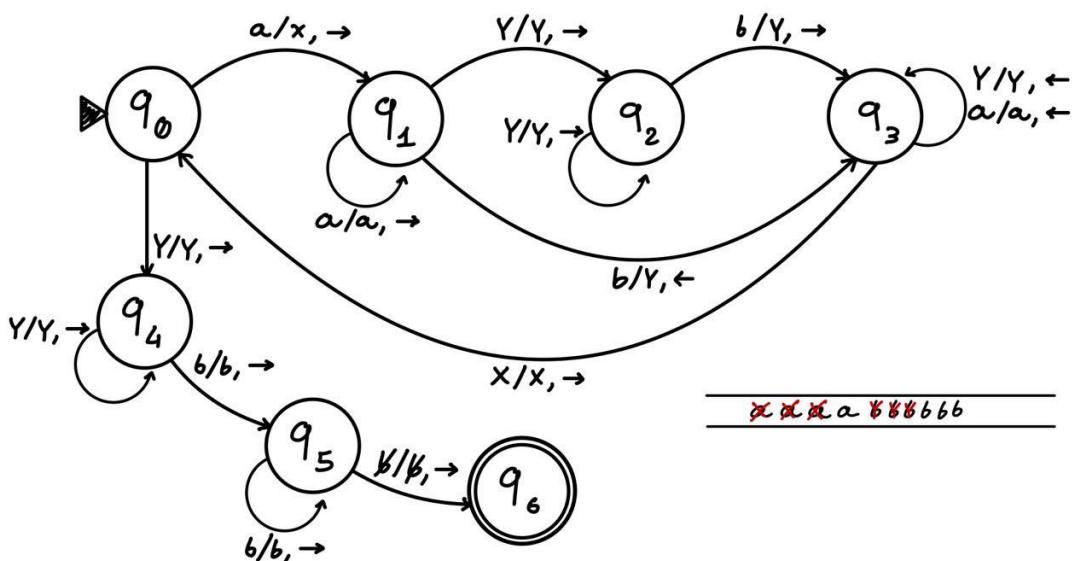
$$\forall w \in \Sigma^* \begin{cases} \text{se } w \in \mathcal{L} \Rightarrow M(w) = 1 \\ \text{se } w \notin \mathcal{L} \Rightarrow ? \end{cases}$$

2.2.1 Programmazione delle Macchine di Turing

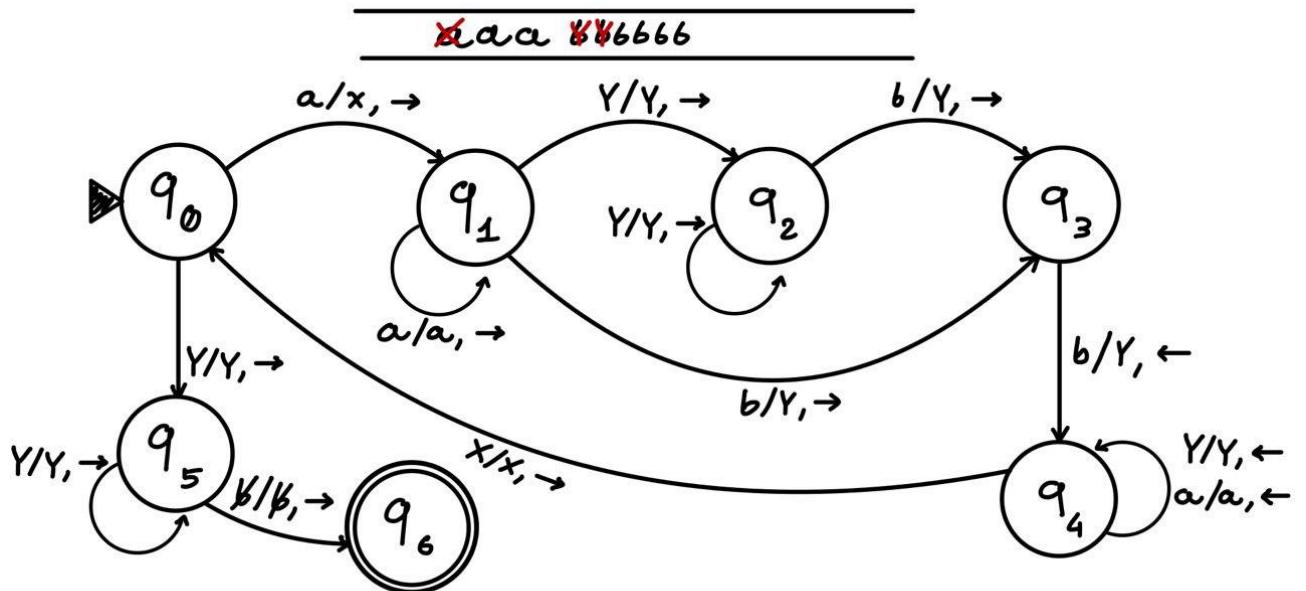
$$\mathcal{L} = \{ww^R \mid w \in (0,1)^*, w^R \text{ è } w \text{ al contrario}\}$$



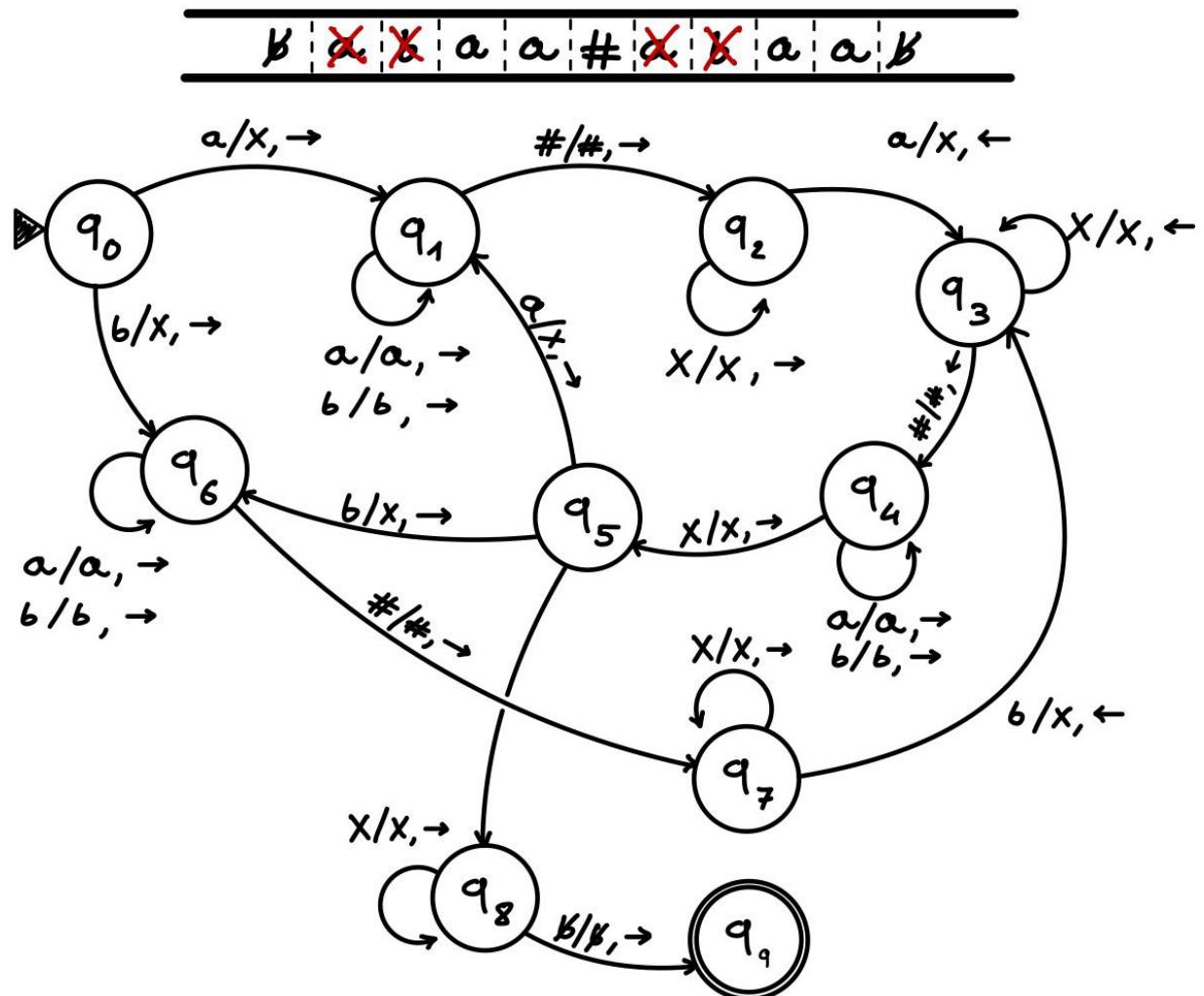
$$\mathcal{L} = \{a^n b^m \mid m > n > 0\}$$



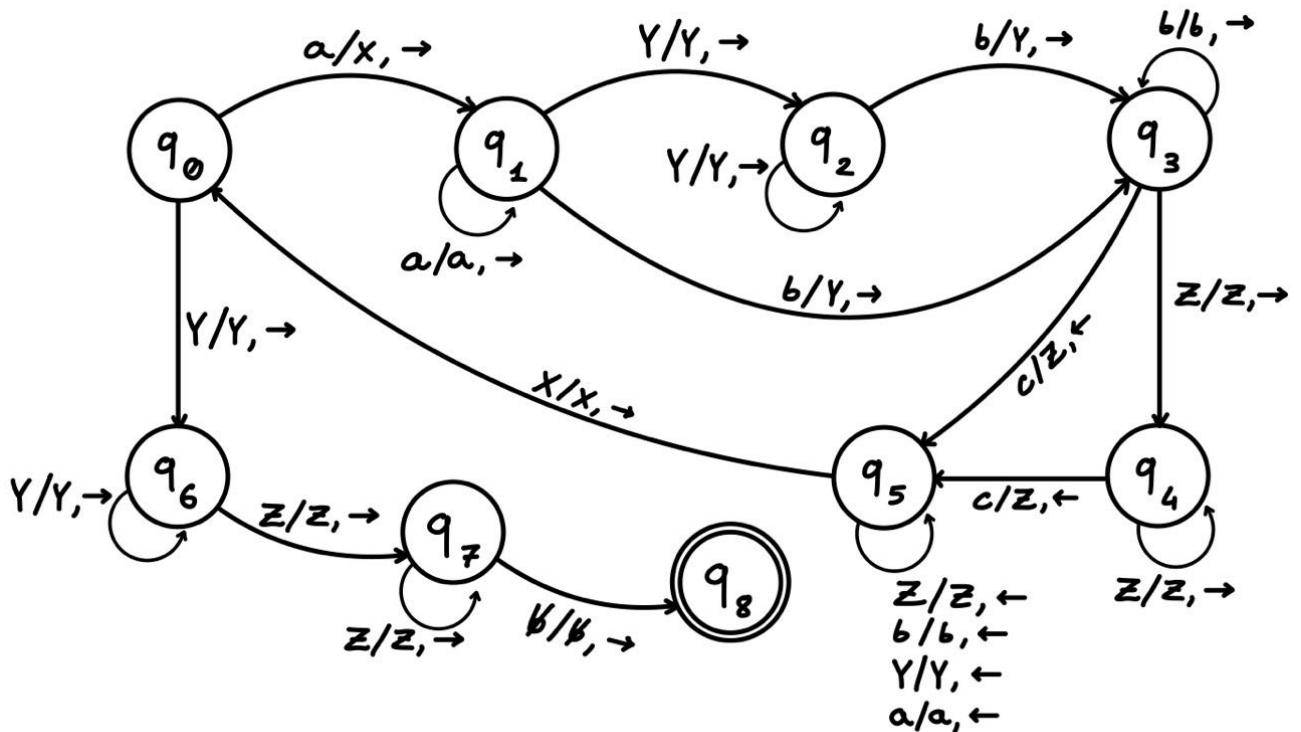
$$\mathcal{L} = \{a^n b^m \mid n > 0 \wedge m = 2n\}$$



$$\mathcal{L} = \{w\#w \mid w \in (a|b)^+\}$$



$$\mathcal{L} = \{a^n b^n c^n | n > 0\}$$



Optional che possiamo aggiungere alle macchine di Turing:

Riprendendo l'esempio della macchina che riconosce il linguaggio $\mathcal{L} = \{ww^R \mid w \in (0|1)^*\}$

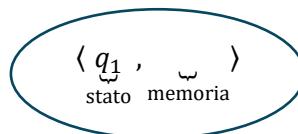
Uno schema molto veloce di come funzionava quella macchina è:



Abbiamo bisogno di due rami ($0|1$) distinti perché la macchina non è capace di ricordarsi quale simbolo ha letto in partenza.

Non sarebbe più bello se la macchina potesse memorizzare i simboli che ha letto?

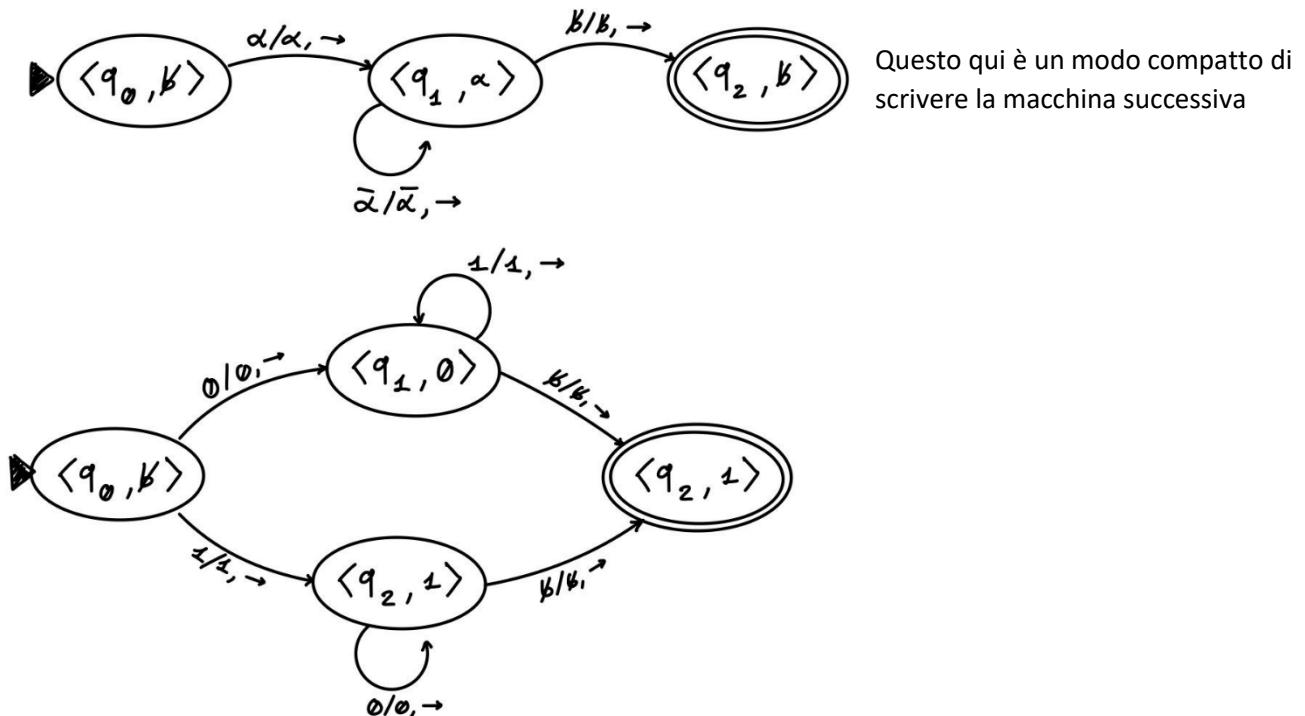
Introduciamo quindi la macchina con la memoria nello stato.



Come sfruttiamo questa cosa?

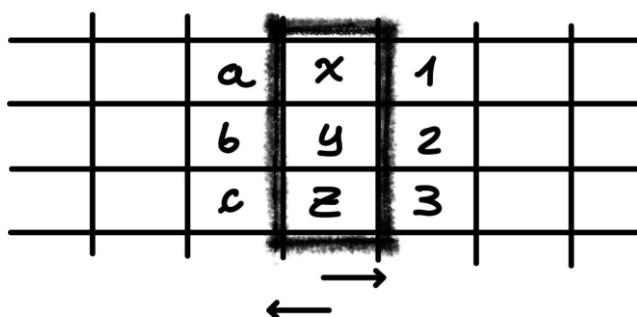
Supponiamo di voler riconoscere il seguente linguaggio indicato tramite espressione regolare

$$\mathcal{L} = 10^* \mid 01^*$$



2.2.2 Macchine multitraccia

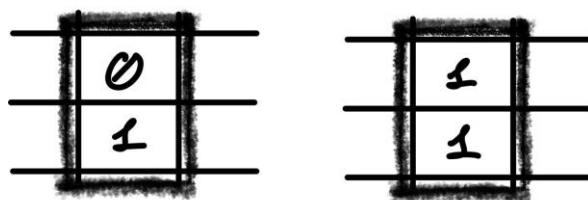
Sono macchine che hanno un nastro e **questo nastro ha più tracce**.



Peculiarità della testina è che è una, quindi se la spostiamo a destra o a sinistra, la sposteremo su tutte le tracce tutta assieme.

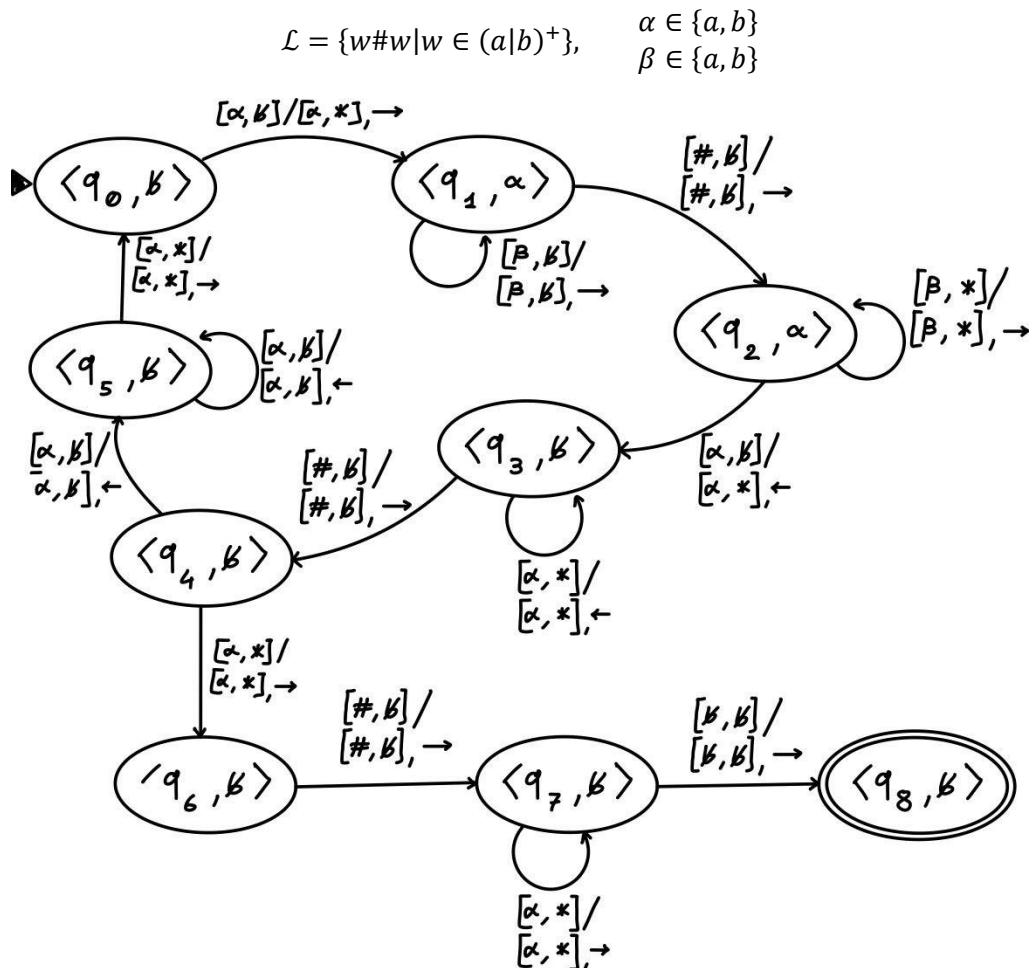
Come programmiamo la funzione di transizione di questa macchina?

La funzione di transizione è determinata da tutti i simboli che leggo su tutte le tracce.



Questi sono due casi distinti della macchina

Vediamo un esempio di utilizzo di una macchina del genere:



Possiamo dire che le macchine multitraccia hanno lo stato nella memoria? Si!

È vero che le macchine con uno stato della memoria hanno lo stesso potere delle macchine standard? Si!

Quindi possiamo concludere che le macchine multitraccia hanno lo stesso potere di calcolo delle macchine standard!

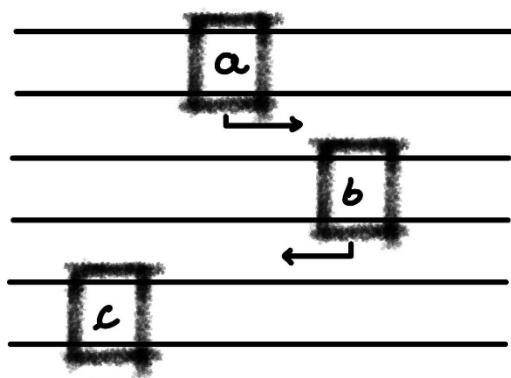
2.2.3 Macchine multinastro

Sono macchine di Turing che hanno più nastri completamente indipendenti.

Ogni nastro ha la propria testina indipendente.

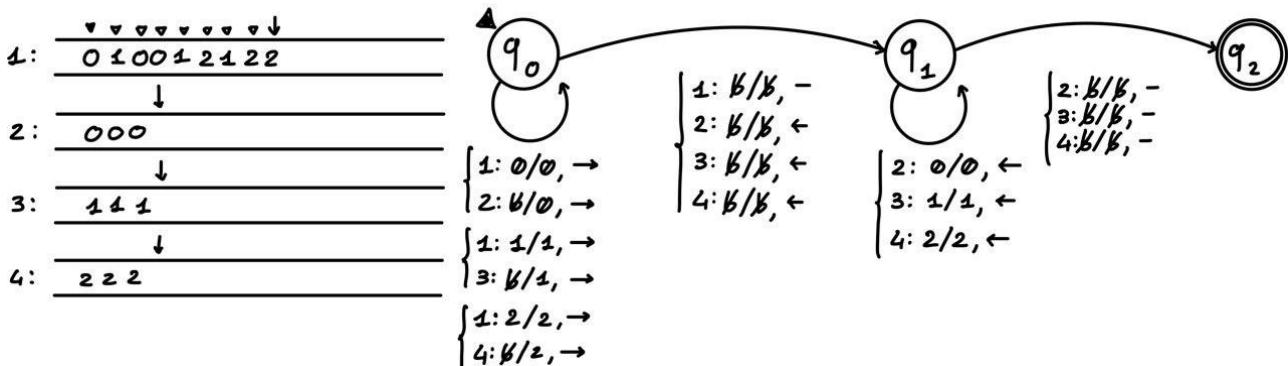
La cosa interessante è proprio questo spostamento delle testine a piacimento.

Il numero di nastri aggiuntivi viene deciso all'inizio della progettazione



Scriviamo una macchina che decida questo linguaggio:

$$\mathcal{L} = \{w \mid w \in (0|1|2) \wedge |0| = |1| = |2|\}$$



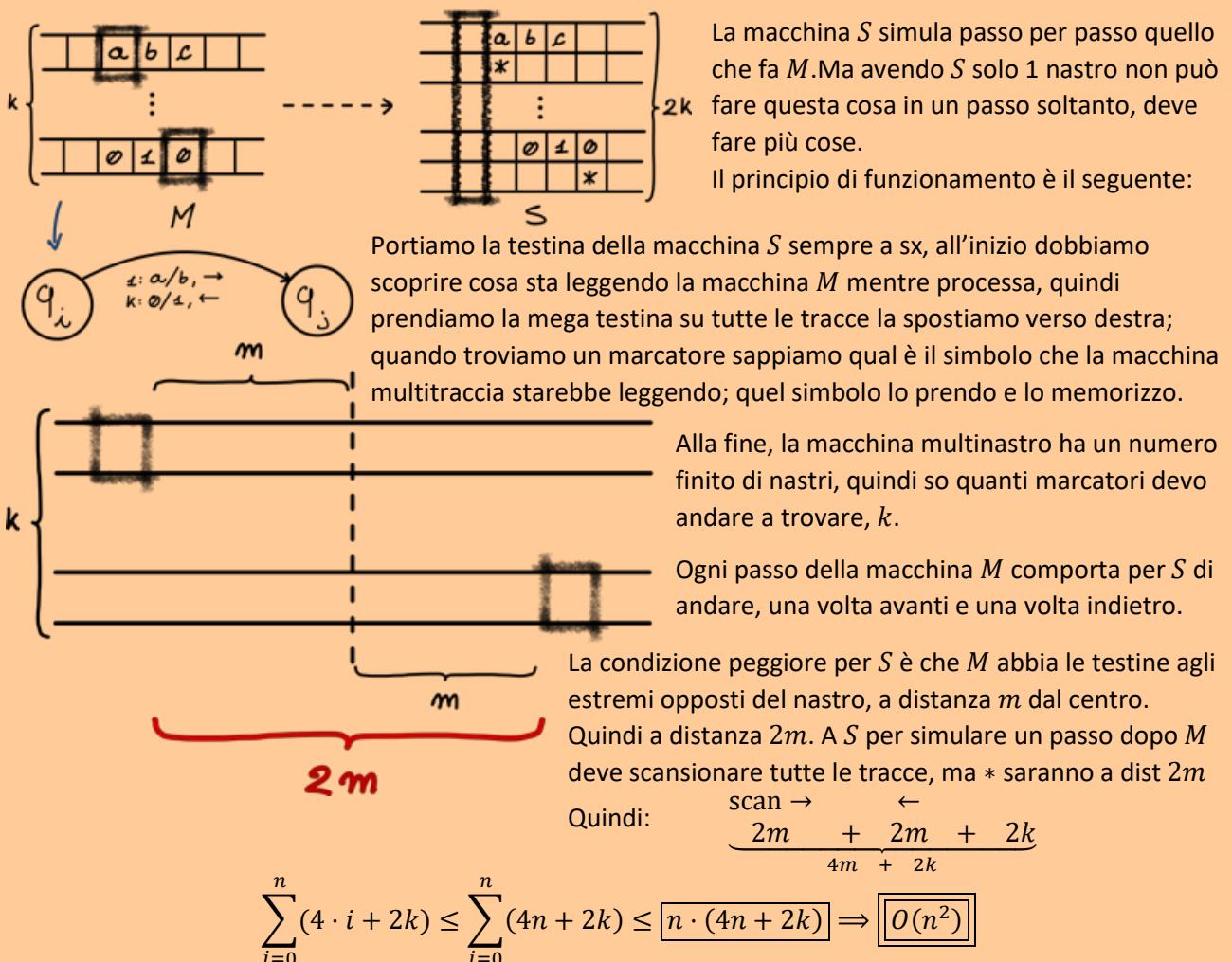
Ora vogliamo simulare il funzionamento di una macchina multinastro M in una macchina S a singolo nastro ma multi traccia

$$\begin{array}{ccc} M & \rightarrow & S \\ \text{multinastro} & & \text{Singolo nastro} \\ & & \text{multi traccia} \end{array}$$

THM (Equivalenza macchine multinastro-multitraccia)

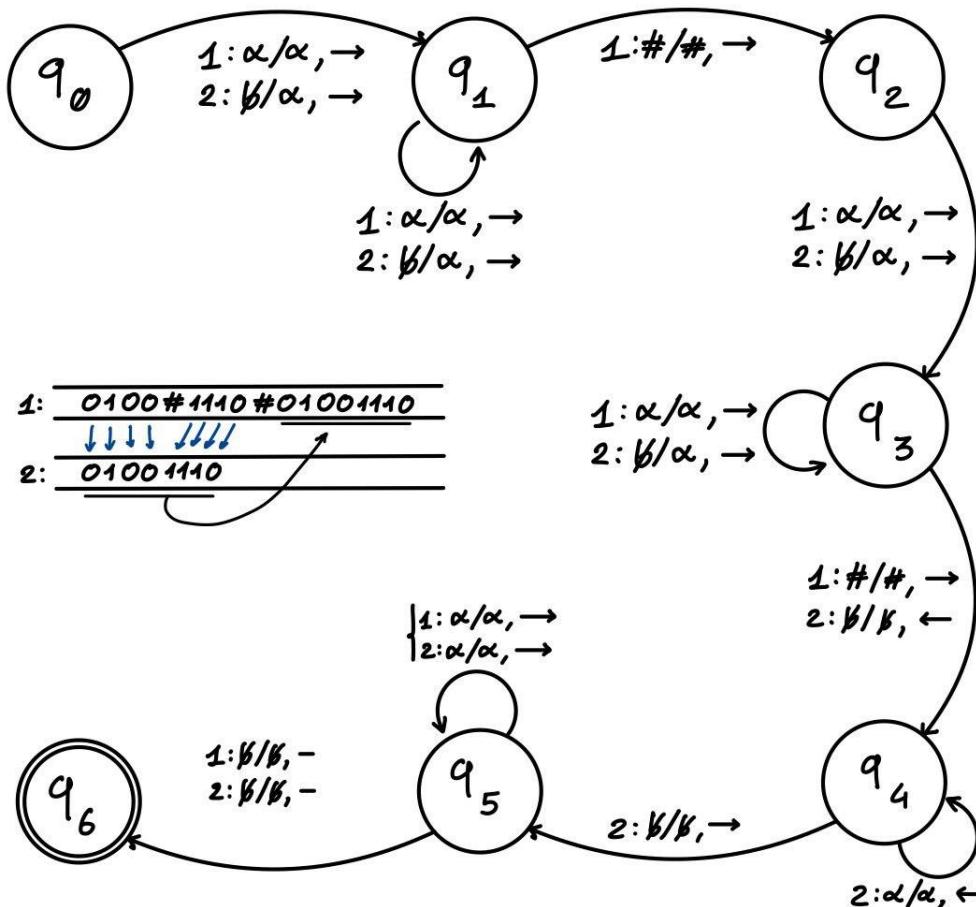
Sia M una macchina multi nastro, allora \exists una macchina S multi-traccia tale che il linguaggio riconosciuto da M è uguale al linguaggio riconosciuto da S , $\mathcal{L}(M) = \mathcal{L}(S)$

Proof: Supponiamo che la macchina M sia a k nastri, la simuliamo tramite S con $2k$ tracce

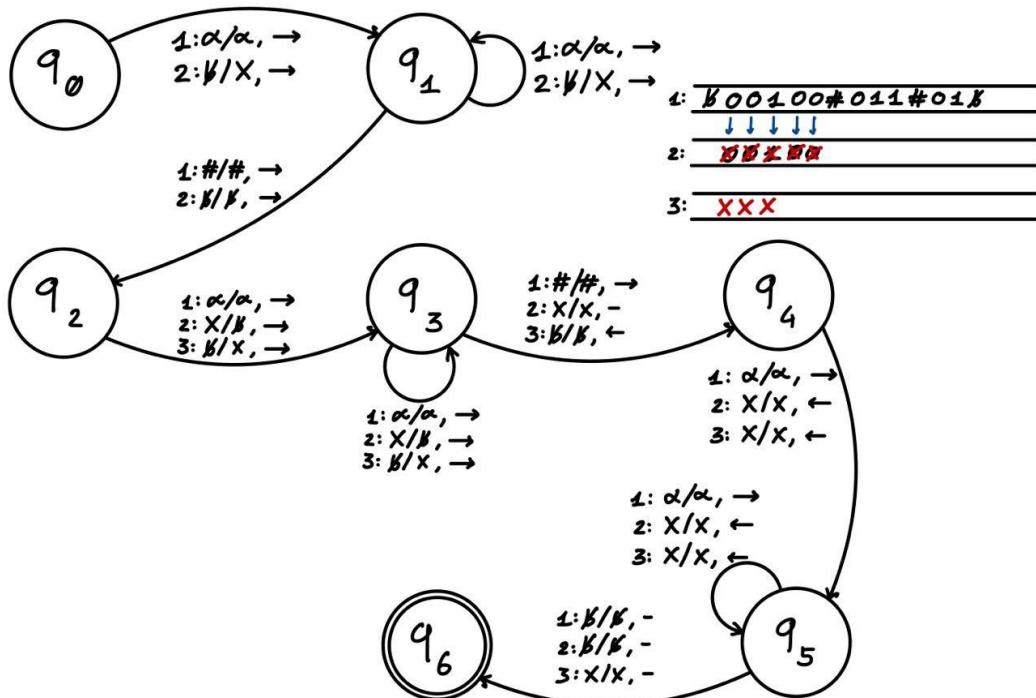


Vediamo ora alcuni esercizi su macchine di Turing multi nastro

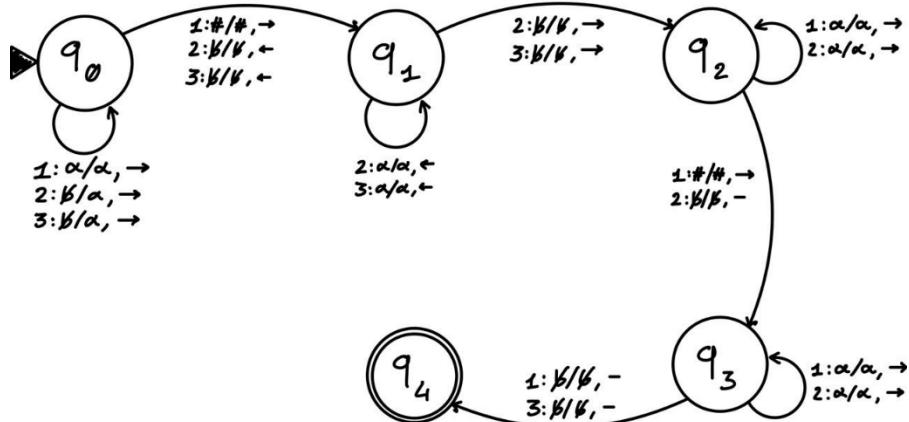
$$\mathcal{L} = \{A\#B\#AB \mid A, B \in (0|1)^+\}, \quad \alpha = \{0,1\}$$



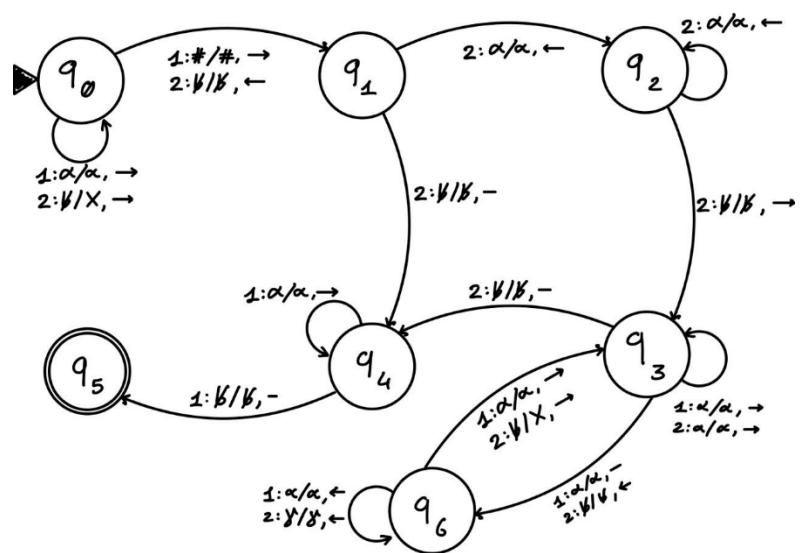
$$\mathcal{L} = \{A\#B\#C \mid A, B, C \in (0|1)^+ \wedge |A| > |B| > |C| \wedge |C| = |A| - |B|\}$$



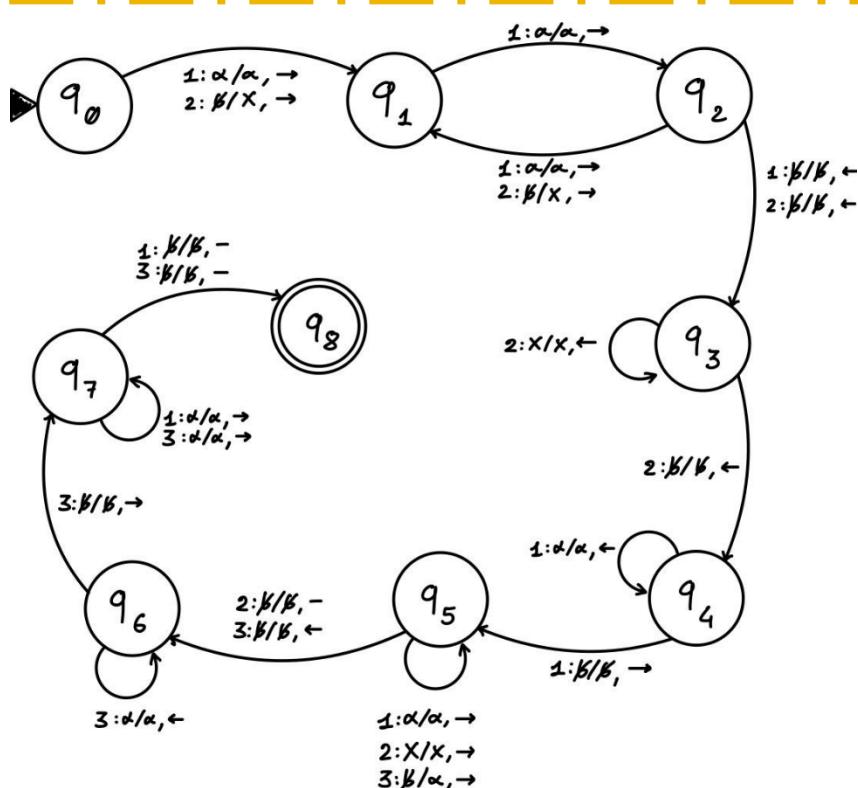
$$\mathcal{L} = \{W\#W\#W \mid W \in (0|1)^*\}$$



$$\mathcal{L} = \{A\#B \mid A, B \in (a|b)^* \wedge A \subseteq B\}$$

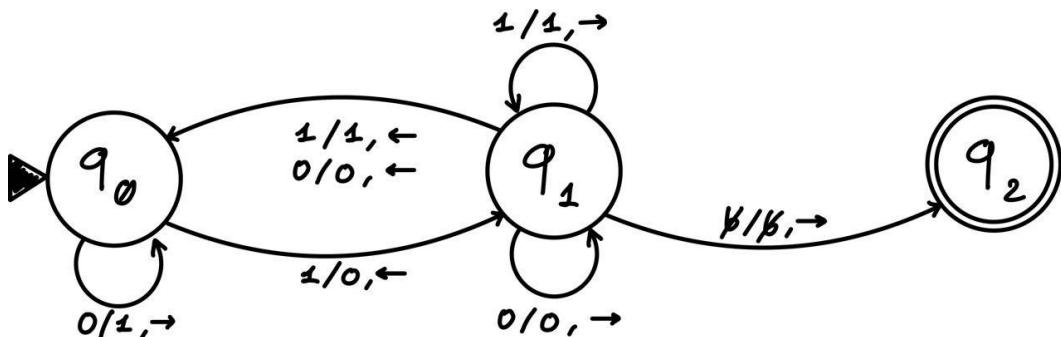


$$\mathcal{L} = \{WW \mid W \in (0|1)^+\}$$



2.3 MACCHINE DI TURING NON DETERMINISTICHE

Vediamo il modello che utilizzeremo per il resto del corso, lo useremo perché è pratico; assumiamo di avere questa macchina qua:



Questa macchina è *non deterministica* in q_1 , ovvero per una particolare coppia *simbolo letto-stato* abbiamo più di una scelta; ovvero può o rimanere in q_1 oppure transire in q_0 .

Il fatto che io da q_1 abbia $(0/0, \leftarrow)$ e $(0/0, \rightarrow)$ è un caso, la cosa importante è la coppia **stato-simbolo letto**.

DEF (Macchina di Turing non deterministica)

Una macchina di Turing non determinista è una tupla

$$N = \langle \Sigma, \Gamma, \delta, Q, q_0, F \rangle$$

1. Σ è l'alfabeto in input
2. Γ è l'alfabeto di nastro $\Rightarrow \Gamma \supset \Sigma$
3. $\delta \in \Gamma$
4. Q è l'insieme finito degli stati
5. q_0 è lo stato iniziale
6. $F \subseteq Q$ è l'insieme degli stati finali
7. δ è la funzione di transizione

Per le MdT Deterministiche era

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$$

Per una Mdt non deterministica

$$\delta: Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{\leftarrow, \rightarrow\}}$$

mappa le coppie stato-simbolo verso l'insieme delle parti delle triple stato-simbolo-direzione

$$\delta(q_1, 0) = \{(q_0, 0, \leftarrow), (q_1, 0, \rightarrow)\}$$

Cose da sottolineare:

? è necessario che δ di N sia non deterministica su tutte le possibili coppie ?

No

Vediamo come questa macchina computa su un particolare input che le diamo;

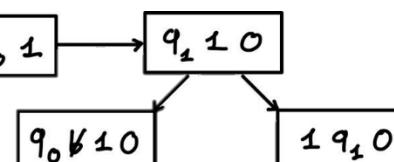
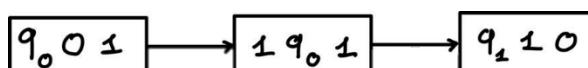
Con le Macchine deterministiche

utilizzavamo le configurazioni

(descrizioni istantanee) ed erano

delle stringhe *stato-la parte non blank del*

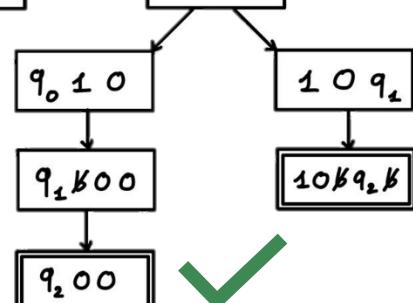
nastro-psz testina.



Questa macchina ha un modo per rifiutare e due modi per accettare.

Quindi se avessimo questa macchina e gli dessimo 01 come input, arriva alla conf $q_1 1 0$ e poi...?

Poi niente, questa macchina non esiste.



E perché dovremmo studiare una macchina che non esiste? Perché è una macchina più semplice. Quindi noi utilizziamo il modello della macchina di Turing non deterministica, sebbene non esista, perché è più semplice ragionarci sopra.

Noi possiamo fare questa assunzione di lavorare sulle macchine di Turing non deterministiche perché abbiamo dei modi per gestirle, per simularle.

Prima però formalizziamo il tutto:

Prop (Le configurazioni per le M non deterministiche sono equivalenti alle conf per quelle deterministiche)

DEF Una config iniziale per una macchina non deterministica è la stessa cosa per una deterministica

DEF (config finale per N)

In una macchina non deterministica una configurazione si dice finale se non ammette conf successive
è una configurazione accettante se lo stato è accettante
è una configurazione non accettante se lo stato non è accettante

L'unica effettiva differenza è che:

Una data configurazione può avere più legal successor.

DEF (computation tree)

Un computation tree per una Macchina non deterministica su una specifica stringa w in input è un albero, e come tale è composto da nodi, rami, uno di questi nodi è la radice e alcune foglie.

Deve catturare tutte le possibili computazioni che la macchina potrebbe fare sulla stringa in input.

Quindi i nodi sono tutte le possibili configurazioni in cui N può mai trovarsi mentre processa la stringa w.

La radice è la configurazione iniziale di N su w

C'è un arco da una configurazione α a una configurazione β se β è un legal successor di α

N accetta il proprio input w sse nel computation tree compare una configurazione accettante

Ma è vero o no che tutti i linguaggi accettati da macchine non deterministiche possono essere accettati da macchine deterministiche? A livello intuitivo la macchina deve fare backtracking

Dobbiamo quindi inventarci qualche trucco per riuscire a far accettare a una macchina deterministica lo stesso linguaggio di una macchina non deterministica.

Supponiamo di avere una macchina	N	e vogliamo trasformala in	M
non det	mono-nastro	det	multi-nastro

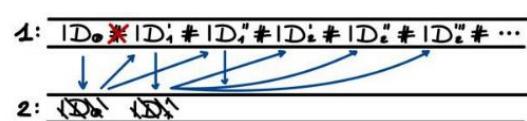
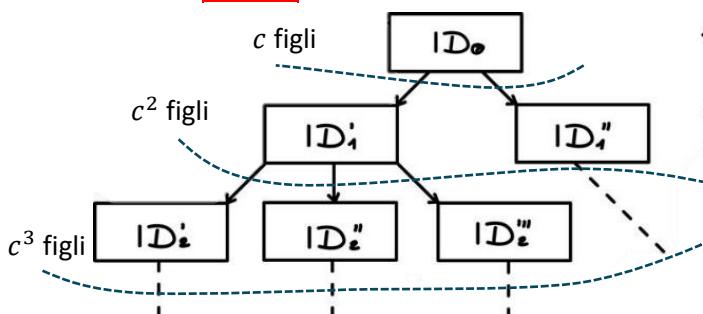
Supponiamo che la computazione di N su un certo input sia così:

Al primo livello dove M trova una configurazione accettante li si ferma e accetta

Se il computation tree di N non ha configurazioni accettanti, allora nemmeno la simulazione di M accetterà

⇒ il linguaggio accettato da N è lo stesso accettato da M

Quanto costa? $O(c^n)$



Quello che facciamo è utilizzare una MdT det. per esplorare l'albero di computazione. Stiamo esplorando l'albero *in ampiezza* (se la facessimo in profondità potremmo andare avanti all'infinito)

Quindi abbiamo che qualsiasi sia il linguaggio accettato da una macchina non deterministica può essere accettato da una macchina deterministica.

Quindi fino a quando siamo interessati a scoprire se un certo problema sia risolvibile o meno (e non ci interessa sapere se efficientemente o no)

2.3.1 Tesi di Church-Turing

Siccome tutto quello che è risolvibile decidibile accettabile da macchine non deterministiche è anche accettabile per macchine deterministiche, quello che posso fare è vedere se ci sta un algoritmo non deterministico, perché? Perché è più facile!

Quello che stiamo facendo è aggiungere *orpelli su orpelli* (mono-nastro → multi-traccia → multi-nastro → non deterministiche → …). Ma rimaniamo sempre in ciò che è calcolabile per la macchina di Turing standard (implementando metodi più veloci, sì, ma è sempre calcolabile da quella standard).

Tesi di Church-Turing

Tutto ciò che è umanamente calcolabile è calcolabile da una macchina di Turing Standard

Introduciamo quindi due classi di Calcolabilità,

DEF (Classe di calcolabilità)

Una classe di calcolabilità è un insieme di linguaggi

RE → Linguaggi Ricorsivamente enumerabili (o indicidibili)

(sempre per ragioni storiche)

Insieme di tutti i linguaggi o problemi di decisione

per i quali esiste una MdT che li accetti ⇒ {Si → tempo finito
No → no garanzie}

R → Linguaggi Ricorsivi (o decidibili)

(un tempo si riteneva calcolabile ciò che è esprimibile tramite funzioni ricorsive)

Contiene tutti quei linguaggi per i quali esiste

una macchina di Turing che li decide ⇒ {Si → tempo finito
No → tempo finito}

2.3.2 Riconoscimento di linguaggi tramite macchine non deterministiche

Riprendiamo un linguaggio già analizzato

$$\mathcal{L} = \{WW \mid W \in (0|1)^+\}$$

Come l'avevamo riconosciuto con la macchina deterministica?

> deve avere un numero paro di caratteri in input,

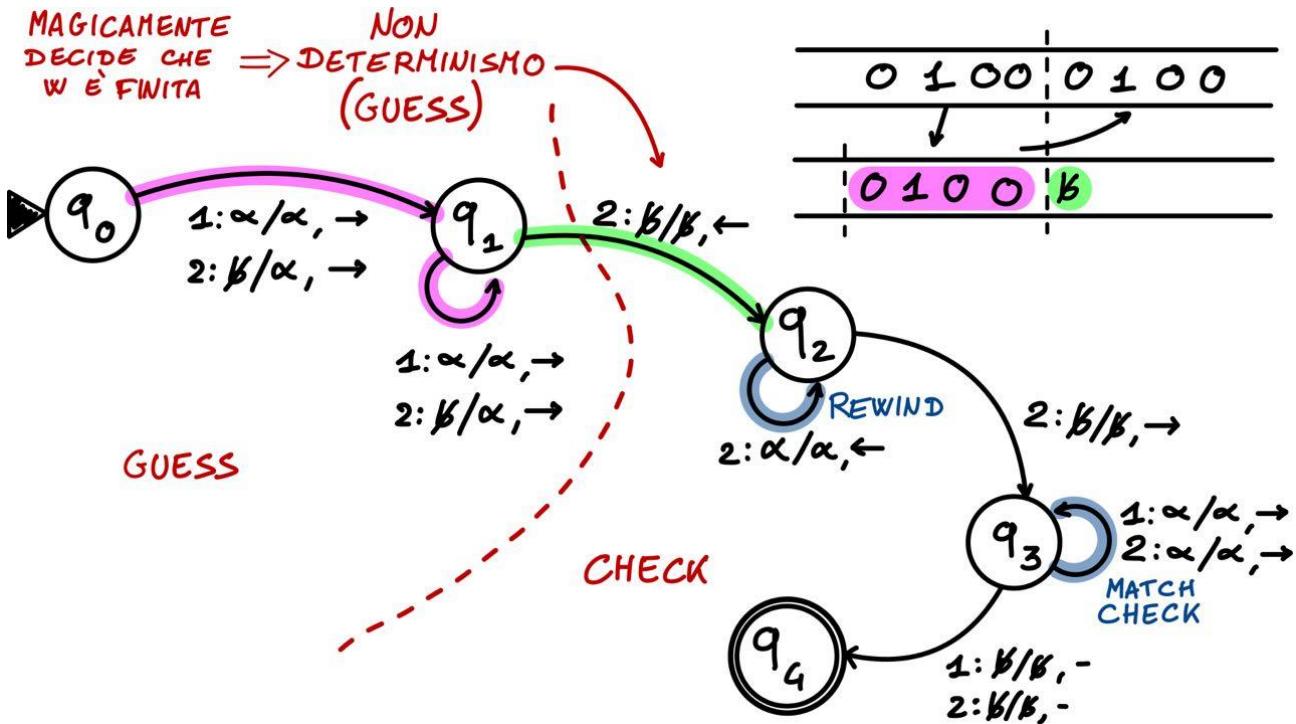
> li contiamo e andiamo a metà input

> controlliamo che le due metà corrispondano

Il design pattern principale delle macchine non deterministiche si chiama *guess & check*, cioè la macchina cerca di indovinare qualcosa e poi controlla che quello che ha indovinato l'ha indovinato bene.

$$\mathcal{L} = \{WW \mid W \in (0|1)^+\}$$

Una MdT non deterministica non ha necessità di conoscere per certo qual è la metà dell'input.

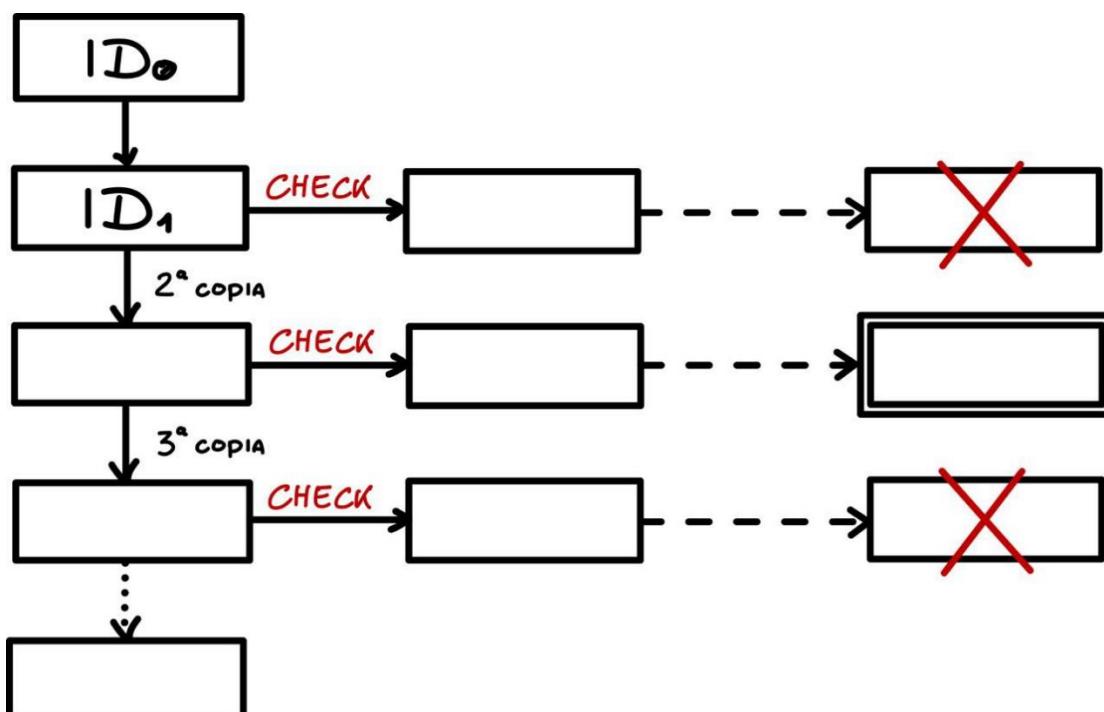


Abbiamo una definizione formale di accettazione di una MdTND

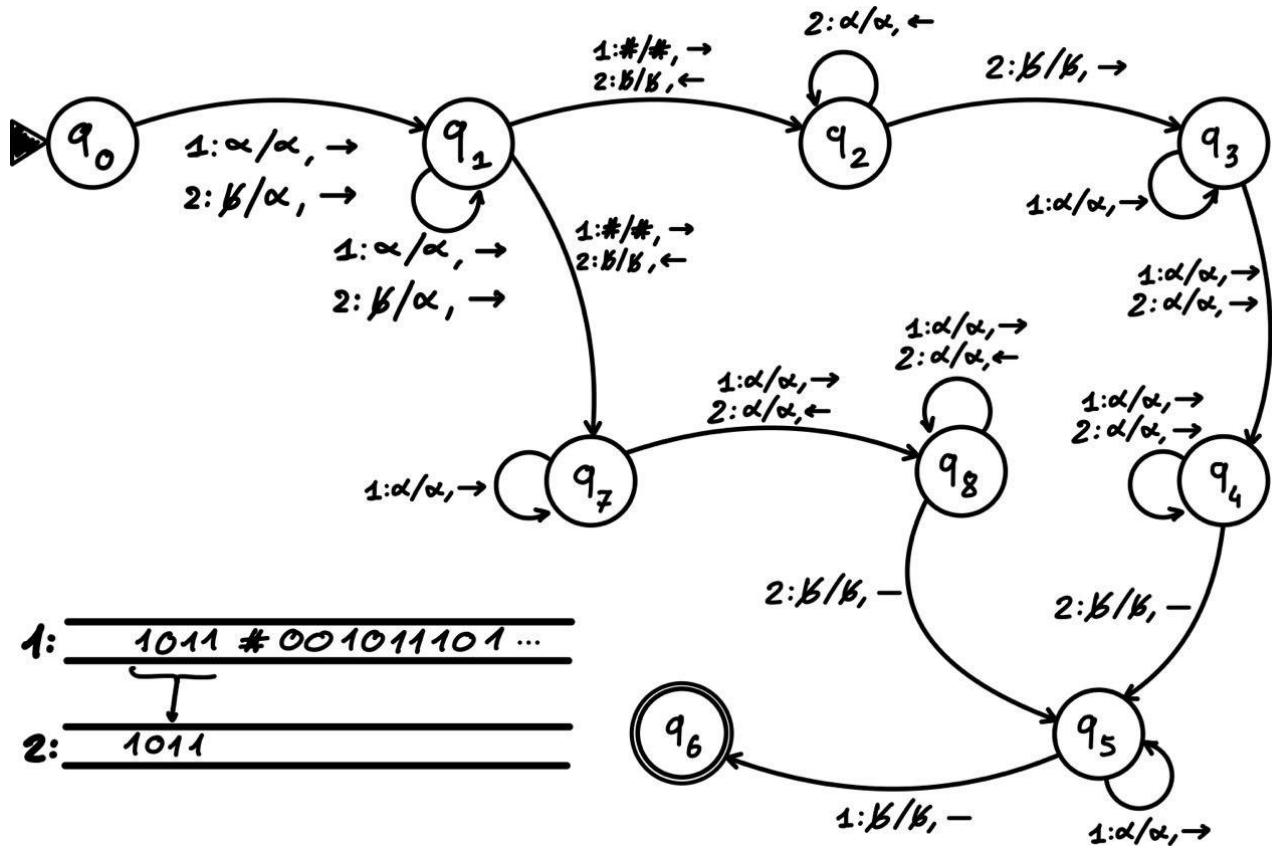
DEF (accettazione di MdTND)

Una MdTND *accetta* il proprio input se nel proprio computation tree c'è un computation branch accettante.

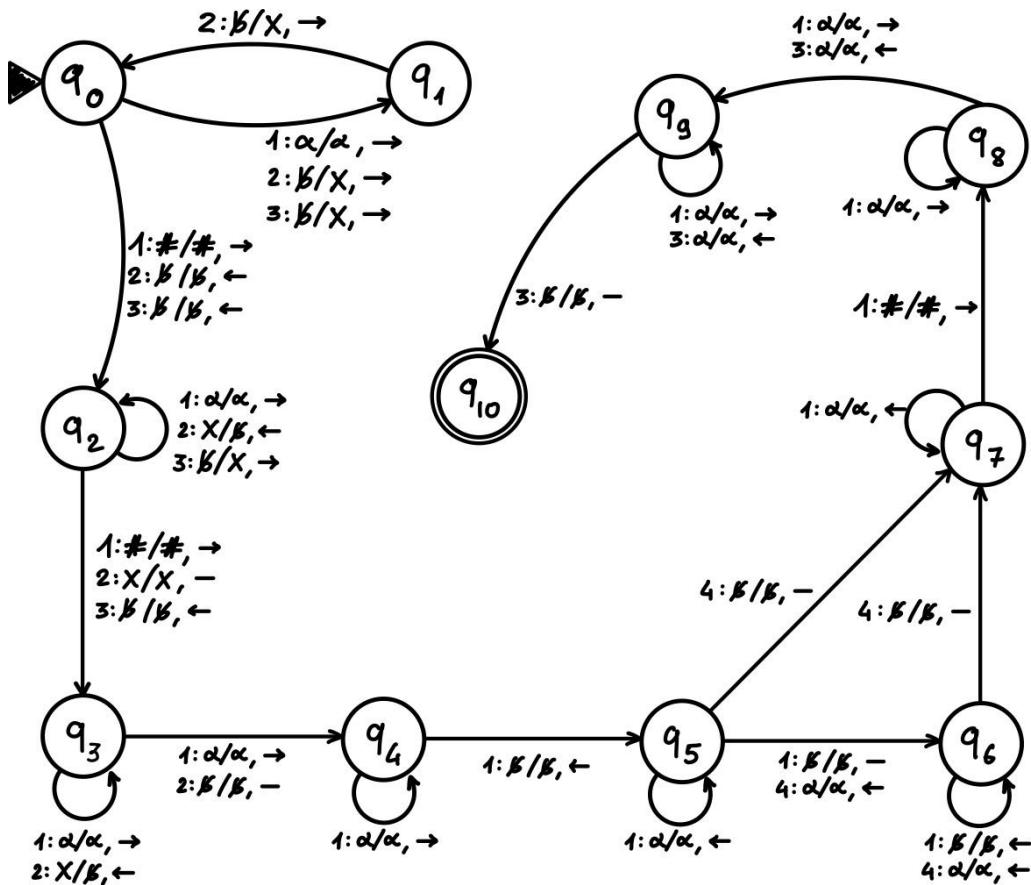
Computation Tree



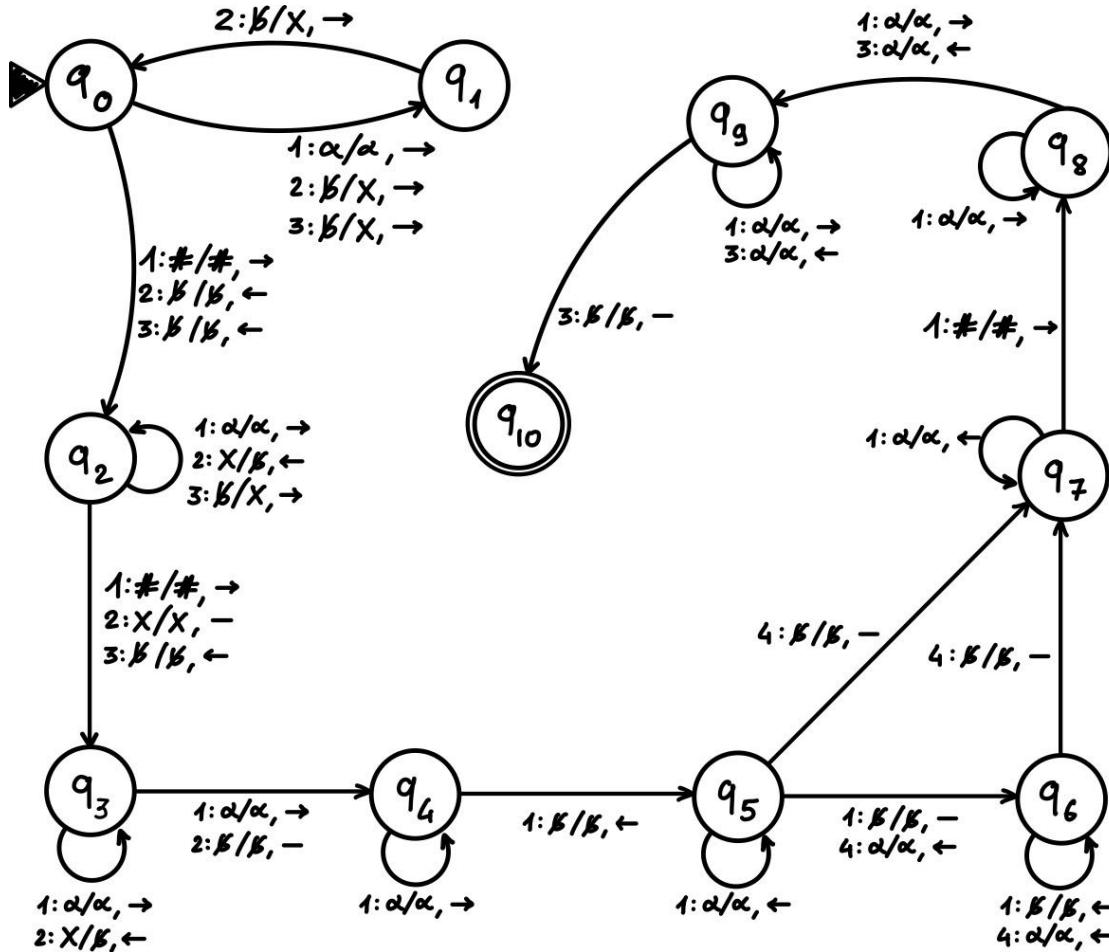
$$\mathcal{L} = \{A \# B \mid A, B \in (0|1)^+, A \subseteq B \vee A^R \subseteq B\}$$



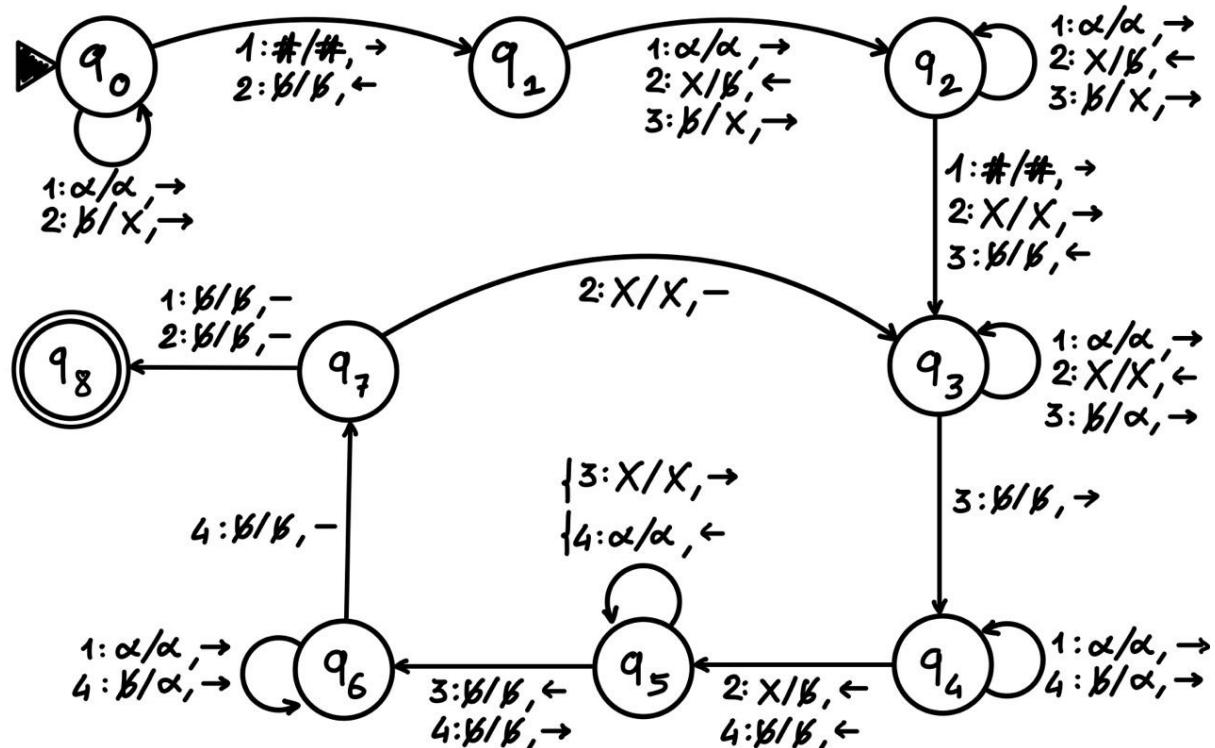
$$\mathcal{L} = \{A \# B \# W \mid W, A, B \in (0|1)^* \wedge |W| > 2|A| - |B| \wedge |B| < 2|A| \wedge (A^R \in W \vee B \subseteq W)\}$$



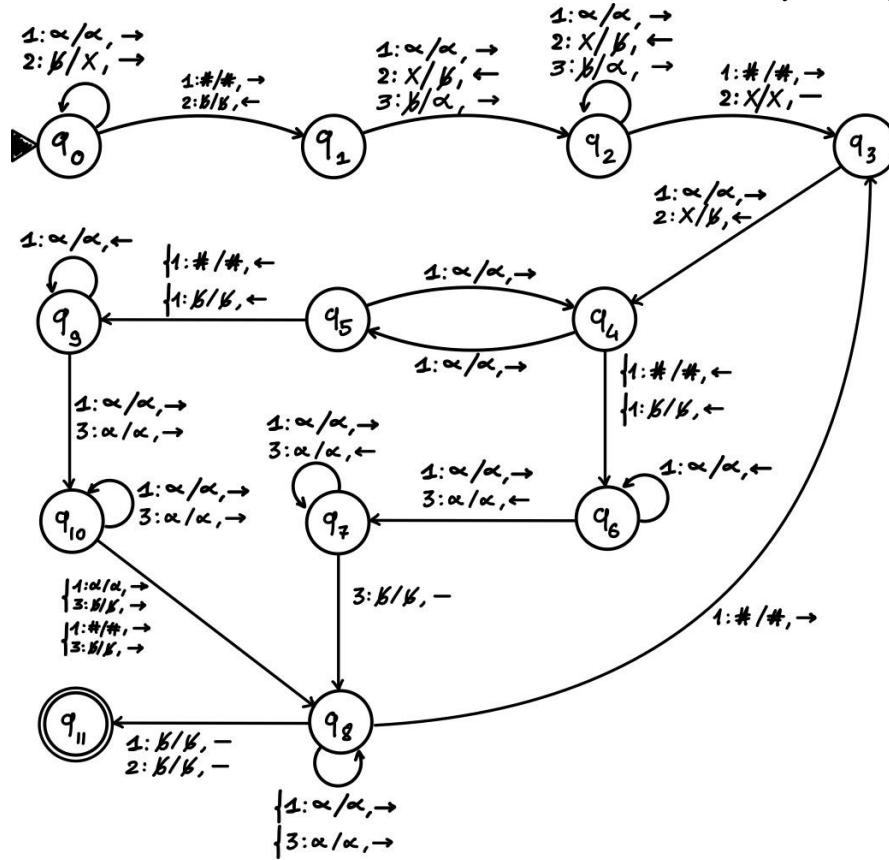
$$\mathcal{L} = \{X^n \# w_1 \# \dots \# w_n \mid n > 0, w_i \in (a|b|c|d)^+ \forall i \text{ t.c. } 1 \leq i \leq n, \exists s_i \subseteq w_i \wedge |s_i| = i, s_i = s_i^R\}$$



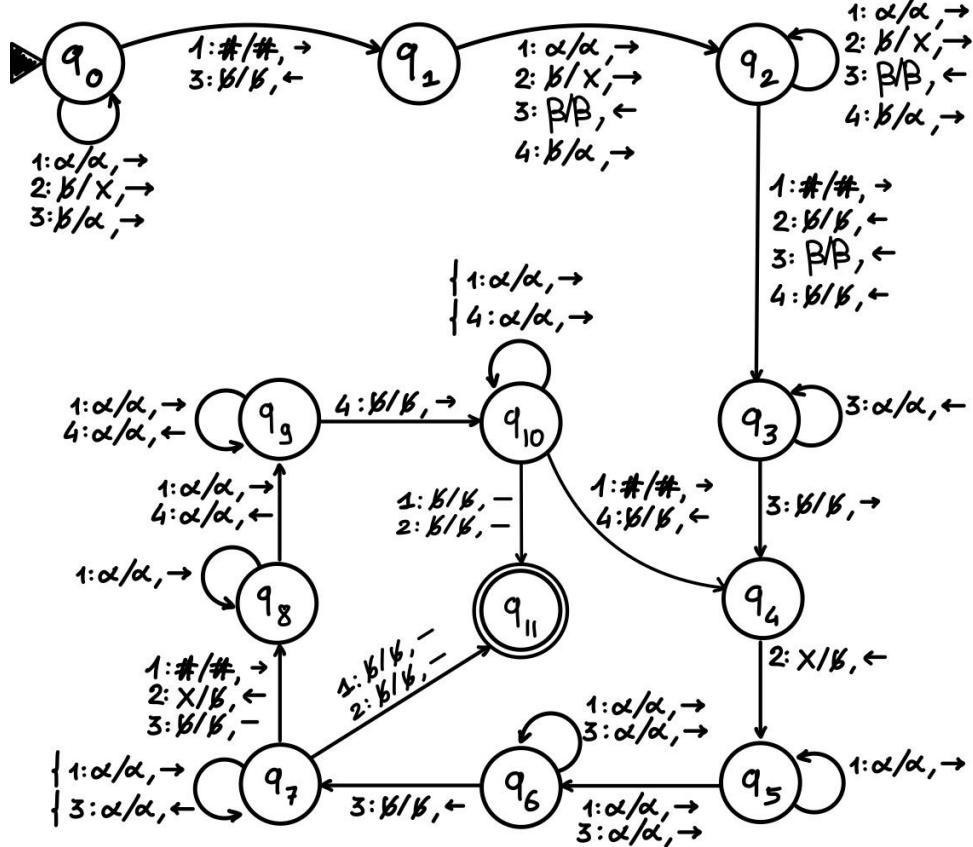
$$\mathcal{L} = \{A \# B \# W_1 W_1 W_2 W_2 \dots W_n W_n \mid A, B, W_i \in (0|1)^+ \wedge |A| > |B| \wedge n = |A| - |B| \wedge |W_i| \geq |B|\}$$



$$\mathcal{L} = \left\{ A \# B \# W_1 \# \dots \# W_n \mid A, B, W_i \in (a|b|c|d)^+ \wedge n = \frac{|A| + |B|}{|A| - |B|} \wedge \begin{cases} B \subseteq W_i & \text{se } |W_i| \text{ è pari} \\ B^R \subseteq W_i & \text{se } |W_i| \text{ è dispari} \end{cases} \right\}$$



$$\mathcal{L} = \left\{ A \# B \# W_1 \# \dots \# W_n \mid A, B, W_i \in (a|b|c)^+ \wedge n = \frac{|A| + |B|}{|A| - |B|} \wedge \begin{cases} A \subseteq W_i & \text{con } j \text{ dispari} \\ B^R \subseteq W_i & \text{con } i \text{ pari} \end{cases} \right\}$$



2.4 MACCHINE DI TURING UNIVERSALI

Ripensiamo alle macchine *non* deterministiche, noi abbiamo che possiamo farne una simulazione deterministica, nella quale il simulante (che è deterministico) sa fare altro che non quello che faccia il simulato? Cioè, noi partiamo da una macchina non deterministica N , la trasformiamo con una opportuna procedura in una macchina deterministica M ; la domanda è: la macchina simulante, cioè quello che noi creiamo che è deterministico, è in grado di riconoscere cose diverse rispetto al simulato?

M è una macchina generica o fa solo quello che fa N ?

Risp: Fa solo quello che fa N .

Il simulante è una macchina la cui funzione di transizione δ viene definita apposta, sa simulare il simulato. Quindi partendo da N , la macchina deterministica simulante è progettata apposta, è *sartoriale*.

Però questa non è la nostra esperienza diretta di computer, per noi ad un computer gli diamo un programma e quello esegue il programma che gli inviamo. Un computer è qualcosa di programmabile, cioè qualcosa la cui funzione di transizione pare prendersela dal programma.

Però abbiamo detto che le MdT sono potenti quanto i computer nostri...

Cioè, i nostri computer, a differenza delle MdT pare possano ricevere in input un programma e adattare la loro funzione di transizione in base al programma ricevuto

Mostreremo che i nostri computer sono delle MdT particolari

Un computer è quello che chiamiamo una **Macchina di Turing Universale**.

Però come facciamo a insegnare a una MdT a comportarsi come un'altra MdT? Ci dobbiamo inventare un codice per i programmi, come abbiamo i codici Java, C++, Python... ci dobbiamo inventare un codice che ci permetta di insegnare a questa macchina universale a fare le cose che fanno le altre MdT

Per le macchine che abbiamo visto finora, il loro *programma* è la funzione di transizione (che è fissato) Ci dobbiamo quindi inventare un modo per codificare le funzioni di transizione in modo tale che queste possano essere usate come input di un'altra macchina e questa macchina universale prende il codice della funzione di transizione di altre macchine e si comporta come quella. Quindi è una macchina che avrà il potere di comportarsi come tutte le altre macchine!

La macchina che consideriamo è una macchina il cui alfabeto di input ha solo due simboli $\Sigma = \{0,1\}$ e invece, l'alfabeto di nastro può essere $\Gamma = \{0,1,\#, \dots\}$

La funzione di transizione ha, generalmente, questa forma:

$$\delta(Q_i, X_j) = (Q_k, X_l, D_m)$$

ie; $\delta(Q_3, \#) = (q_5, 0, \rightarrow)$

Noi codifichiamo i simboli Q_i, X_j, Q_k, X_l, D_m partendo dai loro *pedici*,

$$Q_i = 0^i, \quad X_j = 0^j, \quad sx = D_1, \quad dx = D_2$$

ie.

$$x_1 = 0, \quad x_2 = 1, \quad x_3 = \#, \quad x_4 \dots$$

q_1 = è sempre lo stato iniziale
 q_2 = è sempre il singolo stato accettante

$$c_1 = \boxed{\underbrace{000}_q \underbrace{1}_X \underbrace{000}_q \underbrace{1}_X \underbrace{00000}_q \underbrace{1}_X \underbrace{0}_D \underbrace{1}_D \underbrace{00}_D}$$

c_1 è un'istanza della funzione di transizione, per unirle usiamo una sequenza di c_i separate da 11

$$c_1 11 c_2 11 c_3 11 c_4 \dots$$

↑

Questo modo ci permette di scrivere in una mega stringa binaria cosa fa una MdT. Ci sono infiniti modi ma noi scegliamo questo. Il nostro obiettivo è poter comunicare a una MdT come comportarsi rispetto a un'altra.

Quindi le stringhe binarie possono essere messe in associazione con possibili MdT.

Ci saranno stringhe binarie che non codificano MdT, allora assumiamo che le stringhe binarie che non seguono quella codifica sono macchine con un un solo stato non accettante.

Notiamo che le stringhe iniziano con 0, noi possiamo infilare un 1 di fronte alla codifica di una TM avremo un numero intero. Quindi possiamo contare le MdT.

$$1 \cdot \varepsilon, \quad 1 \cdot 0, \quad 1 \cdot 1$$

DEF (Macchina di Turing Universale)

La Macchina di Turing Universale M_u è una macchina che riceve in input una stringa binaria che codifica una macchina M e poi una stringa w

$$(M, w)$$

La macchina M_u è in grado di dirci se M_i avrebbe accettato o meno w .

La macchina universale *simula* M su w , se
 M accetta $w \Rightarrow M_u$ dirà "si"
 M non accetta $w \Rightarrow M_u$ dirà "no"

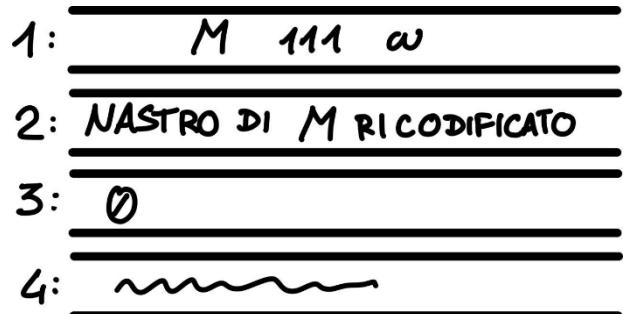
Quindi M_u è un **decisore universale**, perché è un decisore capace di comportarsi come altre macchine che gli vengono descritte in input tramite una stringa M che codifica una funzione di transizione.

Come funziona questa macchina? È una macchina a quattro nastri:

Ricordiamo che, per come abbiamo strutturato la codifica di M , dentro ad esso non compaiono mai tre 1 di fila; quindi, usiamo 111 per separare M e w . In questo modo abbiamo doppio input sul nastro di input.

M_u si comporta passo passo come si comporta M .

Quindi M_u come primo passaggio prende w e lo ricodifica secondo i simboli che ci siamo inventati



$$w = \begin{matrix} 0 & 1 & 0 \\ \overbrace{X_1 X_2 X_3}^{\text{la sua ricodifica}} \end{matrix} \xrightarrow{\text{2° nastro è}} 1010010$$

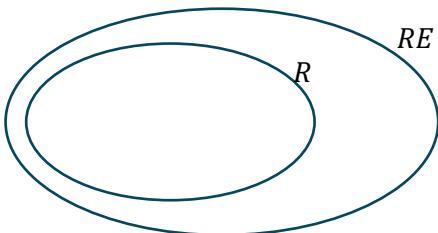
Dopodiché M_u si segna sul terzo nastro lo stato iniziale e posiziona sul secondo nastro la testina sul simbolo visto da M e posiziona sul terzo nastro la testina sullo stato corrente di M simulato.

M_u guardando i suoi nastri 2 e 3 sa, in quella fase di simulazione, in quale stato è M e quale simbolo sta leggendo. Una volta fatto questo M_u può aggiornare lo stato e può andare a ritoccare il nastro simulato di M , andando a sostituire il simbolo. I simboli che abbiamo sul secondo nastro sono simboli simulati, hanno forma 1010010, allora potrebbe servirci un po' di spazio o potremmo dover comprimere un po' di cose; ecco a cosa serve il quarto nastro, per lavori di copia-incolla, per estendere o accorciare il secondo nastro.

- ❓ I nostri computer sono MdTU, quindi avranno una funzione di transizione universale... fissata qual è?
È il microcodice della CPU.

E Questo risponde alla domanda "ma come mai le MdT non si programmano e i computer sì?".

2.5 PROBLEMI RICORSIVI E RICORSIVAMENTE ENUMERABILI



Abbiamo introdotto le classi di calcolabilità R e RE , e abbiamo detto che i problemi possono stare in R e in RE .

Ma possono starci dei problemi qua?

Sì, e ora li andremo a vedere; andiamo a vedere i problemi non ricorsivamente enumerabili

Utilizzeremo ora la codifica di cui prima:

Riconsideriamo macchine che processano in input stringhe binarie; se prendiamo una stringa binaria

00010 ...

Se mettiamo un 1 davanti alla stringa binaria possiamo associarle un numero intero

1 ° 00010 ...

E otteniamo il numero dato dalla codifica, in questo modo possiamo enumerare le stringhe

$\underbrace{1 \circ \varepsilon}_{1^\circ} \quad \underbrace{1 \circ 0}_{2^\circ} \quad \underbrace{1 \circ 1}_{3^\circ} \quad \underbrace{1 \circ 01}_{4^\circ} \quad \underbrace{1 \circ 10}_{5^\circ} \quad \dots$

E le stringhe binarie con indice più basso sono tendenzialmente più corte, stringhe binarie della stessa lunghezza sono ordinate lessicograficamente. Quante sono le stringhe binarie?

Infinite numerabili.

Lo stesso tipo di discorso può essere fatto per le MdT, avendo inventato la codifica di cui prima, per la quale ad ogni MdT può essere associata una codifica binaria, se ci mettiamo un 1 davanti otteniamo un numero; Quindi possiamo contare le Macchine di Turing! Ed esisteranno MdT che riconosceranno lo stesso linguaggio, perché magari hanno funzione di transizione differente, però alla fine fanno la stessa cosa.

Quindi, esattamente come possiamo contare le stringhe binarie possiamo contare le MdT!

DEF (Vettore caratteristico per un linguaggio \mathcal{L})

Il vettore caratteristico del linguaggio \mathcal{L} è una stringa binaria

ie. $\chi_{\mathcal{L}} = \{0 \ 1 \ 1 \ 0 \ 0 \ 1 \ \dots\}$

ci dice in posizione i se la i -esima stringa fa parte del linguaggio o meno

Di conseguenza, siccome un linguaggio, per definizione, è un sottoinsieme delle stringhe costruibili su di un certo alfabeto (in questo caso l'alfabeto binario). Allora il vettore caratteristico di un certo linguaggio sostanzialmente ci dice quali stringhe fanno parte di quel linguaggio.

	w_1	w_2	w_3	w_4	...
M_1	0	1	0	0	...
M_2	1	1	0	1	...
M_3	0	0	0	1	...
M_4	1	0	0	1	...
:	:	:	:	:	⋮

Ora creiamo questa tabella:

Sulle righe di questa tabella compare il *vettore caratteristico* del linguaggio accettato da quella macchina di Touring.

Concentriamoci sulla diagonale di questa matrice:

$$D = \{0 \ 1 \ 0 \ 1 \ \dots\} \quad \bar{D} = \{1 \ 0 \ 1 \ 0 \ \dots\}$$

Possiamo interpretare \bar{D} come vettore caratteristico di un certo linguaggio? Si!

Quindi chiamiamo \mathcal{L}_d il linguaggio il cui vettore caratteristico è \bar{D} .

Esiste una MdT il cui linguaggio è \mathcal{L}_d ?

Se esiste deve apparire in una certa riga della tabella, può essere M_1 ? No, perché in (1,1) è diverso, può essere la macchina M_i ? No, perché in i -esima posizione il vettore della macchina M_i è diverso da \bar{D} in (i, i) .

$\Rightarrow \nexists$ una MdT che accetti $\mathcal{L}_d \Rightarrow \mathcal{L}_d \notin RE$

$$\mathcal{L}_d = \{w_i \mid M_i \# w_i\}$$

Perché esistono i linguaggi indecidibili?

Le MdT sono enumerabili, quindi sono tante quante \mathbb{N} , i linguaggi sono sottoinsiemi di quelle stringhe ($w_1, w_2, w_3 \dots$) Quanti sono tutti i possibili sottoinsiemi di un numero infinito di stringhe? $|\mathbb{R}|$

⇒ Esistono più linguaggi che algoritmi, per questo esistono, perché sono tanti.

2.5.1 Proprietà sui linguaggi

DEF (complemento di un linguaggio)

Sia $\mathcal{L} \subseteq \Sigma^*$, definiamo $\overline{\mathcal{L}}$ come il complemento del linguaggio \mathcal{L} (è il contrario di \mathcal{L} , ciò che sta in \mathcal{L} non sta in $\overline{\mathcal{L}}$ e viceversa)

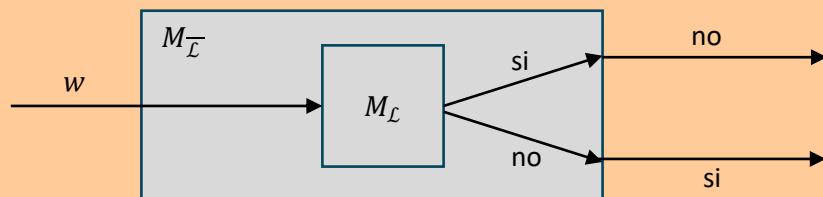
$$\Rightarrow \overline{\mathcal{L}} \subseteq \Sigma^* \setminus \mathcal{L}$$

THM Se $\mathcal{L} \in R \Rightarrow \overline{\mathcal{L}} \in R$

Proof

Siccome $\mathcal{L} \in R \Rightarrow \exists M_{\mathcal{L}}$ in grado di dirci si o no per definizione.

Allora possiamo costruire una macchina $M_{\overline{\mathcal{L}}}$ di questo tipo

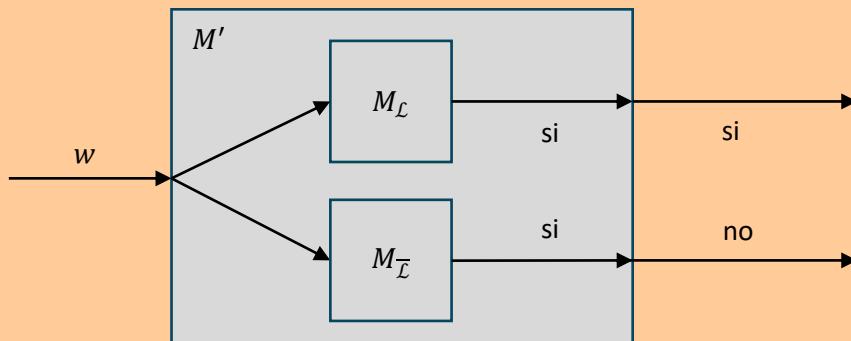


Dato che esiste una macchina che decide una macchina che decide $\overline{\mathcal{L}} \Rightarrow \overline{\mathcal{L}} \in R$

THM Se $\mathcal{L} \in RE \wedge \overline{\mathcal{L}} \in RE \Rightarrow \mathcal{L} \in R$

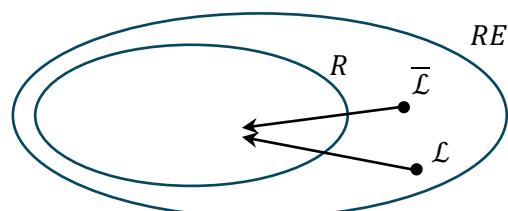
Proof

Dovendo dimostrare che $\mathcal{L} \in R$ ci basta mostrare che esiste una macchina in grado di decidere il linguaggio



Se un linguaggio e il suo complemento sono ricorsivamente enumerabili allora in realtà sono entrambi ricorsivi, perché io posso combinare le macchine e prendere la prima risposta che arriva.

Se ho un linguaggio e il suo complemento in RE
allora in realtà entrambi stanno dentro a R .



3 LINGUAGGIO UNIVERSALE E PROBLEMA DELLA FERMATA

DEF (Linguaggio universale)

È l'insieme delle coppie macchina-stringa, tali che la macchina M accetta w

$$\mathcal{L}_u = \{(M, w) | M \models w\}$$

Domanda: $\mathcal{L}_u \in RE$? (un linguaggio sta in RE se esiste una MdT in grado di accettarlo)
 \Rightarrow Si, perché esiste M_u

Ora ci chiediamo $\mathcal{L}_u \stackrel{\in}{\notin} R$? (un linguaggio sta in R se esiste una MdT in grado di deciderlo)

\Rightarrow

THM $\mathcal{L}_u \notin R$

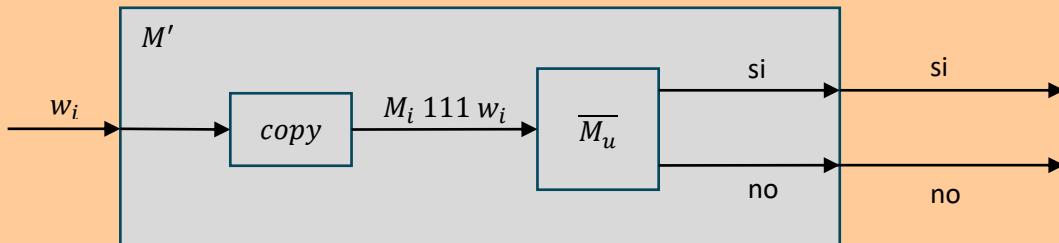
Proof

Supponiamo per assurdo che $\mathcal{L}_u \in R$

Se $\mathcal{L}_u \in R \Rightarrow \overline{\mathcal{L}_u} \in R$ e se $\overline{\mathcal{L}_u} \in R$ allora esiste una macchina, chiamiamola $\overline{M_u}$, che decide $\overline{\mathcal{L}_u}$

Allora, dato che $\overline{M_u}$ decide $\overline{\mathcal{L}_u}$, $\overline{M_u}$ da sempre garanzia di arresto per risposta corretta, è sempre in grado di dire sì ed è sempre in grado di dire no.

Consideriamo questa macchina M'



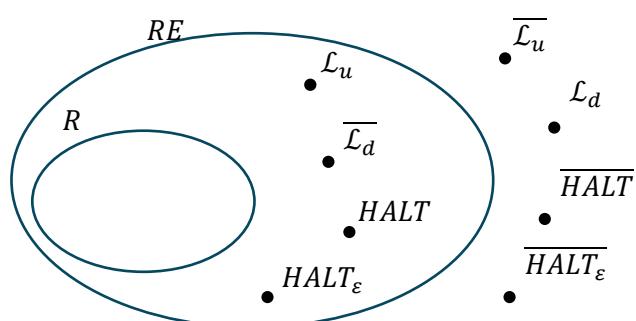
- Qual è il linguaggio riconosciuto da M' ? $\mathcal{L}(M')$
- Se M' ha risposto di sì, che cosa l'ha causato? $\overline{M_u}$ ha risposto di sì...
 - Ma $\overline{M_u}$ stava eseguendo su $M_i 111 w_i$. $\Rightarrow \overline{M_u} \models \overline{\mathcal{L}_u}$
 - $\Rightarrow M_i \not\models w_i$
- Se M' risponde di no, invece, vuol dire che $\overline{M_u}$ ha risposto di no su input $M_i 111 w_i$
 - $\Rightarrow M_i \models w_i$

Quindi M' decide il linguaggio \mathcal{L}_d $\mathcal{L}(M') = \mathcal{L}_d$

Ma ciò non è possibile, perché sappiamo che $\mathcal{L}_d \notin RE \Rightarrow$ assurdo $\Rightarrow \mathcal{L}_u \notin R$

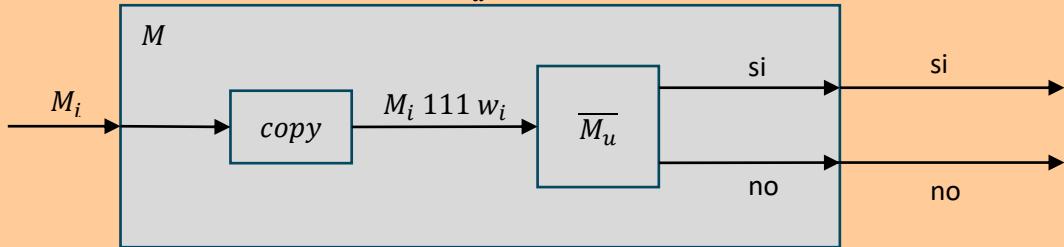
Ma quindi $\overline{\mathcal{L}_u} \in RE$

No, perché se $\mathcal{L}_u \wedge \overline{\mathcal{L}_u} \in RE$ assieme allora \mathcal{L}_u dovrebbe appartenere a R (THM pg precedente)



THM $\overline{\mathcal{L}_d} \in RE$

$$\overline{\mathcal{L}_d} = \{M_i \mid M_i \models w_i\}$$

*Proof*Dobbiamo mostrare una macchina che *accetti* $\overline{\mathcal{L}_d}$ > $M_i \in \overline{\mathcal{L}_d} \Rightarrow M$ risponde sì> $M_i \notin \overline{\mathcal{L}_d} \Rightarrow M$, risponde no oppure M non rispondeQuindi M decide $\overline{\mathcal{L}_d} \Rightarrow \overline{\mathcal{L}_d} \in RE$

(Vedi diagramma a pg 30)

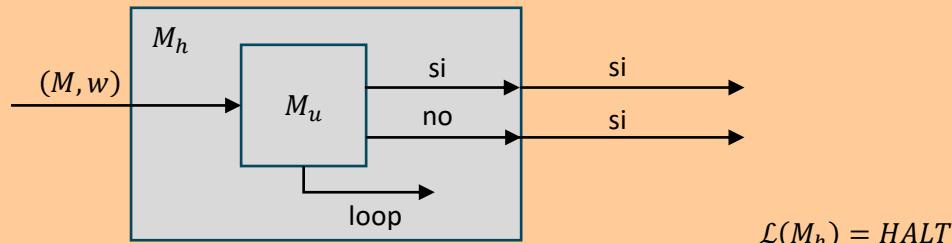
Però, $\overline{\mathcal{L}_d} \in R$? No perché, se così fosse \mathcal{L}_d dovrebbe appartenere ad RE

$$HALT = \{(M, w) \mid M \text{ si arresta su } w\}$$

$$\mathcal{L}_u = \{(M, w) \mid M \text{ accetta } w\}$$

Che differenza c'è tra il linguaggio $HALT$ e il linguaggio \mathcal{L}_u ?Risp: In \mathcal{L}_u la macchina si deve fermare e dire di sì.

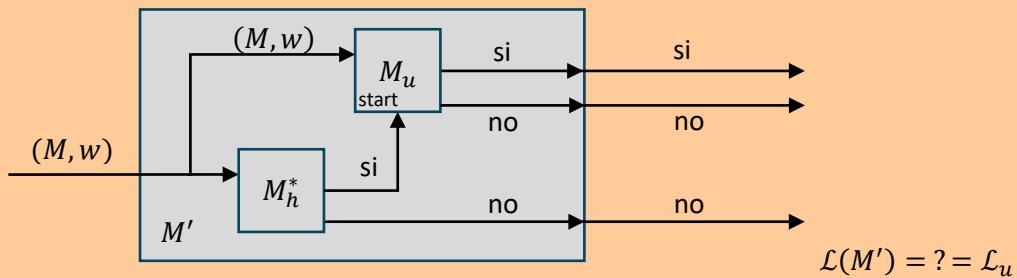
Quindi i due linguaggi sono diversi, sono definiti sulle stesse istanze ma sono diversi.

THM $HALT \in RE$ *Proof*

$$\mathcal{L}(M_h) = HALT$$

> $(M, w) \in HALT \Rightarrow M_h$ risponde sì> $(M, w) \notin HALT \Rightarrow M_h$ non risponde sì

(Vedi diagramma a pg 30)

THM $HALT \notin R$ *Proof*Supponiamo per assurdo che $HALT \in R \Rightarrow \exists$ una macchina M_h^* che decide $HALT$ 

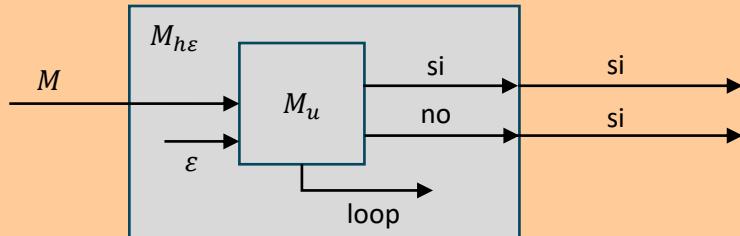
$$\mathcal{L}(M') = ? = \mathcal{L}_u$$

> M' risponde sì $\Rightarrow M \models w$ > M' risponde no $\Rightarrow \begin{cases} M \text{ risponde no su } w \\ M \text{ non si arresta su } w \text{ (no)} \end{cases} \Rightarrow M'$ decide \mathcal{L}_u Se $HALT \in R \Rightarrow$ saremmo in grado di costruire la macchina M' che decide $\mathcal{L}_u \Rightarrow \mathcal{L}_u \in R \Rightarrow$ assurdo.

$$HALT_{\varepsilon} = \{M \mid M \text{ si arresta su } \varepsilon\}$$

THM $HALT_{\varepsilon} \in RE$

Proof



$$\mathcal{L}(M_{h\varepsilon}) = HALT_{\varepsilon}$$

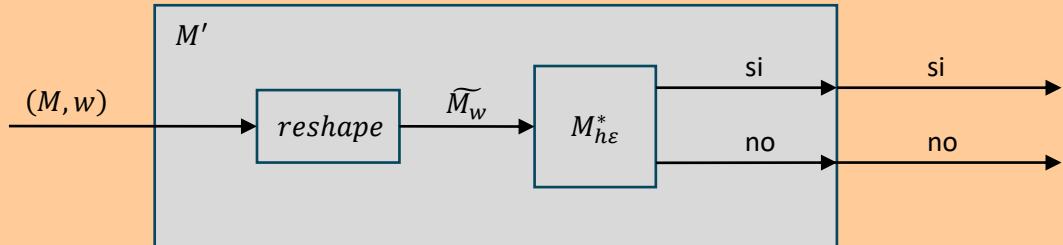
- > $M \in HALT_{\varepsilon} \Rightarrow M_{h\varepsilon} \text{ risponde sì}$
- > $M \notin HALT_{\varepsilon} \Rightarrow M_{h\varepsilon} \text{ non risponde sì}$

Quindi la macchina $M_{h\varepsilon}$ accetta il linguaggio $HALT_{\varepsilon}$

THM $HALT_{\varepsilon} \notin R$

Proof

Supponiamo per assurdo che $HALT_{\varepsilon} \in R \Rightarrow \exists M_{h\varepsilon}^* \text{ che decide } HALT$



$$\mathcal{L}(M') = ? = HALT$$

\widetilde{M}_w : scrive su nastro w , dopodiché fa partire la funzione di transizione di M

- > M' dice sì $\Rightarrow M$ si arresta su w
- > M' dice no $\Rightarrow M$ non si arresta su w

$\Rightarrow M'$ Decide $HALT \Rightarrow$ assurdo

4 RIDUZIONI

Le dimostrazioni dei teoremi del capitolo precedente avevano tutte un approccio simile. Dovevamo sempre dimostrare che un linguaggio apparteneva o meno a una classe di calcolabilità, per mostrare l'appartenenza dobbiamo mostrare un algoritmo; invece, se dobbiamo mostrare che un linguaggio non sta in quei due insiemi; quindi, dobbiamo dire *non esiste alcuna macchina che...* lavoravamo per assurdo; *supponiamo che questa cosa sia decidibile, costruivamo poi un'altra macchinina e ci infilavamo dentro la macchina assunta esistere, e succedevano casini.*

Queste erano delle istanziazioni, ovvero casi particolari di un principio molto più generale ovvero quella della riduzione tra linguaggi.

Con questo strumento tutte le dimostrazioni in cui dovremo mostrare che un linguaggio non appartiene a una certa classe verranno fatte tramite riduzione perché è più facile.

Il concetto di riduzione è fatto da vari elementi e per riuscire ad utilizzarlo dobbiamo stare attenti a una serie di cose e diventa facile perdersi per strada. Se però ci diamo una checklist delle cose da vedere allora la cosa dovrebbe diventare più fattibile. È comune capirlo male (parole del prof).

Per affrontarlo partiamo da una storiella (protagonista il prof).

Quando ero studente al 3° anno stavo facendo una serie di esami, nello specifico web-services, e dovevamo presentare un progetto a scelta, nello specifico una web-app per la gestione dei tornei di calcetto.

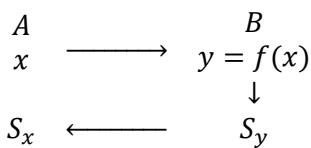
Le squadre si iscrivevano e l'app doveva poi creare un calendario per il torneo. La difficoltà incontrata era la necessità di essere sicuri che nella stessa giornata non finissero partite incompatibili; quindi, l'algoritmo doveva tirare fuori una riallocazione in sottoinsiemi di tutte le possibili partite tale per cui questa partizione fosse coerente. E ad un certo punto arrivò l'idea di "aggirare" il problema, al tempo stavamo seguendo il corso di design patterns e stavamo guardando cose come backtracking... E proprio per quel corso avevamo sviluppato un algoritmo in grado di colorare grafi.

Dato un grafo, fatto di nodi e archi, vogliamo colorare i suoi nodi in maniera tale che due nodi agganciati da un arco abbiano lo stesso colore.

E mi chiesi: posso convertire un'istanza per la generazione del torneo in un grafo da colorare e poi sfruttare la colorazione? Ovvero posso trasformare le istanze di questo problema in quest'altro problema così da usare l'algoritmo per la colorazione dei grafi e sfruttare la colorazione per tirare fuori il calendario.

L'astrazione è questa: abbiamo il problema **A di partenza** che vogliamo risolvere ma non lo vogliamo attaccarlo direttamente, abbiamo poi un problema **B di arrivo**, (quindi le riduzioni hanno un verso)! Vogliamo risolvere le istanze del problema **A** usando un algoritmo pronto per il problema **B**.

Questa trasformazione da **A** a **B** si chiama riduzione. Intuitivamente ridurre significa trasformare le istanze di **A** in istanze di **B** tale per cui una chiamata ad un solutore per **B** ci da una soluzione che possiamo riutilizzare per le istanze di **A**.



```

1. solverA (x : istanza di A) {
2.   y = f(x);      // la riduzione ha un verso
3.   Sy = SolverB(y);
4.   Sx = g(xy);
5.   return Sx;
6. }
```

$$g \approx f^{-1}$$

Tornando alla storiella...

$$\text{calendario} \leq k\text{-col}$$

Posso inventarmi una funzione f opportuna tale per cui io applico questa tecnica? Ovvvero:

Prendo le istanze del calendario, le trasformo in un grafo da colorare, faccio colorare il grafo dal solutore, mi prendo la soluzione (un grafo colorato) e da quella soluzione mi tiro fuori un calendario per il torneo.

$$\begin{matrix} \text{calend} \\ J, M, B, R \end{matrix} \leq \begin{matrix} k\text{-col} \\ \langle G, k \rangle \end{matrix}$$

Generiamo tutte le coppie possibili per il calendario:

$$\begin{pmatrix} J, M & J, R & M, R \\ J, B & M, B & B, R \end{pmatrix}$$

Partizioniamo poi questo insieme in sottoinsiemi che ci dicono quali sono le partite che vanno giocate nella stessa giornata

Una soluzione del problema del calendario è una partizione dell'insieme tale per cui in ogni singolo sottoinsieme non ci sono due partite incompatibili, due partite sono incompatibili se hanno una squadra in comune.

Da un lato dobbiamo generare insiemi di partite compatibili.

Da un lato abbiamo che le partite che finiscono nella stessa giornata non devono collidere.

Una soluzione del problema di colorare un grafo è un assegnamento di colori ai nodi tale per cui non accade che due nodi agganciati abbiano lo stesso colore.

Dall'altro dobbiamo generare colorazioni sensate.

Dall'altro lato dobbiamo avere una colorazione tale per cui i nodi con lo stesso colore non sono agganciati

Come sfruttiamo la faccenda?

$$\begin{matrix} \text{calend} \\ J, M, B, R \end{matrix} \xrightarrow{f} \begin{matrix} k\text{-col} \\ \langle G, k \rangle \end{matrix}$$

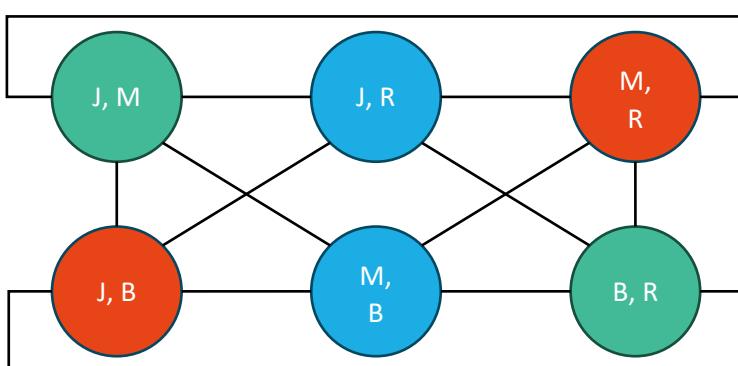
In input di f abbiamo una lista di squadre, in uscita di f ci sta un grafo e di un numero di colori

f prende in input la lista delle squadre e genera un grafo con tanti nodi quanti sono tutte le possibili partite; f collega con un arco tutte le partite incompatibili. L'idea è che i nodi colorati con un certo colore sono le partite che si svolgono in una stessa giornata, quindi $k = \text{numero di giornate}$. (4 squadre = 3 giornate)

Intuitivamente, quindi, una riduzione è una trasformazione di istanze di un problema di partenza in un problema di arrivo. Direzione.

Stabilire problema di partenza e di arrivo ci permette di capire cosa sono l'input e l'output di f .

f deve essere però progettata in maniera tale che la trasformazione in output è tale per cui la soluzione dell'istanza del problema di arrivo può essere riconvertita agevolmente in una soluzione dell'istanza del problema di partenza.



Quando siamo in grado di ridurre un problema A ad un problema B scriviamo:

$$A \leq B$$

DEF (Riduzione)

Siano A, B due linguaggi, la funzione f è una riduzione da A a B se per ogni stringa w

$$w \in A \Leftrightarrow f(w) \in B$$

è inoltre f è calcolabile.

Ma che cos'è una funzione calcolabile?

Per definire quand'è che una funzione è calcolabile dobbiamo vedere le MdT che calcolano un output,
Usiamo quindi un *Trasduttore*:

DEF (Trasduttore)

Un trasduttore è una Macchina di Turing, che calcola funzioni, caratterizzata da tre nastri:

1. INPUT Read-only
2. WORKTAPE Read/Write
3. OUTPUT Write-only

Un trasduttore M calcola la funzione f se per ogni stringa w , quando M esegue su w , al suo arresto lascia $f(w)$ in output.

La chiamiamo *trasduttore* per convenienza nostra, per capire se è un decisore o un calcolatore,
Quando è un calcolatore lo chiamiamo trasduttore, quando è un decisore Macchina di Touring.

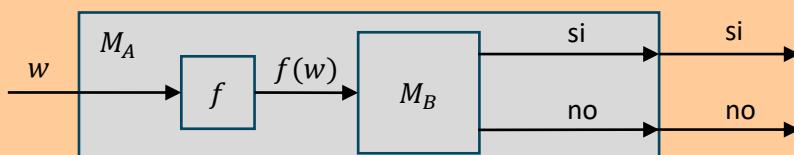
Una funzione calcolabile è una funzione per il quale esiste un trasduttore che la calcola sempre in tempo finito. Quindi una funzione è calcolabile se esiste un trasduttore che la calcoli in tempo finito.

THM: Siano A, B due linguaggi tali che $A \leq B$

- 1) Se $A \notin R \Rightarrow B \notin R$
- 2) Se $A \notin RE \Rightarrow B \notin RE$

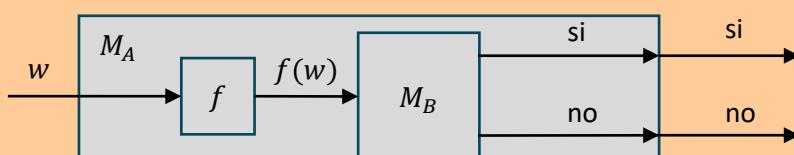
Proof

Assumiamo per assurdo che $A \notin R \wedge B \in R$, ciò vuol dire che esiste una MdT M_B che decide B
Ma allora potremmo costruirci un decisore per A in questo modo:



Quindi può esistere una MdT M_A che decide A , ma ciò è una contraddizione perché $A \notin R \Rightarrow$ assurdo

Assumiamo per assurdo che $A \notin RE \wedge B \in RE$, ciò vuol dire che esiste una MdT M_B che accetta B
Ma allora potremmo costruirci un accettatore per A in questo modo:



Quindi può esistere una MdT M_A che accetta A , ma ciò è in contraddizione perché $A \notin RE \Rightarrow$ assurdo

Corollario

Dalla logica: $X \Rightarrow Y \equiv \neg X \vee Y \equiv \neg Y \Rightarrow \neg X$; Applicandolo al teorema di prima:

Siano A, B due linguaggi tali che $A \leq B$

Se $B \in R \Rightarrow A \in R$

Se $B \in RE \Rightarrow A \in RE$

Vediamo ora un altro linguaggio che vogliamo dimostrare essere indecidibile ma questa volta dimostriamo la sua indecidibilità tramite riduzione

$$\mathcal{L}_{ne} = \{M_i \mid \mathcal{L}(M_i) \neq \emptyset\}$$

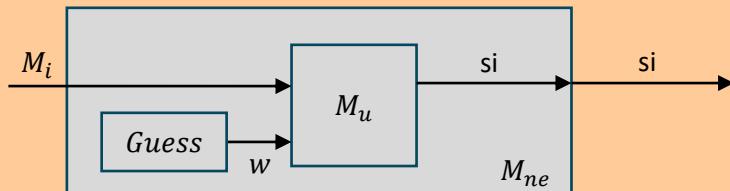
Questo linguaggio definisce le macchine che accettano almeno una stringa.

Vogliamo studiare la proprietà di decidibilità di questo programma

THM $\mathcal{L}_{ne} \in RE$

Proof

Costruiamo una macchina per \mathcal{L}_{ne}



Un algoritmo per questa macchina è questo, la macchina M_{ne} non sa fin dal principio qual è la stringa che M_i può accettare, quindi la indovina. La indoviniamo, piuttosto che provarla su tutte le w perché su alcune potrebbe non terminare.

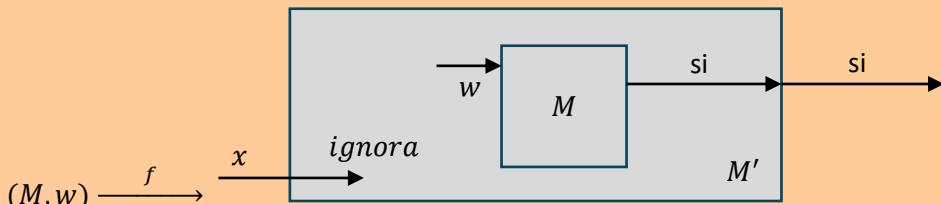
Quindi $\mathcal{L}_{ne} \in RE$

THM $\mathcal{L}_{ne} \notin R$

Proof

$$\begin{array}{ccc} \mathcal{L}_u & \xrightarrow{\quad \mathcal{L}_u \leq \mathcal{L}_{ne} \quad} & \mathcal{L}_{ne} \\ (M, w) & \xrightarrow{f} & M' \end{array}$$

$$\begin{aligned} (M, w) \in \mathcal{L}_u &\rightarrow f((M, w)) = M' \in \mathcal{L}_{ne} \\ (M, w) \notin \mathcal{L}_u &\rightarrow f((M, w)) = M' \notin \mathcal{L}_{ne} \end{aligned}$$



La funzione f partendo dalla coppia (M, w) genera la macchina M' , è una macchina che ignora il proprio input, scrive sul nastro w ed esegue la funzione di transizione di M sulla w che si è scritta da sola.

- > $\Rightarrow (M, w) \in \mathcal{L}_u \Rightarrow \mathcal{L}(M') = \Sigma^* \neq \emptyset$
- > $\Leftarrow (M, w) \notin \mathcal{L}_u \Rightarrow \mathcal{L}(M') = \emptyset$

4.1 POST CORRESPONDENCE PROBLEM (PCP)

Problema di corrispondenza di Post

È un problema su stringhe che si configura così: Abbiamo due liste di stringhe definite sullo stesso alfabeto

	A	B	
r_1	1	111	s_1
r_2	10111	10	s_2
r_3	10	0	s_3

Quello che ci chiediamo è *date due liste di stringhe A e B, è vero o no che $\exists i_1, i_2, \dots, i_n, n > 0$ tali per cui*

$$r_{i_1} \circ r_{i_2} \circ \dots \circ r_{i_n} = s_{i_1} \circ s_{i_2} \circ \dots \circ s_{i_n}$$

Un'istanza sì di questo problema è un'istanza tale per cui esiste una sequenza di indici tali che, se prendiamo le stringhe dalla lista A secondo l'ordine di quegli indici e le stringhe dalla lista B secondo l'ordine di quegli indici arriviamo a costruire la stessa cosa.

ie. una soluzione per l'istanza sopra scritta è

$$\begin{aligned} & 2, 1, 1, 3 \\ r \Rightarrow & 10111 \circ 1 \circ 1 \circ 10 = 101111110 \\ s \Rightarrow & 10 \circ 111 \circ 111 \circ 0 = 101111110 \end{aligned}$$

Questo problema sta in *RE* o no ? (Se sta in *RE* deve esistere una MdTN che lo accetti)

Si, perché con una MdTN possiamo indovinare una sequenza finita di indici, concateniamo le due stringhe, essendo una sequenza finita il tempo con cui svolgiamo le operazioni è tempo finito, confrontiamo, se sono uguali diciamo di sì.

Questo algoritmo da garanzia di risposta no ?

No.

Il problema sta in *R* ?

No.

Lo dimostriamo tramite una riduzione su \mathcal{L}_u , in particolare lo ridurremo prima a una versione modificata, detta *Modified PCP (MPCP)* che ridurremo a sua volta a *PCP*

$$\mathcal{L}_u \leq MPCP \leq PCP$$

Si, le riduzioni godono della transitività, lo vedremo nello specifico più avanti.

MPCP: è una variante di *PCP* in cui abbiamo le stesse cose, due liste, le stringhe, ... vogliamo calcolare una sequenza di indici... L'unica differenza è che su *MPCP* il primo indice dev'essere 1 per forza.

Possiamo quindi dire che *esiste una riduzione poiché MPCP è un caso particolare di PCP*?

No, perché potrebbe essere che *MPCP*, essendo un caso particolare sia più facile; quindi, ci serve una dimostrazione formale

Riduciamo \mathcal{L}_u a $MPCP$

$$\begin{array}{ccc} \mathcal{L}_u & \rightarrow & MPCP \\ (M, w) & & (A, B) \end{array}$$

Quali sono le istanze di \mathcal{L}_u ?

Coppie macchina-stringa

Quali sono le istanze di $MPCP$?

Due liste A e B

Dobbiamo inventarci come trasformare una coppia (M, w) in due liste di stringhe A e B

E questa trasformazione non deve essere "a buffo"; deve trasformare istanze sì di \mathcal{L}_u in istanze sì di $MPCP$ e istanze no di \mathcal{L}_u in istanze no di $MPCP$.

Istanze sì di \mathcal{L}_u

Coppie (M, w) tali per cui se la M processa w a un certo punto si ferma e dice "sì"

Istanze sì di $MPCP$

Coppie di liste di stringhe A e B tali per cui esiste una sequenza di indici che, se prendiamo le stringhe provenienti da A e provenienti da B costruiamo la stessa cosa.

Istanze no di \mathcal{L}_u

Coppie (M, w) tali per cui M non dice di sì

Istanze no di $MPCP$

Coppie di liste di stringhe tali per cui non esiste la sequenza

Inventiamoci ora un modo per trasformare le istanze (M, w) in due liste di stringhe:

Quello che noi dobbiamo ottenere è far sì che tramite le stringhe delle liste A e B vogliamo simulare il comportamento della macchina M su w . Quindi dobbiamo inventarci un modo per infilare stringhe in A e B tali per cui se la macchina M accetta w riusciamo ad avere le corrispondenze, a costruire una sol di $MPCP$; se la macchina M non accetta w allora noi questa ricostruzione non dobbiamo essere in grado di farla.

L'idea alla base della riduzione è la seguente:

Ricordiamo che la computazione di ogni MdT su una certa stringa può essere codificata tramite la sequenza dell'*instantaneous description* che la macchina attraversa, cioè la sequenza delle configurazioni che la macchina attraversa sono la storia del calcolo di M su w , e queste configurazioni? Le possiamo rappresentare come stringhe. Se pezzi di configurazioni le possiamo infilare dentro A e dentro B siamo più o meno in grado di ricostruire la faccenda.

Noi avremo che una computazione di M su w sarà

$\# \alpha_1 \# \alpha_2 \# \alpha_3 \# \dots$

$\alpha_n \vdash_M \alpha_{n+1}$

Quindi noi andremo a infilare pezzi di questa stringa dentro ad A e B ,

Dobbiamo quindi inventarci delle coppie di stringhe (r_i, s_i) in maniera tale che tutto torni.

Ci sono cinque classi di coppie di stringhe (r_i, s_i) che servono a permetterci di ricostruire il funzionamento di M su w :

$$\begin{aligned} r &\rightarrow \# \alpha_1 \# \alpha_2 \\ s &\rightarrow \# \alpha_1 \# \alpha_2 \# \alpha_3 \end{aligned}$$

La stringa codificante la computazione di M su w sarà possibile ottenerla sia tramite la concatenazione degli r_i sia tramite la concatenazione degli s_i , ma s_i sarà "*un passo avanti*" rispetto a r_i . Le stringhe provenienti da A potranno raggiungere quelle provenienti da B solo se la macchina entra in uno stato accettante.

	A	B	
1)	#	$\#q_0w_1 \cdots w_n\#$	
2)	X #	X #	$\forall X \in \Gamma$
3)	qX ZqX $q\#$ $Zq\#$	Yp pZY $Yp\#$ $pZY\#$	$\delta(q, x) = (p, Y, \rightarrow)$ $\delta(q, x) = (p, Y, \leftarrow)$ $\delta(q, \#) = (p, Y, \rightarrow)$ $\delta(q, \#) = (p, Y, \leftarrow)$
4)	$X \quad q \quad Y$ $X \quad q$ $q \quad Y$	q q q	
5)	$q\#\#$	#	
	$q \in F$		

Cosa fa questa funzione di trasformazione f ?

Prende in input la coppia (M, w) e sputa fuori liste A e B dove stanno coppie di stringhe che seguono queste regole

Supponiamo di avere una coppia (M, w) da dare in input a f , ora disegniamo M , diciamo cos'è w e vediamo cosa esce fuori al termine della trasformazione:

Supponiamo quindi che M sia questa cosa e che $w = 01$

Quindi abbiamo questa coppia (M, w) che viene data in pasto alla funzione f di riduzione. La funzione f di riduzione legge la funzione di transizione di M che gli viene data in input tramite la codifica delle macchine, f quindi riceve M , riceve w e deve sputare in output le liste A e B .

Vediamo la computazione di questa macchina sulla stringa $w = 01$

$$q_1 01 \vdash 1q_2 1 \vdash 10q_1 \not{1} \vdash 1q_2 01 \vdash q_3 101$$

Quello che vogliamo vedere è che la simulazione che otteniamo tramite la riduzione che abbiamo proposto prima va a replicare esattamente il funzionamento di questa macchina

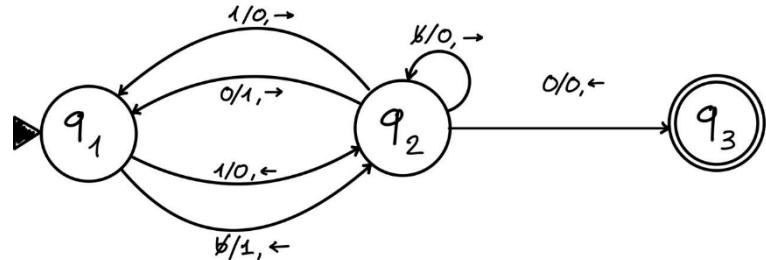
$$\begin{array}{lllllll} A & & & & & & B \\ \hline \# & & & & & & \#q_1 01 \# \\ 0 & & & & & & 0 \\ 1 & & & & & & 1 \\ \not{1} & & & & & & \not{1} \\ \# & & & & & & \# \\ \hline q_1 & 0 & & & & 1 & q_2 \\ 0 & q_1 & 1 & & & q_2 & 0 & 0 \\ 1 & q_1 & 1 & & & q_2 & 1 & 0 \\ q_2 & 1 & & & & 0 & q_1 & \\ \vdots & & & & & \vdots & & \end{array}$$

Concentriamoci su B sulla prima coppia *stato-simbolo* $[q_1 0]$

Sulla tabella corrisponde a $[1q_2]$ che corrisponde al ramo da q_1 a q_2 leggo 0 scrivo 1 e vado verso destra. Quindi posso estendere i ... come nella colonna successiva.

Questa cosa va avanti fino a quando noi raggiungeremo una configurazione finale tale per cui possiamo utilizzare le regole di 4) nella quale sono più lunghe le stringhe provenienti da A che quelle da B ; e a questo punto la stringa da A raggiungerà quella B .

La 5) serve a chiudere il pattern.



Dobbiamo dimostrare che questa è una riduzione da \mathcal{L}_u a $MPCP$

- 1) Dobbiamo dimostrare che, se partiamo da un'istanza sì di $\mathcal{L}_u \Rightarrow$ il trasformato della coppia (M, w) è una coppia di liste di stringhe A, B tali per cui ammettono una soluzione secondo $MPCP$

La dimostrazione sta nella costruzione, essa permette di simulare il funzionamento di M su w tramite la concatenazione di stringhe che descrivono le configurazioni che la macchina attraversa di passo in passo.

Se M accetta w la sequenza di indici che possiamo costruire per la soluzione di $MPCP$ parte dalla prima coppia per cui partiamo da:

$$\begin{aligned} A &\rightarrow \# \\ B &\rightarrow \#q_0w\# \end{aligned}$$

Abbiamo poi regole di II classe che ci permettono di ricopiare i pezzi della configurazione distanti dalla testina. Mentre le regole di III classe ci permettono di allungare le stringhe per le porzioni di configurazioni che stanno nell'intorno della testina e quindi di simulare il comportamento della macchina.

Quindi se $M \models w$ noi saremo in grado di concatenare pezzi di stringhe fino ad arrivare al momento in cui in queste configurazioni apparirà uno stato finale; se appare lo stato finale potremo usare le regole di IV classe nelle quali le istanze di A sono più lunghe di quelle di B e permettiamo alla stringa proveniente da A di raggiungere la stringa proveniente dalla lista B . Fino a quando come ultima coppia possiamo usare la regola di V classe; che ci permette di matchare le due stringhe.

Con questo dimostriamo che, se partiamo da un'istanza sì di \mathcal{L}_u , stiamo costruendo il trasformato della coppia (M, w) cioè, un'istanza sì del problema $MPCP$.

- 2) Supponiamo che la riduzione ci abbia costruito un'istanza sì di $MPCP$, dobbiamo dimostrare che stavamo partendo da un'istanza sì di \mathcal{L}_u .

Se siamo arrivati a un'istanza sì di $MPCP$; l'istanza sì che abbiamo costruito non è un'istanza a caso, è un'istanza che ha la forma delle due colonne A e B della tabella delle regole di classe.

Quindi una possibile soluzione dovrà partire per forza con la I regola di classe (e conseguenti coppie di stringhe). Che inizierà a far partire la simulazione della macchina sulla stringa in input; poiché le coppie di stringhe ce le siamo inventate apposta per simulare il funzionamento della macchina, se riusciamo a costruire delle stringhe che quando si estendono arrivano a costruire la stessa cosa è perché a un certo punto avremo usato coppie della classe IV e V, di conseguenza nella simulazione della macchina M deve arrivare nello stato accettante, da cui l'istanza di partenza di \mathcal{L}_u da cui partivamo è un'istanza sì.

Questa cosa dimostra che:

$$\mathcal{L}_u \leq MPCP \Rightarrow MPCP \notin R$$

La cosa interessante di tutto questo magheggio strano è che abbiamo dimostrato l'indecidibilità di $MPCP$ senza fare dimostrazioni strane dell'inesistenza di macchine etc. etc. Ma abbiamo preso un problema indecidibile (\mathcal{L}_u) e abbiamo dimostrato che le sue istanze sì possono essere mappate su quelle di $MPCP$ e che le istanze no di \mathcal{L}_u possono essere mappate su istanze no di $MPCP$.

Per questa ragione, siccome $MPCP$ mostra una struttura simile a \mathcal{L}_u , dev'essere per forza indecidibile. Altrimenti potremmo risolvere \mathcal{L}_u tramite un risolutore di $MPCP$.



La riduzione precedente ci dimostra che $MPCP$ è indecidibile, come facciamo a dimostrare che $PCP \notin R$?

Con un'altra riduzione

$$\begin{array}{ccc} MPCP & \leq & PCP \\ (A, B) & \rightarrow & (C, D) \end{array}$$

Per capirla meglio svolgiamo la riduzione su di un problema specifico

A	B	\rightarrow	C	D
1	111		1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

Che fa questa funzione di transizione? Prende la coppia di liste A e B con le sue stringhe al suo interno; sputa fuori due liste di stringhe C e D (chiamate così solamente per distinguerle) in cui le coppie di stringhe che fanno parte di (C, D) provengono dalle coppie di stringhe che stavano in (A, B) ; Quindi prendiamo (A, B) le modifichiamo un pochettino in maniera tale da ottenere la riduzione.

Per ognuna delle stringhe provenienti da A mettiamo una stringa in C in cui ogni simbolo della stringa di partenza *viene seguito da un asterisco*.

Le stringhe provenienti da B vengono trasformate in stringhe di D in un modo simile, nel quale *l'asterisco invece di seguire il simbolo, lo precede*.

Come si può vedere, allo stato attuale della trasformazione, l'istanza (C, D) non ammetterà mai una soluzione perché tutte le stringhe della lista C partono con un simbolo diverso dalle stringhe della lista D . Quindi la riduzione non è ancora finita, dobbiamo aggiungere dei pezzi.

Il pezzo aggiuntivo è che in C prendiamo la stringa proveniente dalla prima coppia $(1 *)$ e ne facciamo un'altra copia, mettiamo un asterisco all'inizio e poi la ricopiamo. Per D è semplicemente quella di prima ricopiata.

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 *	* 1 * 1 * 1
10	0		1 * 0 *	* 1 * 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D
1	111		* 1 *	* 1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1 *	* 1 * 0
10	0		1 * 0 *	* 0

A	B	\rightarrow	C	D

<tbl_r cells="5" ix="4

La parte fondamentale è che la funzione di riduzione non può sapere se l'istanza di partenza è 'sì' o 'no'. Perché per saperlo dovrebbe tentare di risolvere un problema indecidibile, ma la funzione di trasformazione dev'essere calcolabile, ovvero la sua risposta la deve dare in tempo finito, quindi f non dovrà mai tentare di risolvere l'istanza di partenza. Deve fare una trasformazione di struttura.

Per questo problema specifico lo fa nel modo descritto sopra.

Ora dobbiamo dimostrare che $MPCP$ si riduce a PCP

- 1) Supponiamo che partiamo da una coppia di liste (A, B) che sia un'istanza 'sì' di $MPCP$, Se è un'istanza sì di $MPCP$ allora f ha sputato fuori un'istanza 'sì' di PCP .

Vuol dire che esiste una sequenza di indici tale per cui costruiamo le stesse stringhe, per convenienza le chiamiamo r_i, s_i e x_i, y_i , giusto per orientarci.

A	B	\rightarrow	C	D
1	111		0 * 1 *	* 1 * 1 * 1
10111	10		1 1 *	* 1 * 1 * 1
10	0		2 1 * 0 * 1 * 1 * 1 *	* 1 * 0
			3 1 * 0 *	* 0
			4 \$	* \$
r_i	s_i		x_i	y_i

Se (A, B) è un'istanza sì abbiamo una sequenza di indici particolari perché parte con 1, quindi:

$$1 \quad i_2 \quad i_2 \quad \dots \quad i_m$$

Tale per cui avremmo che

$$r_{i1} \circ r_{i2} \circ r_{i3} \circ \dots \circ r_{im} = s_{i1} \circ s_{i2} \circ s_{i3} \circ \dots \circ s_{im}$$

Partendo da questa che è una soluzione per l'istanza di $MPCP$ mostriamo che c'è una soluzione anche per l'istanza di PCP ; e facciamo così:

$$x_{i1} \circ x_{i2} \circ x_{i3} \circ \dots \circ x_{im} \quad y_{i1} \circ y_{i2} \circ y_{i3} \circ \dots \circ y_{im}$$

Questa sequenza di stringhe costruita con questi pezzi è *quasi* uguale, perché le stringhe x_{i1}, x_{i2}, \dots hanno gli asterischi dopo mentre quelle y_{i1}, y_{i2}, \dots non hanno gli asterischi dopo.

Se metto un asterisco così mi risolvo un pezzo del problema. L'unica questione è che la stringa x_{im} finisce con * mentre y_{im} finisce con un simbolo standard, e li utilizzo la coppia $(\$, \$)$ e in questo modo le due stringhe sono uguali.

$$\underbrace{x_{i1}}_{x_0} \circ x_{i2} \circ x_{i3} \circ \dots \circ x_{im} \underbrace{\$}_{x_{k+1}} \quad \underbrace{y_{i1}}_{y_0} \circ y_{i2} \circ y_{i3} \circ \dots \circ y_{im} \underbrace{\$}_{y_{k+1}}$$

E questo dimostra che, se partiamo da un'istanza sì di $MPCP$, arriviamo ad un'istanza 'sì' di PCP .

- 2) Ora dobbiamo dimostrare che, se siamo arrivati ad un'istanza 'sì' di PCP è perché partivamo da un'istanza 'sì' di $MPCP$.

Supponiamo di **essere arrivati** ad un'istanza 'sì' di PCP , quindi avremo una sequenza di indici tali per cui se concateniamo le stringhe costruiamo la stessa cosa. Per com'è stata costruita nello specifico l'istanza dei PCP la soluzione deve per forza partire da indice 0 e finire sull'ultima coppia perché abbiamo costruito l'istanza in questo modo. La sua soluzione deve per forza partire dalla coppia di indice 0, perché è l'unica coppia che ha lo stesso simbolo all'inizio. Quindi noi potremmo trasformare una soluzione dell'istanza di arrivo di PCP in una soluzione dell'istanza di partenza di PCP , perché abbiamo *forzato* tramite una costruzione opportuna che tutte le possibili soluzioni sull'istanza di arrivo contemplino solamente come partenza la prima coppia. Quindi se siamo arrivati ad un'istanza sì di PCP è perché partivamo da un sì di $MPCP$

Quindi $MPCP$ si riduce a PCP e quindi PCP è indecidibile



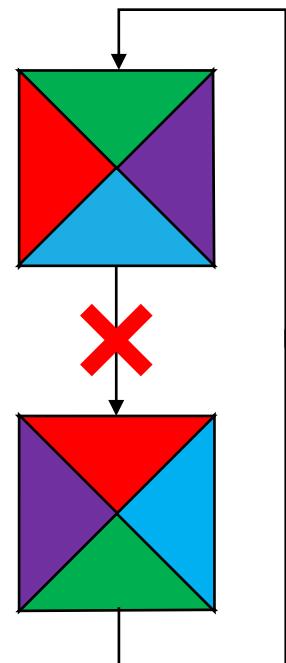
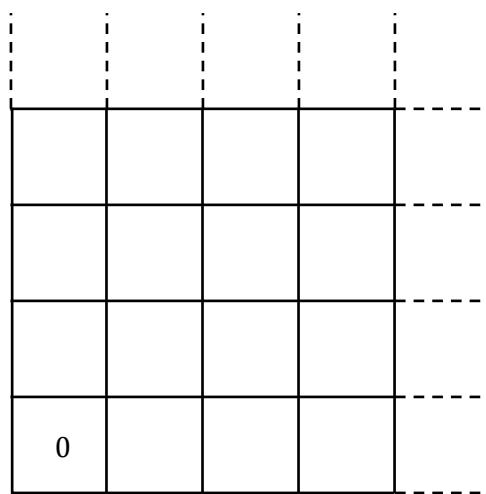
4.2 TILING PROBLEM

Problema del tassellamento

Dimostreremo che anche questo problema è indecidibile attraverso a una riduzione.

Consideriamo questo problema:

Supponiamo di avere una superficie che vogliamo ricoprire con delle piastrelle, questa superficie può essere suddivisa in dei riquadri (di unità 1) e noi dovremmo andare a sistemare dentro ai riquadri delle piastrelle.



Questa superficie, essendo il primo quadrante cartesiano è una superficie infinita che ha un'origine, $(0,0)$. Avremo varie piastrelle che potranno essere accostate orizzontalmente o verticalmente secondo alcune regole.

Come sono fatte queste piastrelle:

Una piastrella è divisa in quattro parti ciascuna con un colore diverso; ognuna di queste piastrelle è caratterizzata dalla colorazione e dalla loro disposizione. Le piastrelle non possono essere girate. Non possiamo allungarle ma hanno dimensione fissata. Devono essere messe una a fianco all'altra se si vanno a toccare con il lato con lo stesso colore.

Tra le varie piastrelle ne avremo una particolare che andrà posta per forza in posizione 0.

❓ Dato un sistema di questo tipo, è vero o no che è possibile ricoprire tutto il primo quadrante cartesiano?

Questo problema si può dimostrare essere indecidibile!

INPUT: $T = (D, d_0, H, V)$

D : insieme finito di tipi di piastrelle

$H \subseteq D \times D$: insieme delle regole di affiancamento orizzontale

$d_0 \in D$: è il tipo di piastrella ammessa in $(0,0)$

$V \subseteq D \times D$: insieme delle regole di affiancamento verticale

Un Tiling attraverso il sistema T è una funzione $f: \mathbb{N} \times \mathbb{N} \rightarrow D$, che mappa la posizione della cella sul quadrante con il tipo di piastrella che vogliamo infilare lì tale per cui l'affiancamento verticale e orizzontale rispetta le regole.

- > $f(0,0) = d_0$
- > $(f(m,n), f(m+1, n)) \in H \quad \forall m, n \in \mathbb{N}$
- > $(f(m,n), f(m, n+1)) \in V \quad \forall m, n \in \mathbb{N}$

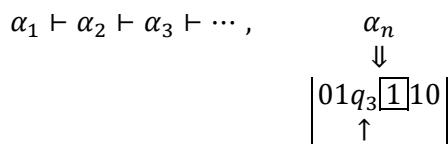
Come abbiamo detto questo problema è indecidibile, e per dimostrarlo proporremo una riduzione del tipo:

$$\begin{array}{ccc} \overline{\text{HALT}_\varepsilon} & \leq & \text{Tiling} \\ M & \xrightarrow{g} & T = g(M) \end{array}$$

La T che sputiamo fuori è un sistema di Tiling che ammetta un reale Tiling se e solo se non si arresta mentre processa la stringa vuota. Quindi, come abbiamo fatto per il PCP dovremo codificare nel problema di Tiling le computazioni della macchina, quindi dobbiamo inventarci questa cosa.

Ora invece che, come è stato fatto per PCP, spiegare prima la riduzione, facciamo prima l'esempio così da rendere più abbordabile l'intuizione.

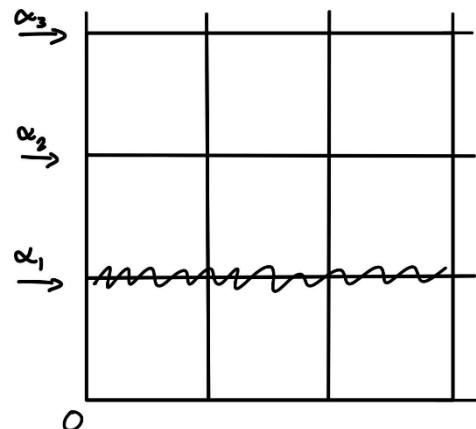
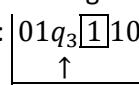
Noi dobbiamo riuscire a individuare nel problema di Tiling dove codificare la computazione di una MdT. Ricordiamo che le computazioni di MdT possono essere efficacemente rappresentate come sequenze di configurazioni che la macchina attraversa mentre processa una certa stringa.



Un'assunzione che facciamo sulla macchina M (istanza di \overline{HALT}_ϵ) è che: se noi limitiamo la MdT ad un nastro *semi-infinito*, cioè che ha un punto di partenza e non sposta mai la testina alla sinistra di quel punto di partenza, la macchina ha lo stesso potere delle macchine standard. Questo ci serve per semplificare la riduzione.

Noi sfrutteremo i bordi fra ogni livello di piastrelle; perché essendo un reticolo, le piastrelle le possiamo immaginare come 1°, 2°, 3° fila... Il bordo fra le file lo sfrutteremo per codificare le configurazioni della macchina che calcola.

Quindi dovremmo avere un numero di colori sufficientemente ampio, tale per cui i colori che mi appaiono sulla prima riga mi stanno descrivendo una configurazione di questo tipo:



Noi sappiamo, per le regole di affiancamento, che i colori a cavallo fra la prima e la seconda fila devono essere gli stessi, altrimenti non riusciremmo ad impilare le piastrelle. L'intuizione è dunque questa: *avere una tavolozza di colori sufficientemente ampia, tale per cui tramite quei colori riusciamo a codificare configurazioni di MdT.*

Quindi sul bordo fra la fila 1 e la fila 2 avremo la prima configurazione (α_1), sul bordo tra la seconda fila e la terza fila avremo la seconda (α_2).

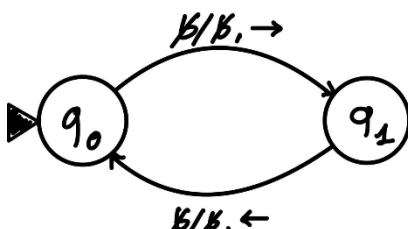
L'intuizione è dunque inventarsi delle piastrelle tali che, quando le andiamo a posare sul piano, il loro affiancamento mi sta codificando il funzionamento della macchina M .

Sulla parte orizzontale noi dobbiamo codificare la sequenza delle configurazioni che la MdT M attraverserebbe nel momento in cui stesse progettando la stringa vuota. In maniera tale che, se M non si arresta mentre processa la stringa vuota, allora di file di piastrelle ne possiamo mettere infinite.

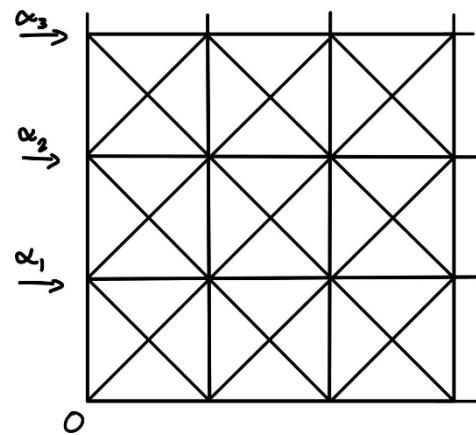
Se la macchina si ferma, allora le piastrelle ad un certo punto si fermano perché poi la tipologia di piastrelle che avremo non ci permetterà di piazzare una nuova fila; se invece la macchina M non si arresta su ϵ , allora noi avremo tipologie di piastrelle che ci permetteranno di mettere file di piastrelle una sull'altra senza fine.

Siccome noi dobbiamo codificare il funzionamento della MdT che può essere generico a noi servono molti tipi di piastrelle e far riferimento ai colori diviene complicato; quindi, invece di colorare le quattro porzioni delle piastrelle gli metteremo un'etichetta.

Supponiamo di avere questa macchina: che non si arresta su ϵ .



Utilizzando le etichette piuttosto che i colori dovrà succedere che le sezioni che si toccano dovranno avere lo stesso nome.



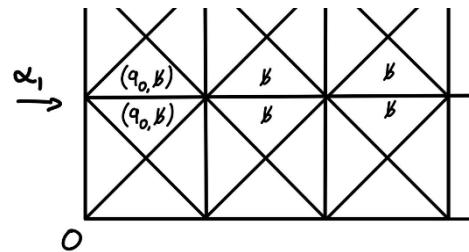
Qual è la configurazione iniziale della macchina su stringa vuota?

$$\alpha_1 = q_0 \mathcal{B} \mathcal{B} \mathcal{B} \mathcal{B}$$

Noi vorremmo che le etichette che appaiono sul bordo tra la prima e la seconda fila siano (q_0, \mathcal{B}) che ci dice che in posizione 1 abbiamo la testina e stiamo leggendo bianco mentre siamo in q_0 .

Però siccome non possiamo andare a guardare tutte le sfumature prendiamo queste etichette.

Poi avremo \mathcal{B} nelle altre sezioni a lato perché la configurazione è scritta così all'infinito. E quindi le porzioni sopra avranno le stesse etichette (q_0, \mathcal{B}) e \mathcal{B} .

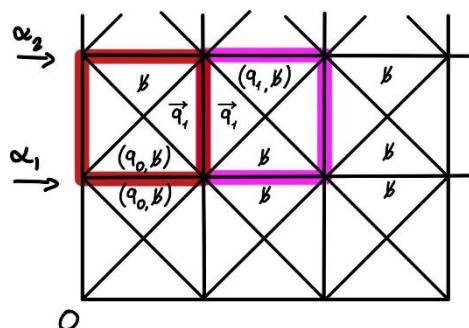


❓ Ora come facciamo ad imporre che α_2 vada a codificare la configurazione successiva?

❓ Ma prima: data la configurazione α_1 qual è la configurazione successiva secondo questa macchina?

$$\alpha_1 \vdash \alpha_2 = \mathcal{B} q_1 \mathcal{B} \mathcal{B} \mathcal{B}$$

Quindi se voglio che questo affiancamento di piastrelle mi vada a replicare il passaggio $\alpha_1 \vdash \alpha_2$, allora queste due piastrelle mi devono dare la possibilità di codificare che lo stato passi da q_0 a q_1 e vada avanti di una posizione. Per farlo stabilisco delle etichette di colori che replicano queste cose. Quindi scriverò $\boxed{\mathcal{B} q_1 \mathcal{B} \mathcal{B}}$



Avremo quindi bisogno di: una piastrella che simula il passaggio di q_0 su \mathcal{B} e di una piastrella che simula il passaggio di q_1 verso di lì.

Per fare ciò e non avere cose strane ai lati dovremmo utilizzare dei colori strani in modo tale che **questa piastrella** posso essere affiancata solamente a **quest'altra**. Per farlo usiamo un'etichetta che chiamiamo \vec{q}_1 indicando che la testina è in transizione sullo lo stato q_1 verso destra.

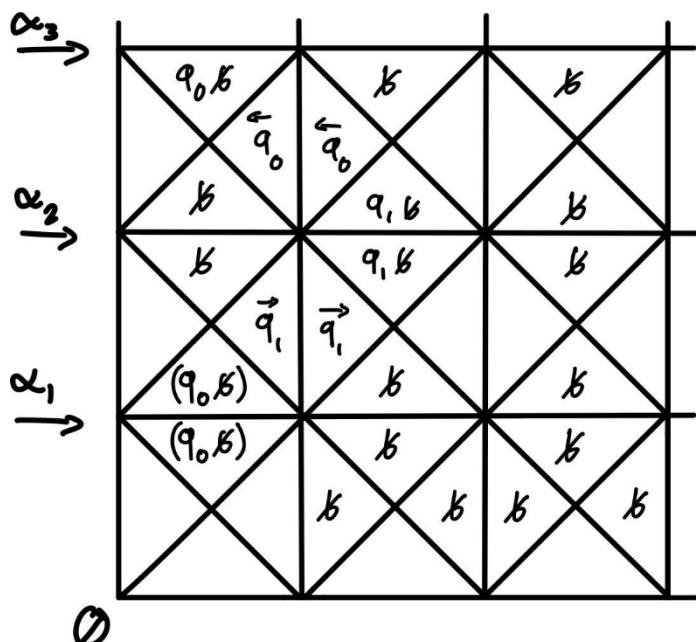
Quindi noi avremo necessità di u piastrelle che abbiano **questa faccia** in modo tale che noi sul bordo di α_2 lo stiamo codificando.

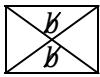
$$\alpha_3 = q_0 \mathcal{B} \mathcal{B} \mathcal{B} \dots$$

Avremo dunque una fila di piastrelle così:

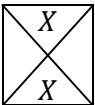
Quindi noi ci stiamo inventando delle piastrelle i cui colori, identificati tramite etichette, hanno lo scopo primario di codificare la sequenza di configurazioni che la macchina M attraversa mentre processa la stringa vuota guardando ai bordi orizzontali fra i livelli di piastrelle. Ora quest'idea va generalizzata ad una generica MdT M che potremmo avere in input alla funzione di trasformazione g .

Ora generalizziamo in modo da descrivere g in modo generico:



Dalle piastrelle  se avessimo dei simboli nelle sezioni superiori delle piastrelle a lato, essi andrebbero replicati; quindi, servono delle piastrelle che simulano il fatto che, se la testina su nastro è distante, lì non deve succedere niente e quindi quel colore dal bordo di sotto dev'essere passato al bordo di sopra

Quindi avremo una prima tipologia di piastrelle:

- 1)  $\forall X \in \Gamma$ Sui bordi non infilo niente per dire che è l'etichetta vuota e noi possiamo affiancare piastrelle con quel bordo verticale solo se hanno la stessa etichetta, quindi se non c'è niente posso mettere piastrelle solo con quella porzione vuota, non ci posso affiancare una piastrella dove lì c'è scritto altro; è come se fosse un nome aggiuntivo.

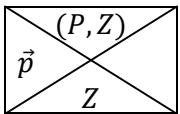
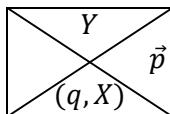
Ci servono ora piastrelle che codifichino la funzione di transizione e lo spostamento della testina da un lato all'altro. Perché la tipologia 1 sono quelle piastrelle che simulano una porzione di nastro lontana dalla testina, e lì c'è un'inerzia, il simbolo è quello e quello deve rimanere fino a quando non si avvicina la testina.

- 2) Supponiamo che dobbiamo codificare una generica transizione di configurazioni in cui abbiamo

$$qXZ \rightarrow YpZ$$

$$\delta(q, X) = (P, Y, \rightarrow)$$

Quindi se quello è un generico pezzettino di una configurazione e siamo in presenza di un pezzo di funzione di transizione così, allora vuol dire che transiamo da qXZ a YpZ , adesso dobbiamo farne le piastrelle abbiammo:



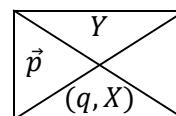
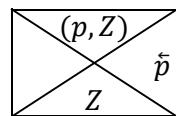
$$\begin{aligned} \forall q, p \in Q \setminus F \\ \forall X, Y, Z \in \Gamma \end{aligned}$$

Queste sono le piastrelle che simulano scrittura e spostamento di testina a destra

Ora ci servono le piastrelle che simulano scrittura e spostamento di testina a sinistra, avremo quindi:

$$ZqX \rightarrow pZY$$

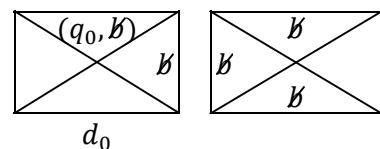
$$\delta(q, X) = (P, Y, \leftarrow)$$



$$\begin{aligned} \forall q, p \in Q \setminus F \\ \forall X, Y, Z \in \Gamma \end{aligned}$$

Quindi per ogni combinazione di simboli di nastro dobbiamo avere piastrelle di quel tipo; quindi, stiamo carrozzando il sistema di tiling con tutti i possibili casi che la macchina si potrebbe trovare di fronte mentre computa. Facciamo questo perché non possiamo preventivamente simulare la macchina per stabilire se si ferma o meno, perché la funzione deve produrre un sistema di tiling in tempo finito, quindi dobbiamo fare una trasformazione generica.

- 3) Come facciamo ad essere sicuri che nella prima riga vada a finire proprio la configurazione iniziale? Così:



Dimostriamo che la trasformazione funziona: dobbiamo dimostrare che M è un'istanza sì di $\overline{\text{HALT}_\varepsilon} \Leftrightarrow T$ ottenuto da $T = g(M)$ è un'istanza sì del problema del Tiling, un'istanza è sì se ammette un piastrellamento.

is Si Se M non si arresta su $\varepsilon \Rightarrow T$ ammette un tiling

L'intuizione riguarda il modo specifico in cui abbiamo definito le tipologie di piastrelle.

Posizioniamo in $0,0$ d_0 e riempiamo tutta la prima fila con le piastrelle di 3° tipologia, dopodiché noi possiamo impilare su 2°, 3°, ... fila solo piastrelle che codificano la sequenza di configurazioni di M , perché così abbiamo scelto i colori. Quindi noi avremo una fila aggiuntiva di piastrelle ogni volta la macchina avrà a disposizione uno step aggiuntivo.

Siccome la M non si ferma su input vuoto essa farà infiniti step e quindi potremo mettere infinite file.

Quindi se M non si ferma su $\varepsilon \Rightarrow$ riusciamo a piastrellare il primo quadrante. ■

is No Se M si arresta su $\varepsilon \Rightarrow$ noi non siamo in grado di piastrellare la superficie

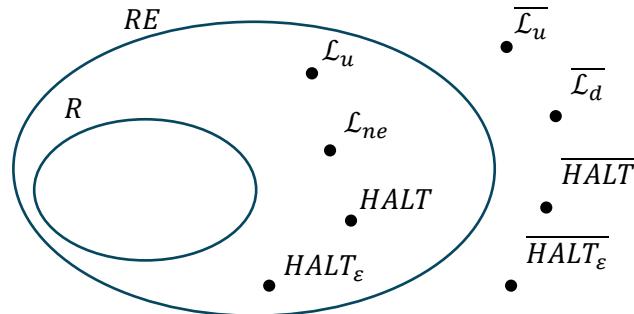
Invece di dimostrare questo dimostriamo il suo converso, dimostriamo che se T ottenuto da $g(M)$ ha prodotto un sistema di tiling che ammette un tiling, allora era perché M non si arresta su ε quando processa.

Rivediamo come sono state definite le piastrelle, esse codificano la sequenza di configurazioni che la macchina attraversa nel momento in cui calcola. Le tipologie di piastrelle che abbiamo definito non ammettono stati terminali; quindi, se siamo riusciti a piastrellare tutto la macchina non è mai passata per uno stato terminale, da cui M non si arresta mentre processa la stringa vuota.

Questo ci dimostra che $\overline{HALT}_\varepsilon$ può essere ricodificato sul problema del Tiling ergo non possiamo sperare che il problema del Tiling sia decidibile perché $\overline{HALT}_\varepsilon$ è indecidibile.

4.3 LA CLASSE CO-RE

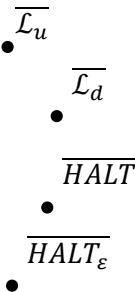
I problemi indecidibili che abbiamo visto finora sono tutti un po' particolare, c'è una certa relazione tra un linguaggio indecidibile e il suo complemento.



$\boxed{\mathcal{L}_u}$

Perché $\mathcal{L}_u \in RE$?

Perché esiste una macchina
in grado di accettarlo



Perché $\overline{\mathcal{L}_u} \notin RE$?

Perché altrimenti entrambi starebbero in R .

L'intuizione: perché $\overline{\mathcal{L}_u}$ sta fuori RE ?

Perché esistono macchine che non arrivano mai ad accettare. Se dobbiamo dire "sì è vero la macchina non si ferma" e noi la simuliamo, noi non ci arriveremo mai a vederlo che la macchina non si è fermata per fermarsi e dirci di sì.

$\boxed{\mathcal{L}_{ne}}$

Perché $\mathcal{L}_{ne} \in RE$?

Noi riusciamo a dire di sì perché indoviniamo una cosa che accetta,
la simuliamo in tempo finito e siamo in grado di dire di sì

Perché $\overline{\mathcal{L}_{ne}} \notin RE$?

Perché per poter rispondere "sì questa macchina non accetta niente" dovrei provare tutte le stringhe e vedere che non si arresta o dice di no, ma mi servirebbe tempo infinito.
Quindi non sono in grado di dire in tempo finito "questa macchina non accetta niente".

\boxed{HALT}

Perché $HALT \in RE$?

Perché basta simulare, se
la macchina si ferma diciamo sì

Perché $\overline{HALT} \notin RE$?

Perché dovremmo poter
dire che si ferma in tempo finito

$\boxed{HALT_\varepsilon}$

Perché $HALT_\varepsilon \in RE$?

Perché noi simuliamo la macchina
su ε , se si ferma diciamo di sì

$HALT_\varepsilon$ dove starà? $HALT_\varepsilon \notin RE$.

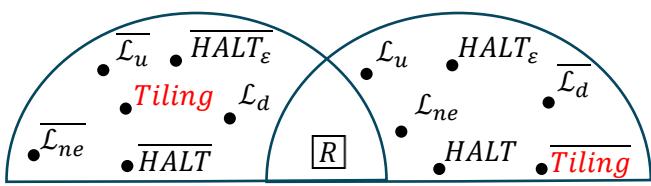
Quindi questa è l'intuizione dei problemi che abbiamo visto finora.

Cioè, i problemi che stanno in RE sono tali per cui noi siamo in grado di rispondere 'sì' in tempo finito. I problemi che abbiamo visto finora (in RE e non in R) hanno il loro complemento fuori RE ; però il complemento che hanno è un linguaggio particolare, tutti questi linguaggi che abbiamo visto che non stanno in RE hanno una peculiarità, sono quei linguaggi per i quali noi siamo in grado di rispondere "no" in tempo finito sono quindi "decidibili a metà"; quindi:

- > Quelli in RE rispondiamo 'sì' in tempo finito ma non diamo garanzie sul 'no'
- > Quelli fuori RE ($CO\text{-}RE$) sono linguaggi per i quali rispondiamo 'no' in tempo finito e non diamo garanzie sul 'sì'

Possiamo quindi definire un'altra classe di decidibilità che va a raccogliere i linguaggi per i quali rispondiamo 'no' in tempo finito.

Dove stanno *Tiling* e \overline{Tiling} ?



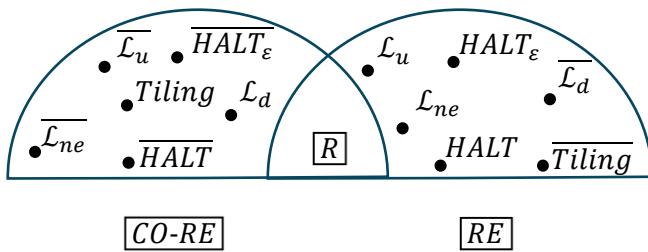
$\boxed{CO\text{-}RE}$

\boxed{RE}

In $CO\text{-}RE$ ci stanno i linguaggi i cui complementi stanno in RE .

La domanda ora è:

Ci sta un problema qui? → 



Ovvero esistono problemi che non stanno né in RE né in $CO-RE$?

Ne andremo ora a vedere uno!

Ma per farlo ci servono un po' di proprietà:

THM: $CO-RE \cap RE = R$

Proof

Per dimostrare questa cosa noi dobbiamo mostrare due cose

1) $CO-RE \cap RE \subseteq R$

Sia \mathcal{L} un linguaggio tale che $\mathcal{L} \in CO-RE \cap RE$, noi mostreremo che \mathcal{L} sta in R

Siccome $\mathcal{L} \in CO-RE$, per definizione di $CO-RE \Rightarrow \overline{\mathcal{L}} \in RE$

Dal fatto che $\mathcal{L} \in RE$ e $\overline{\mathcal{L}} \in RE \Rightarrow \mathcal{L} \in R$

2) $R \subseteq CO-RE \cap RE$

Sia $\mathcal{L} \in R$, se $\mathcal{L} \in R \Rightarrow \overline{\mathcal{L}} \in R$, avrà quindi una macchina che è in grado di dare garanzia di risposta sia su 'sì' che su 'no'. E questo ci dimostra che $\mathcal{L} \in CO-RE \wedge \mathcal{L} \in RE$

THM: $CO-RE \neq RE$

Proof

Supponiamo per assurdo che $CO-RE = RE$, poiché $CO-RE \cap RE = R$, se fosse vero che $CO-RE = RE$ avremmo $RE = R$, ma ciò è impossibile per il teorema di prima!

THM: Siano A, B due linguaggi tali che $A \leq B$. Se $A \notin CO-RE \Rightarrow B \notin CO-RE$

Ora andiamo a caratterizzare un problema che non sta né in RE né in $CO-RE$

Per farlo utilizzeremo delle riduzioni.

$$\text{HALT}_V = \{M \mid \forall w \quad M \text{ si arresta su } w\}$$

È l'insieme delle MdT tale per cui datagli un qualsiasi input questa macchina si ferma.

Perché $\text{HALT}_V \notin RE$?

Perché dovremmo provare ∞ stringhe

Perché $\text{HALT}_V \notin CO-RE$?

Perché la macchina andrebbe avanti all'infinito

Per dimostrare che

- 1) $\text{HALT}_V \notin RE$
- 2) $\text{HALT}_V \notin CO-RE$

Ci servono due riduzioni:

1. Che tolga HALT_V da RE
2. Che tolga HALT_V da $CO-RE$

$$\begin{aligned} &\Rightarrow \overline{\text{HALT}_\epsilon} \leq \text{HALT}_V \\ &\Rightarrow \text{HALT}_\epsilon \leq \text{HALT}_V \end{aligned}$$

Partiamo da 2) perché è più facile. Cioè, da dimostrare che $\text{HALT}_V \notin \text{CO-RE}$ facendo la riduzione:

$$\begin{array}{ccc} \overline{\text{HALT}_\varepsilon} & \leq & \text{HALT}_V \\ M & \xrightarrow{f} & N = f(M) \end{array}$$

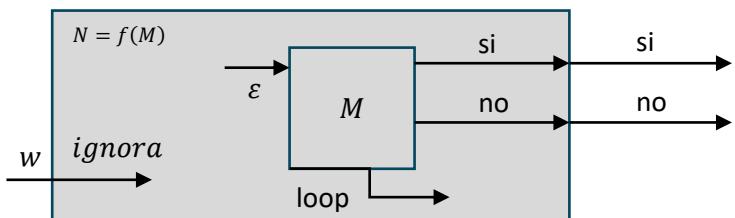
Mostriamo ora che f trasforma istanze 'sì' di HALT_ε in istanze 'sì' di HALT_V e istanze 'no' di HALT_ε in istanze 'no' di HALT_V :

\Rightarrow Supponiamo che $M \in \text{HALT}_\varepsilon$, da cui M si arresta su ε

Siccome N ignora totalmente il proprio input e quello che fa è simulare M su ε allora la macchina N su tutte le stringhe w darà sempre la stessa risposta, ovvero la risposta di M su ε , a noi non importa cosa sia questa risposta, ci importa che N si arresta sempre. Da cui M si arresta su ε .

$$\Rightarrow N \in \text{HALT}_V$$

Ciò dimostra che $\text{HALT}_V \notin \text{CO-RE}$ ■



\Leftarrow Supponiamo che $M \notin \text{HALT}_\varepsilon$, da cui M non si arresta su ε

Siccome N ignora il proprio input e simula M su ε , se M su ε looppa allora N loopperà per tutti i propri input, non si ferma mai. Quindi in questo caso la macchina N non si arresta.

$$\Rightarrow N \notin \text{HALT}_V$$

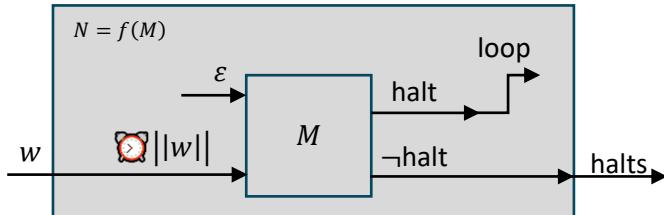
Caso 1)

$$\begin{array}{ccc} \overline{\text{HALT}_\varepsilon} & \leq & \text{HALT}_V \\ M & \xrightarrow{f} & N = f(M) \end{array}$$

Usa la lunghezza di w come un contatore.

Cosa fa N ? Prende w in input e simula M su ε per un numero di passi pari alla lunghezza di w , quindi se $|w| = 10 \Rightarrow N$ simulerà 10 passi di M su ε , se $|w| = 25, N$ simulerà M su ε per 25 passi. Se all'interno di questo *time-bound* la simulazione di M su ε ha raggiunto una configurazione finale (non importa se accettante o meno), cioè, se all'interno di questa simulazione bounded M si è arrestata allora la macchina N deliberatamente looppa, inizia a girare all'infinito.

Se in questa simulazione limitata a un numero di passi (che è la lunghezza di w) M non ha raggiunto una configurazione finale, allora N si ferma; può rispondere sì, può rispondere no, non importa, N si ferma.



\Rightarrow Sup. che $M \in \overline{\text{HALT}_\varepsilon}$, da cui M non si arresta su ε

Se M non si arresta su ε , vuol dire che qualsiasi sia il time-bound sulla simulazione, M su ε in quel numero finito di passi non avrà raggiunto una configurazione finale.

Di conseguenza noi possiamo dare in input a N una stringa w di qualsiasi lunghezza e quindi una qualsiasi stringa, la simulazione di M su ε , siccome M non si ferma su ε non arriverà mai a una configurazione finale. Quindi la macchina N si arresta su qualsiasi stringa. Perché qualsiasi stringa non darà mai un bound sufficientemente ampio da vedere M arrestarsi su ε

$$\text{Segue } N \in \text{HALT}_V$$

\Leftarrow Sup. che $M \notin \overline{\text{HALT}_\varepsilon}$, da cui M si arresta su ε

Se M si arresta su ε lo farà in tempo finto. Diciamo che si arresti in t passi.

Se $|w| \geq t$ la configurazione arriverà alla configurazione finale, e N vedrà M arrestarsi e quindi su quelle stringhe lunghe N looppa.

Se $|w|$ è troppo corta da non permettere a N di vedere che M si arresta su ε allora N si arresterà su quelle stringhe.

Segue che N si arresta su stringhe w con $|w| \leq t$ e M looppa su stringhe w con $|w| \geq t$.

$$\text{Da cui } N \notin \text{HALT}_V$$

5 TEOREMA DI RICE

Tra i vari linguaggi che abbiamo incontrato, abbiamo visto il linguaggio vuoto e il linguaggio non vuoto

$$\begin{aligned}\mathcal{L}_e &= \{M \mid \mathcal{L}(M) = \emptyset\} \\ \mathcal{L}_{ne} &= \{M \mid \mathcal{L}(M) \neq \emptyset\}\end{aligned}$$

Questi sono linguaggi che contengono codifiche di MdT il cui linguaggio ha una certa proprietà, può essere vuoto/non vuoto. Noi abbiamo trovato che questi linguaggi sono *indecidibili*, ma il fatto che lo siano è solo un caso particolare di un risultato, una proprietà, molto più generale che riguarda i linguaggi che hanno questa forma. Cioè, i linguaggi caratterizzati da codifiche di MdT che devono soddisfare una proprietà.

Intuizione: Possiamo quindi definire il concetto di **proprietà di MdT**. Una MdT ha una certa proprietà se ha delle caratteristiche. Per formalizzare questa cosa definiamo formalmente una proprietà di MdT

DEF: Proprietà di MdT

Una proprietà di MdT è un insieme di codifiche di Macchine di Turing.

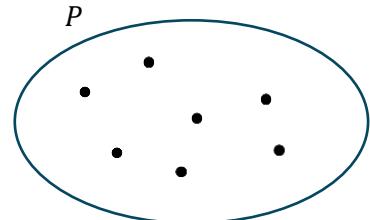
Quindi se abbiamo una proprietà P - che è un insieme – una MdT ha una certa proprietà se $\in P$.

Noi possiamo caratterizzare le proprietà delle MdT in questo modo: associato a questo insieme abbiamo un linguaggio così definito

$$L_P = \{M \mid M \in P\}$$

Dobbiamo però stare attenti che una proprietà può fare riferimento a due categorie di caratteristiche della macchina può essere una proprietà che riguarda:

- **Proprietà di macchine** (*com'è fatta la macchina*)
ie. Quanti stati ha, come computa, ...
- **Proprietà dei linguaggi delle macchine**
(riguardano i linguaggi della macchina)
più astratte



DEF: Proprietà Banale

Una proprietà P è banale se contiene tutte le macchine oppure se non ne contiene nessuna.

❓ Quanto è difficile stabilire se una macchina ha una proprietà banale ❓

È banale (per l'appunto)

Esempi:

$$P_1 = \{M \mid M \text{ ha 5 stati}\}$$

$$P_2 = \left\{ M \mid \begin{array}{l} \mathcal{L}(M) \text{ contiene solo stringhe} \\ \text{di lunghezza pari} \end{array} \right\}$$

$$P_e = \{M \mid \mathcal{L}(M) = \emptyset\} = \mathcal{L}_e$$

Noi P_2 e P_e che riguardano solamente il linguaggio riconosciuto dalla macchina le chiamiamo **proprietà semantiche**. Quindi:

DEF: Proprietà semantiche (di linguaggi)

Una proprietà P di macchine si dice semantica se

$$\begin{aligned}\forall M_1, M_2 : \mathcal{L}(M_1) &= \mathcal{L}(M_2) \\ \text{Se } M_1 \in P &\Rightarrow M_2 \in P\end{aligned}$$

L'appartenenza o meno di macchine a una proprietà semantica P è caratterizzata solamente dal linguaggio che riconosce. Nel momento in cui due macchine distinte riconoscono uno stesso linguaggio esse avranno o no la stessa proprietà semantica.

L'appartenenza a una proprietà semantica di una certa macchina è solamente il suo linguaggio. Per questa ragione le proprietà semantiche le chiamiamo anche *proprietà di linguaggi*.

DEF: Proprietà dei linguaggi RE

Una proprietà P di linguaggi RE è semplicemente un sottoinsieme di RE

$$P \subseteq RE$$



Anche per le proprietà di linguaggi abbiamo le proprietà banali

DEF: Proprietà banali dei linguaggi

Una proprietà P di linguaggi di RE è banale se

$$P = \emptyset \vee P = RE$$

Avevamo visto che \mathcal{L}_e e \mathcal{L}_{ne} sono linguaggi *indecidibili*, ma sono un caso particolare di un risultato molto più generale, ovvero

THM di Rice

Ogni proprietà non banale dei linguaggi RE è indecidibile.

Se P è un insieme di linguaggi noi non possiamo inserire linguaggi dentro P , perché non abbiamo un modo per rappresentarli. Quello che possiamo fare è identificare un linguaggio in maniera compatta tramite la MdT che lo riconosce, ed ecco la connessione tra proprietà di linguaggi e proprietà semantiche.

$$L_P = \{M \mid \mathcal{L}(M) \in P\}$$

Abbiamo quindi il collegamento perché le proprietà semantiche sono anch'esse insiemi di macchine.

Proof:

Sia P una proprietà non banale di RE abbiamo due casi possibili

- a) $\emptyset \notin P$
- b) $\emptyset \in P$

Partiamo da a)

Se $P \subseteq RE \Rightarrow P \neq \emptyset$ e siccome $\emptyset \notin P$, allora dentro P dovrà starci un linguaggio $L \neq \emptyset$, ovvero

$$\exists L \in RE : L \neq \emptyset \wedge L \in P$$

Se $L \in RE$ e $L \neq \emptyset \Rightarrow \exists$ una macchina M_L che accetta L (questo perché $L \in RE$)

Dimostriamo l'indecidibilità del linguaggio $L_P = \{M \mid \mathcal{L}(M) \in P\}$ tramite una riduzione

$$\begin{array}{ccc} L_u & \leq & L_P \\ (M, w) & \xrightarrow[f]{} & N = f(M)' \end{array}$$

N dev'essere una macchina il cui linguaggio ha la proprietà $P \Leftrightarrow M \models w$

Dobbiamo dimostrare che questa è $L_u \leq L_P$:

\Rightarrow Sup. che $(M, w) \in L_u$, da cui $M \models w$

Notiamo che M viene eseguita sempre su w

Segue che $\mathcal{L}(N) = L \Rightarrow N \in L_P$

\Leftarrow Sup. che $(M, w) \notin L_u$, da cui $M \not\models w$

La seconda fase non parte, quindi N si blocca

Segue che $\mathcal{L}(N) = \emptyset \Rightarrow N \notin L_P$

Ciò dimostra che la riduzione è corretta e trasforma istanze 'sì' di L_u in istanze 'sì' di L_P e \equiv per ist. 'no'. Quindi il linguaggio L_P è indecidibile $\therefore L_u$ è indecidibile.

Consideriamo il caso b) Abbiamo che $\emptyset \in P$

Consideriamo $\overline{P} = RE \setminus P$ Se $\emptyset \in P$ allora $\emptyset \in \overline{P}$? No. Quindi $\emptyset \notin \overline{P}$

Se P è non banale, per assunzione, \Rightarrow possiamo dire che \overline{P} è anch'esso non banale.

Quindi per il caso a) $L_{\overline{P}}$ è *indecidibile*

Siccome stiamo considerando proprietà di linguaggi in RE , tutti i linguaggi o stanno in \overline{P} o ne stanno fuori.

E se stanno fuori \overline{P} stanno in P quindi vale che

$$\overline{L_{\overline{P}}} = L_P$$

Se L_P fosse decidibile \Rightarrow lo sarebbe anche $\overline{L_{\overline{P}}}$.

Ma se $\overline{L_{\overline{P}}}$ fosse decidibile, quindi $\in R$, lo sarebbe anche $L_{\overline{P}}$, ma ciò non è possibile. Da cui L_P è indecidibile.

Abbiamo questo risultato molto generale, che ci dice che ogni qualvolta che noi siamo davanti a un linguaggio che si configura come il linguaggio di una proprietà non banale, per questa ragione il linguaggio è indecidibile; cioè, decidere proprietà non banali di linguaggi è una cosa che non possiamo mai fare. Siamo capaci di rispondere solo alle cose banali, solo alle proprietà a cui diciamo sempre di sì o sempre di no, solo a quelle siamo in grado di rispondere in maniera automatica. Per tutte le altre proprietà che siano non banali noi abbiamo che è impossibile trovare algoritmi che decidano quelle proprietà. E questa cosa noi la possiamo sfruttare per vedere varie indecidibilità.

Un primo esempio è:

$$\begin{array}{ll} L_e & L_{ne} \\ L_e = \{M | \mathcal{L}(M) = \emptyset\} \end{array}$$

Siccome L_e è un linguaggio che contiene *codifiche di MdT*, L_e si configura come proprietà di macchine. Questa è una proprietà di macchine che riguarda i linguaggi delle macchine; quindi, è una proprietà semantica o una proprietà di linguaggi.

Se volessimo dimostrare che L_e è una proprietà **semantica** di macchine, come si potrebbe fare?

Consideriamo due macchine M_1 e M_2 tali che $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, se $M_1 \in L_e$ è perché $\mathcal{L}(M_1) = \emptyset$, ma allora se $\mathcal{L}(M_2) = \mathcal{L}(M_1) \Rightarrow M_2 \in L_e$ da cui L_e è una proprietà semantica.

Ora ci chiediamo:

❓ L_e è una proprietà banale ?

L_e è non banale se: $L_e \neq \emptyset \vee L_e \neq RE$

L_e è non banale perché non è vuota, contiene il linguaggio vuoto. $L_e = \{\emptyset\}$
e poi anche tutti i linguaggi diversi dal linguaggio vuoto $\notin L_e$.

Quindi L_e è indecidibile.

$$L_{ne} = \{M | \mathcal{L}(M) \neq \emptyset\}$$

Che cosa possiamo dire di L_{ne} ? Possiamo dire sicuramente che è una proprietà di macchine, perché è un linguaggio che contiene codifiche di macchine. Vediamo poi se la proprietà è semantica o meno; il Teorema di Rice si applica solamente se è una proprietà semantica. Se la proprietà non è semantica ma riguarda una caratteristica della macchina, Rice non dice niente e quella proprietà può essere decidibile o indecidibile.

Prendiamo ora P_2

$$P_2 = \{M \mid \mathcal{L}(M) \text{ contiene solo stringhe di lunghezza pari}\}$$

?

È una proprietà di macchine ?

Sì, perché contiene codifiche di macchine.

?

Questa proprietà è banale o non banale ?

Non banale, perché dentro $P_2 = \{\emptyset, \{00\}, \dots\}$

?

È una proprietà di linguaggi o semantica ?

Sì, perché è relativa solamente al linguaggio che la macchina riconosce. Abbiamo quindi che P_2 è una proprietà di linguaggi.

?

Questa proprietà contiene tutto RE ?

No, perché i linguaggi con stringhe dispari $\notin P_2$

Quindi P_2 è una proprietà di linguaggi non banale, e per questa ragione, per il teorema di Rice, decidere è indecidibile.

Prendiamo invece adesso P_1

$$P_1 = \{M \mid M \text{ ha 5 stati}\}$$

È una proprietà di macchine; non semantica (lo si capisce dalla definizione) se volessimo dimostrarlo formalmente dovremmo trovare un controesempio, cioè due macchine M_1 e M_2 che abbiano lo stesso linguaggio, una $M_1 \in P_1$ e l'altra $M_2 \notin P_1$. Prendiamo quindi una macchina M_1 a cinque stati che riconosce un certo linguaggio, da M_1 generiamo a una macchina M_2 in cui aggiungiamo un sesto stato, che non fa niente, è uno stato irraggiungibile che non è attaccato a nulla. M_2 quindi, è una macchina che ha lo stesso linguaggio di M_1 ma il suo numero di stati è differente. Avremo quindi che

$$\begin{cases} M_1 \in P_1 \\ M_2 \notin P_1 \end{cases}$$

Di conseguenza la proprietà è non semantica, cioè, è relativa a com'è fatta la macchina più che al linguaggio che le macchine riconoscono.

5.1.1 Esempi di applicazione

a) Data una macchina M decidere se $\mathcal{L}(M)$ è finito (se ha un numero finito di stringhe)

$$L_a = \{M \mid \mathcal{L}(M) \text{ è finito}\}$$

È una proprietà di macchina

È una proprietà semantica

È una proprietà non banale ($\emptyset \in L_a \wedge (a^n \mid n > 0) \notin L_a$)

Quindi per il Teorema di Rice L_a è indecidibile.

b) Data una macchina M decidere se $\mathcal{L}(M)$ è infinito

È una proprietà di macchina

È una proprietà semantica

È una proprietà non banale

Quindi L_b , per il Teorema di Rice è indecidibile

c) Data una macchina M decidere se $\mathcal{L}(M)$ è un linguaggio accettato solo da macchine con 5 stati.

È una proprietà di macchina
È una proprietà semantica
È una proprietà banale

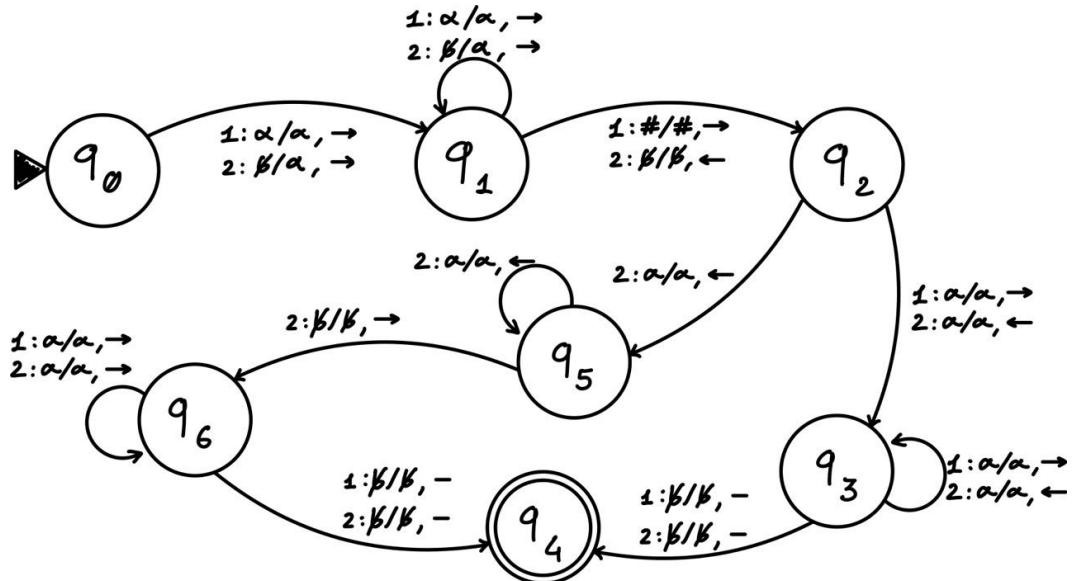
Ogni linguaggio può essere riconosciuto da MdT con quanti stati si vogliono. Nel momento in cui abbiamo una macchina che riconosca un linguaggio, noi la possiamo inflazionare di stati inutili e quindi lo stesso linguaggio viene riconosciuto da MdT con un altro numero di stati.

Di conseguenza, i linguaggi che vengono riconosciuti solamente da MdT con cinque stati non ci stanno. Perché nessun linguaggio che sia riconosciuto da un MdT con cinque stati non può essere riconosciuto da una MdT con quindici.

Quindi in realtà $L_c = \{\emptyset\}$ e per questa ragione è *decidibile*.

Supponiamo di avere questo linguaggio:

$$L = \{W\#A \mid W \in (0,1)^+, A = W \vee A = W^R\}$$



$$P_1 = \{M \mid \mathcal{L}(M) = L\}$$

Questo è il problema di decidere se una macchina ha come linguaggio un linguaggio fissato

Questo problema è decidibile o indecidibile?

È indecidibile, perché è una proprietà di macchine semantiche e non banale perché dentro ha L e fuori ha le cose $\neq L$. Quindi non è né vuota ne ha tutto RE.

$$P_2 = \left\{ M \mid \begin{array}{l} \mathcal{L}(M) = L \wedge \\ \text{Ogni stringa di } L, M \text{ la accetta in al più 100 passi} \end{array} \right\}$$

Questo problema è decidibile o indecidibile?

È decidibile, è una proprietà di macchine, ma è una proprietà banale perché $P_2 = \{\emptyset\}$ non contiene nessuna macchina perché non c'è nessuna macchina che ci dia garanzia di riconoscere L in cento passi. Sulle stringhe w lunghe 200 quella macchina non ha nemmeno il tempo di leggere w . Quindi la P è vuota.

$$P_3 = \{M \mid M \text{ non accetta } L \cap \{0,1,\#\}^{100}\}$$

Ovvero in P_3 ci stanno quelle MdT che non accettano il linguaggio delle stringhe che stanno in L e sono lunghe cento caratteri.

Questo problema è decidibile o indecidibile?

È indecidibile, è una proprietà di macchine, è non banale perché qualche macchina ci sta in P_3 e qualcun'altra non ci sta, è semantiche perché se siano M_1 e M_2 due macchine che abbiano lo stesso linguaggio, supponiamo che $M_1 \in P_3$, allora $\mathcal{L}(M_1) \neq L \cap \{0,1,\#\}^{100}$, ma M_2 avrà anch'essa un linguaggio diverso, quindi anche $M_2 \notin P_3$.

6 COMPLESSITÀ STRUTTURALE

Finora abbiamo visto le classi RE e R , d'ora in poi ci andremo a concentrare solamente sui problemi $\in R$. Ovvero dei problemi ricorsivi, dei problemi decidibili.

C'è tutta un'area di ricerca dell'informatica teorica che si focalizza all'interno della classe R , cioè stabilirne la complessità. La questione è meramente di tipo storico perché, quando questa teoria nacque, negli anni 30 con Turing, le persone era interessate dal capire se un problema fosse decidibile o meno e tutto era molto astratto, non esistevano macchine reali di calcolo sofisticate, solo roba rudimentale non programmabili. E quello che si chiedevano era *"se mai fossi in grado di costruire questa macchina, che genere di problemi sarebbe in grado di risolvere?"* e di fatto, i primi risultati di indecidibilità ($\approx 1935/36$) riguardavano una macchina astratta. Cioè, questa macchina non esisteva ma si sapeva già che, se mai fossimo stati in grado di costruirla, ci sarebbero stati problemi che quella macchina non sarebbe stata in grado di risolvere. Quindi la gente si buttò sulla decidibilità dei problemi.

Quindi lo studio della complessità strutturale dei problemi che adesso definiremo nasce da una questione meramente tecnologica, cioè in quel momento avevamo a disposizione delle macchine reali e la gente si è iniziata a chiedere *"questo problema in quanto lo vedrò risolto?"*. Questa domanda nasce dal fatto che la macchina esisteva e la gente si trovava lì ad aspettare la risposta, quando non c'erano le macchine non era una questione che veniva presa in considerazione, le persone si chiedevano *"è in grado di farlo o no?"*. Quando iniziarono a programmare i primi computer si resero conto che c'erano problemi molto più difficili di altri. Cioè, problemi che richiedevano – con gli algoritmi di cui disponevano – molto più tempo di altri problemi. A quel punto la questione che nacque è *ma l'algoritmo che abbiamo per risolvere questo problema è lento perché siamo noi che non siamo furbi a sufficienza per tirarne fuori uno veloce? O il problema è talmente complicato che noi non riusciremmo mai ad avere un algoritmo veloce?* E quindi questa cosa venne formalizzata dopo un bel po' di tempo (Hartmanis, Stearns, Lewis 1965). Introdussero il concetto di **complessità temporale** e **complessità spaziale** delle macchine di Turing che adesso definiremo in maniera più precisa, con questa nozione:

La complessità temporale di una MdT è legata al numero di passi che la macchina fa prima di fermarsi e dare una risposta

In maniera similare, sempre in quegli anni introdussero il concetto di complessità spaziale:

Ma quante celle deve la macchina deve leggere o scrivere prima di dare la risposta?

Perché io le celle le posso sovrascrivere, quindi magari sovrascrivendole in maniera opportuna posso mantenere il numero di celle usate più piccolo, e questo andava a stimare la quantità di memoria che era necessaria in un computer per poter risolvere un certo problema.

Questo studio si chiama **"complessità strutturale"** perché si vuole occupare della complessità, della struttura dei problemi, cioè:

Com'è che la struttura di un problema influenza o meno la complessità degli algoritmi che lo risolvono?

L'oggetto qua è **la complessità del problema** a differenza della complessità degli algoritmi trattata nel corso di Algoritmi e Strutture di Dati. Noi ci interessiamo, vogliamo capire:

"Ma la struttura di questo problema come impatta la complessità degli algoritmi che lo risolvono?"

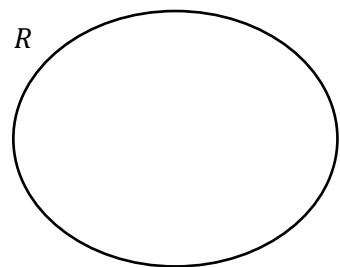
Questo problema è tosto di suo tale per cui tutti gli algoritmi che lo risolvono prenderanno sempre tempo esponenziale?

Questa è la domanda che ci porremo da qui in po'

Dato un problema stabilire la famiglia di problemi di complessità simile a cui appartiene

6.1.1 Classe DTime

Ci focalizzeremo sull'insieme dei problemi decidibili. L'obiettivo è iniziare ad arricchire la struttura di questo insieme andando a identificare all'interno di R classi di linguaggi di difficoltà differente, troveremo classi di problemi facili, classi di problemi difficili o, equivalentemente, classi di linguaggi facili e classi di linguaggi difficili. Perché non sospendiamo l'assunzione che abbiamo fatto, cioè che studiamo i problemi tramite i linguaggi.



Definiamo la complessità temporale delle MdT o degli algoritmi

DEF: (Computation time)

Sia M una macchina di Turing e w una stringa in input per M . Il *computation time* di M su w è il numero di passi che M esegue prima di arrestarsi su w .

Se M è non deterministica, il computation time è dato dalla lunghezza del branch di computazione più lungo.

DEF: (time function)

Sia $t(n)$ una funzione $t: \mathbb{N} \rightarrow \mathbb{N}$ tale che $t(n)$ è non decrescente e strettamente positiva, questa funzione la chiamiamo *time function*.

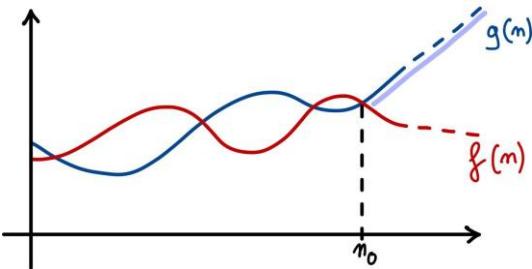
DEF: (running time)

Sia $t(n)$ una time function, la MdT M ha running time $t(n)$ se per tutte le stringhe w , a parte un numero finito, il computation time di M su w non eccede $t(\|w\|)$

DEF: (notazione asintotica)

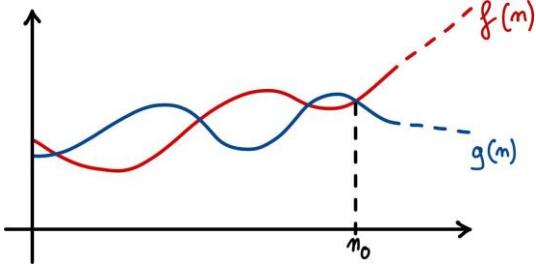
> BIG O , una funzione $f(n) \in O(g(n))$ per un'altra funzione $g(n)$ se:

$$\exists c, n_0, \quad : \forall n \geq n_0, \quad f(n) \leq c \cdot g(n)$$



> BIG Ω , una funzione $f(n) \in \Omega(g(n))$ per un'altra funzione $g(n)$ se:

$$\exists c, n_0, \quad : \forall n \geq n_0, \quad f(n) \geq c \cdot g(n)$$



> BIG Θ , una funzione $f(n) \in \Theta(g(n))$ se $f(n)$ è sia $O(g(n))$ che $\Omega(g(n))$

Perché introduciamo questo? Perché attraverso questo introduciamo il concetto di complessità di problemi

DEF: (time complexity upper bound)

Il time complexity upper bound di un problema P è $O(f(n))$ (dove $f(n)$ è una time function) se tutti gli algoritmi o tutte le MdT che risolvono P hanno un running time che è $O(f(n))$.

Quindi noi diciamo che il time complexity upper bound di un problema è $O(f(n))$ se tutti gli algoritmi che lo risolvono hanno un running time che è $O(f(n))$.

Diciamo che il time complexity *lower bound* di un problema è $\Omega(f(n))$ se tutti gli algoritmi che lo risolvono hanno un running time che è $\Omega(f(n))$.

DEF: (time complexity *lower bound*)

Il time complexity *lower bound* di un problema P è $\Omega(f(n))$ (dove $f(n)$ è una time function) se tutti gli algoritmi o tutte le MdT che risolvono P hanno un running time che è $\Omega(f(n))$.

⇒ la complessità temporale di un problema è legata alla complessità temp. degli algoritmi che lo risolvono.

Andiamo ora a definire all'interno della classe R delle classi di complessità. Noi abbiamo detto che R è una classe di decidibilità, o di calcolabilità, dentro R ci sta tutto quello che è calcolabile. Adesso il nostro intento è andare a identificare dentro R ciò che è a bassa o alta complessità.

DEF: (Classe di complessità *temporale*)

Sia $t(n)$ una time function, denotiamo $DTIME(t(n)) = \left\{ L \mid \exists M \text{ deterministica che } \underline{\text{decide}} \text{ in tempo } O(t(n)) \right\}$

> Definiamo formalmente la classe P (Polynomial Time)

$$P = \bigcup_{c \geq 1} DTIME(n^c)$$

? Ordinare un array sta in P ?

No, Perché non è un problema di decisione.

? Sommare due numeri sta in P ?

No, Perché non è un problema di decisione.

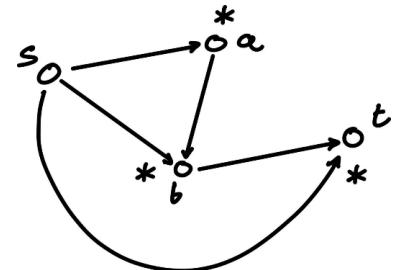
6.1.2 Classe NTime

> Reachability

Dato un grafo diretto, una sorgente e una destinazione, vogliamo stabilire se esiste un percorso dal nodo sorgente al nodo destinazione.

$$Reach = \left\{ \langle G, s, t \rangle \mid \begin{array}{l} G \text{ è un grafo diretto,} \\ s, t \text{ sono nodi } \in G, \\ \exists \text{ un percorso da } s \text{ a } t \in G \end{array} \right\}$$

? $Reach \in P$?



Si, c'è l'algoritmo di Dijkstra

> Primes

L'insieme degli interi n , rappresentati in binario, tali che n è un numero primo.

$$Primes = \{n \mid n \text{ è un numero primo}\}$$

? Qual è un algoritmo naïve per risolvere questo problema ?

Preso n dividerlo per $2, 3, 4, 5, \dots, n-1$. Facciamo $\approx O(n)$ divisioni, ogni divisione ci costa polinomiale.

? Questa complessità è polinomiale ?

L'RT è valutato sulla taglia dell'input, che è esponenziale sulla sua taglia. Questo algoritmo è exp, perché il numero di divisioni che facciamo è pari al valore del numero in input.

$$n \approx \log_2 \|n\|, \quad O(2^{\|n\|} \cdot \|n\|^k)$$

Detto ciò, in realtà $Primes \in P$ dagli inizi del 2000, e (a detta del prof) la complessità è $O(n^6)$

> Sat

Soddisfacibilità di formule booleane

Noi ci interesseremo di formule in Forma Normale Congiuntiva (CNF), sono formule booleane che hanno una forma particolare, sono formule che sono date dalla congiunzione di un numero finito di clausole

$$\mathcal{C} = c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

Ognuna di queste clausole c_i è di questa forma qui:

$$c_i = l_1 \vee l_2 \vee \cdots \vee l_m$$

Cioè, una clausola è la disgiunzione di uno o più letterali, ovvero una variabile booleana o il suo negato

$$l_j = x_k \text{ oppure } \neg x_k$$

Quindi una formula CNF, ad esempio, può essere:

$$\mathcal{C} = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_5 \vee \neg x_6)$$

Come si vede: ha una struttura particolare, ha le clausole che sono in congiunzione tra di loro e ogni clausola è una disgiunzione di letterali, dove un letterale è o una variabile booleana oppure il suo negato.

Il problema della Soddisfacibilità è questo: è l'insieme delle formule φ tali che φ è in CNF e φ è soddisfacibile

$$Sat = \left\{ \varphi \mid \begin{array}{l} \varphi \text{ è in CNF} \\ \varphi \text{ è soddisfacibile} \end{array} \right\}$$

❓ Quand'è che una formula è soddisfacibile ?

Quando esiste un assegnamento di verità per le sue variabili che rende la formula vera.

Quindi, in questo caso, se noi diamo: $x_1 = V, x_2 = F, x_5 = V$ soddisfacciamo la formula φ .

Quindi questo è il problema della Soddisfacibilità di una formula booleana di questa forma.

Supponiamo che qualcuno ci dia un assegnamento di verità per le variabili,

❓ Quanto tempo ci mettiamo a stabilire se quell'assegnamento soddisfa o meno la formula ?

Lineare o polinomiale.

Qualcuno ci da un assegnamento, nel momento in cui abbiamo un assegnamento in mano, in tempo polinomiale noi siamo in grado di stabilire se la formula è soddisfatta o meno, perché dobbiamo andare a guardare tutte le clausole e vedere se almeno uno dei suoi letterali è soddisfatto dall'assegnamento.

Poiché la formula ha quella forma molto particolare; noi, clausola per clausola, andiamo a vedere se uno dei letterali viene valutato vero dall'assegnamento che ci viene dato.

❓ Quanti sono i possibili assegnamenti booleani per le nostre variabili ?

Sono 2^n

❓ Testare tutti i possibili assegnamenti booleani per vedere se esiste uno che soddisfa la formula, quanto ci costa ?

Ci costa qualcosa del tipo $O(2^n \cdot \text{poly}(\|\varphi\|))$

Quindi questo algoritmo che testa se una formula è soddisfacibile o meno ci impiega tempo esponenziale.

Da questo noi non siamo in grado di concludere che $Sat \in P$, non lo possiamo escludere, semplicemente questo algoritmo non lo colloca in P .

> Indipendent Set

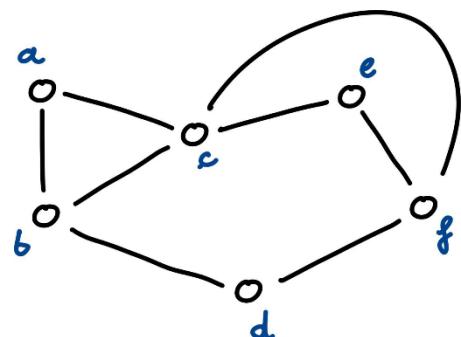
Dato un grafo, un independent set di un grafo è un insieme di nodi che non sono agganciati in alcun modo.

$$S = \{a, e, d\}$$

È un independent set perché fra nessuna delle coppie dei nodi di S è collegato.

? Dato un grafo esiste un indipendent set ?

Si, ogni nodo singolo



La ricerca di IS piccoli è un problema estremamente facile, e che non ci interessa, Diamo quindi una definizione differente dell' IS :

$$IS = \left\{ \langle G, k \rangle \mid \begin{array}{l} \exists \text{ un } IS \text{ di} \\ \text{taglia } k \in G \end{array} \right\}$$

Questo è il problema dell'Independent Set, che ci permette di ottenere un problema di decisione.

Supponiamo che qualcuno ci dia un insieme S ;

? Quanto tempo ci mettiamo a stabilire che quell'insieme S è o no un IS ?

Tempo polinomiale, per ogni nodo dobbiamo vedere se in S ci sta roba attaccata a quel nodo.

Lo facciamo per tutte le coppie, che sono quadratiche, quindi in tempo quadratico sappiamo farlo.

? Ma quanti sono i possibili indipendent set di taglia k ?

$$\binom{n}{k} \cdot O(n^2)$$

Sono $\binom{n}{k}$, quindi dovremo generare $\binom{n}{k}$
e poi avere un costo quadratico

Quindi questo algoritmo è esponenziale

Però questi problemi, Sat e IS , abbiamo visto che hanno algoritmi che girano in tempo esponenziale perché stiamo escludendo di utilizzare macchine non deterministiche, se avessimo macchine ND noi potremmo andare molto più velocemente, perché qual è l'intuizione, per esempio su IS :

Una Macchina ND invece di provare tutti i possibili sottoinsiemi di taglia k , per vedere se c'è un IS di taglia k quello che può fare è *indovinare* un IS di taglia k se c'è.

Praticamente il lavoro che può fare la Macchina ND, ipotizzando che questa sia la configurazione iniziale ID_0 Su IS che deve fare la macchina?

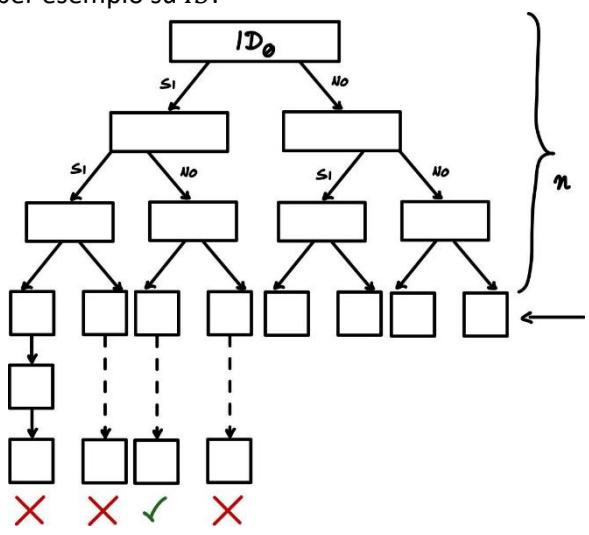
Preso ogni nodo deve stabilire se prenderlo o meno S/N. Poi avremo un'altra cosa così in cui decido se prendere o meno un nodo. E lo possiamo fare per tutti gli n nodi, per poi arrivare a delle configurazioni in cui la macchina ha deciso, per tutti i nodi, se li prende o meno (dove c'è la \leftarrow a destra). Dopodiché la macchina avrà scritto su nastro, per ogni nodo, se lo prende o meno, dopodiché c'è tutta la fase di check, che è deterministica, in cui la macchina che fa?

1. Verifica che il numero dei nodi scelti sia $\leq k$
2. Verifica che quello che abbiamo scelto forma un IS .

? Quanto tempo ci mette la macchina?

Tempo polinomiale; lineare a fare il guess e quadratico a fare il check.

Similmente per Sat ; una macchina non deterministica può fare una cosa simile, indovina in tempo lineare un assegnamento di verità per le variabili e dopodiché verifica che quello che ha indovinato effettivamente soddisfi la formula. Quanto tempo ci mettiamo? Polinomiale.



Questi problemi sono interessanti perché ci identificano una classe di complessità interessante, che è la classe dei problemi risolvibili in tempo polinomiale non deterministico.

DEF:

Sia $t(n)$ una time function, la classe di complessità $NTime(t(n)) = \{L \mid \exists M \text{ Non deterministica che decide } L \text{ in tempo } O(t(n))\}$

> Definiamo formalmente la classe NP (Non deterministic Polynomial Time)

$$P = \bigcup_{c \geq 1} NTime(n^c)$$

Quindi $NTime$ (a differenza di $DTime$) è l'insieme dei linguaggi che possono essere decisi da MdT *non deterministiche* in tempo $O(t(n))$

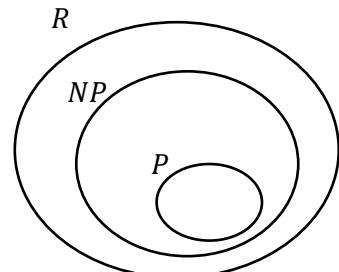
6.2 RIDUZIONI POLINOMIALI

Per come è intuitibile, noi abbiamo che $P \subseteq NP$.

Perché tutto quello che può essere fatto in tempo polinomiale deterministico può essere anche fatto in tempo non deterministico polinomiale.

La relazione opposta, cioè se $NP \subseteq P$, da cui seguirebbe $P = NP$; è un problema correntemente aperto. Riguarda, una cosa che abbiamo già più o meno affrontato quando abbiamo visto la simulazione di Macchine non deterministiche da parte di Macchine deterministiche.

Noi abbiamo che il costo della simulazione, in quel caso, è di tipo esponenziale e qui il problema è lo stesso, noi abbiamo che i linguaggi di NP sono decidibili da MdTN in tempo polinomiale, se noi le simulassimo nel modo in cui abbiamo visto noi avremmo una macchina deterministica in grado di decidere lo stesso linguaggio in tempo esponenziale. Quindi da un punto di vista della decidibilità del linguaggio, il linguaggio sarebbe comunque decidibile, però l'overhead di tipo esponenziale non ci permette di collocare il linguaggio dentro la classe P . Ma ciò non significa che un linguaggio di NP sicuramente non sta in P , perché nessuno ha dimostrato che non sia possibile risolvere in tempo polinomiale deterministico un problema NP "tosto".


DEF: (riduzione polinomiale)

Siano A e B due linguaggi, una riduzione polinomiale da A a B è una funzione $f: \Sigma^* \rightarrow \Sigma^*$ tale che f è calcolabile in tempo polinomiale e $\forall w, w \in A \Leftrightarrow f(w) \in B$

Denotiamo la riduzione con

$$A \leq_P B$$

È la stessa cosa di una riduzione, stiamo aggiungendo il vincolo che la trasformazione debba essere calcolata in tempo polinomiale, ricordiamo che il concetto di calcolabilità l'avevamo collegato alla nozione di *Trasduttore*, quindi di MdT che sono in grado di calcolare un certo risultato. Adesso stiamo dicendo che il numero di passi di quel trasduttore che implementa f deve essere polinomiale nella taglia dell'input.

DEF: (NP-Hardness)

Un linguaggio L è *NP-Hard* se $\forall L' \in NP$ è possibile $L' \leq_P L$

Intuitivamente questa cosa significa che L è almeno difficile quanto tutti i linguaggi di NP .

DEF: (NP-Completeness)

Un linguaggio L è

NP-completo se:

- 1) $L \in NP$
- 2) L è *NP-Hard*

Qual è la differenza tra un linguaggio *NP-Hard* e uno *NP-completo*?

Per essere un linguaggio *NP-completo*, **non solo** dev'essere *NP-Hard* ma deve anche appartenere a NP

Di conseguenza un linguaggio *NP-Hard* è un linguaggio che è **almeno difficile** quanto tutti i problemi di NP

Un Linguaggio *NP-completo* è almeno difficile quanto tutti i linguaggi di *NP* e inoltre è un linguaggio di *NP*

Quindi un linguaggio *NP* completo è un linguaggio tra i più difficili di *NP*.

Per esempio, prendiamo il linguaggio universale $L_u = \{M, w \mid M \models w\}$, è vero che L_u è *NP-Hard*?

Sì, perché è almeno tanto quanto tutti i linguaggi di *NP*.

Non è *NP-completo*, perché $L_u \notin R$.

THM:

Sia L un linguaggio *NP-completo*. Allora se

$$L \in P \Leftrightarrow P = NP$$

Proof:

$$\Leftarrow P = NP \Rightarrow L \in P$$

Siccome L è *NP-completo* $\Rightarrow L \in NP$, di conseguenza, se $P = NP$ avremmo che $L \in P$

$$\Rightarrow L \in P \Rightarrow P = NP$$

Sappiamo che $P \subseteq NP$. Per dimostrare che $P = NP$ ci manca dimostrare $NP \subseteq P$.

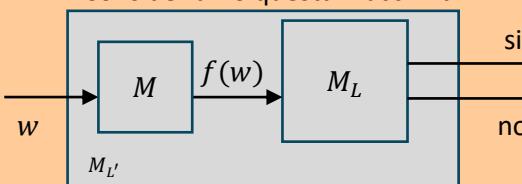
Se riusciamo a mostrare che il fatto che $L \in P \Rightarrow NP \subseteq P$ allora abbiamo fatto.

- Siccome L è *NP-Completo* $\Rightarrow L$ è *NP-Hard*

Se L è *NP-Hard* $\Rightarrow \forall L' \in NP \quad L' \leq_P L$ Quindi c'è una funzione di trasformazione f

Calcolabile in tempo polinomiale deterministico che mappa istanze 'sì' di L' su istanze 'sì' di L e istanze 'no' di L' su istanze 'no' di L .

Consideriamo questa macchina:



che prende in input w , la trasforma secondo f per ottenere $f(w)$, dopodiché viene mandata in input alla macchina che decide il linguaggio L , se M_L risponde 'sì' rispondiamo 'sì' se M_L risponde 'no' rispondiamo 'no'.
Che linguaggio sta decidendo $M_{L'}$? L'

Ma $M_{L'}$ esegue in tempo polinomiale o no?

$$n = \|w\|$$

- 1) $w \rightarrow f(w) = y$

$$c \geq 1, \text{ fixed}$$

Quant'è la taglia di y ? n^c

Quindi la trasformazione di f su w è

$$O(n^c)$$

- 2) Qual è il running time di M_L ? È polinomiale, perché assumiamo che $L \in P$

$$O(n^d)$$

Cosa riceve in input M_L ? $f(w)$ che è grande n^c , quindi rispetto alla taglia iniziale di w , in quanto
Tempo runna la macchina M_L ?

$$O(n^{c^d}) = O(n^{c \cdot d})$$

Quindi il running time totale di questa macchina è

$$O(n^c + n^{c \cdot d})$$

Quindi abbiamo due cose, $M_{L'}$ è una macchina che decide L' e lo fa in tempo polinomiale

Quindi tramite la riduzione $L' \leq_P L$ noi siamo in grado di risolvere L' in tempo polinomiale

Ma L' era un linguaggio generico di *NP*, ciò vuol dire che qualsiasi sia un linguaggio di *NP*, noi tramite L siamo in grado di deciderlo in tempo polinomiale, da cui $NP \subseteq P$.

Combinato a quello che già noi sappiamo che $P \subseteq NP$, otteniamo che $P = NP$.



Sat è NP-completo

Esiste una dimostrazione, sulla quale non ci vogliamo soffermare per il momento, del fatto che *Sat* sia *NP-completo*, di fatto è stato il primo linguaggio definito *NP-completo*, la vedremo più avanti.

Noi abbiamo una *sorgente NP-completa*, noi sappiamo che esiste un linguaggio *NP-completo*, la cosa interessante è che una volta che ne abbiamo uno, mostrare che altri linguaggi sono *NP-Hard* è più semplice, perché possiamo sfruttare le riduzioni.

Esiste una riduzione molto sofisticata, abbastanza generale che fa vedere che qualsiasi linguaggio di *NP* noi consideriamo è possibile ridurlo a *Sat*.

Una volta che abbiamo *Sat* è possibile dimostrare l'*NP-Hardness* di altri problemi tramite altre vie, tramite riduzioni. L'intuizione è che siccome una riduzione sostanzialmente ci mostra che se noi riduciamo $A \leq B$, allora B sarà almeno difficile quanto $A \Rightarrow$ se il problema di partenza sarà *NP-Hard* \Rightarrow il problema di arrivo sarà *NP-Hard*.

THM: Transitività delle riduzioni polinomiali

Siano A, B, C tre linguaggi, se:

$$A \leq_P B \wedge B \leq_P C \Rightarrow A \leq_P C$$

Proof:

$$A \leq_P B \Rightarrow \exists f: A \xrightarrow{f} B \quad \text{il tempo di trasformazione sia } O(n^c) \quad c \geq 1, \text{ fixed}$$

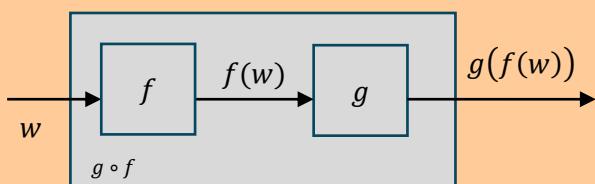
$$B \leq_P C \Rightarrow \exists g: B \xrightarrow{g} C \quad \text{il tempo di trasformazione sia } O(n^d) \quad d \geq 1, \text{ fixed}$$

Vogliamo mostrare che $A \leq_P C$ e, da definizione, abbiamo che ciò accade $\Leftrightarrow \exists$ una riduzione in tempo polinomiale da A verso C . Vogliamo mostrarla:

La riduzione da A a C è la composizione $g \circ f$

$$\Rightarrow A \leq_P C \quad \text{si ottiene applicando } g(f(w))$$

La infiliamo in un nostro trasduttore:



Quello che ci manca da verificare è se questa composizione è calcolabile in tempo polinomiale;

- focalizziamoci sul calcolo di $f(w)$ può essere fatto in $O(n^c)$, per assunzione quant'è grande la stringa sputata fuori dal primo pezzo della formula? $\|f(w)\| = O(n^c)$

- $g(f(w))$, noi sappiamo che g può essere calcolata in $O(n^d)$, quindi sarà $O(\|f(w)\|^d)$

$$\text{Che, per composizione, sarà } O(n^{c \cdot d}) = O(n^{c+d})$$

Quindi una composizione di due trasformazioni polinomiali ci dà una trasformazione polinomiale. ■

Ma allora come sfruttiamo questa proprietà per poter mostrare che un certo linguaggio sia *NP-Hard*?

THM:

Sia A un linguaggio *NP-Hard*, se B è un linguaggio tale che $A \leq_P B \Rightarrow B$ è *NP-Hard*

Proof:

$$\text{Poiché } A \text{ è } NP\text{-Hard}, \quad \forall L' \in NP \quad L' \leq_P A$$

$$\text{Ma da ipotesi noi abbiamo che } A \leq_P B \quad \Rightarrow L' \leq_P B \quad \Rightarrow B \text{ è } NP\text{-Hard}$$

■

Cos'è *NP-Hard*?

Sono *NP-Hard* tutti i problemi almeno difficili quanto *NP*

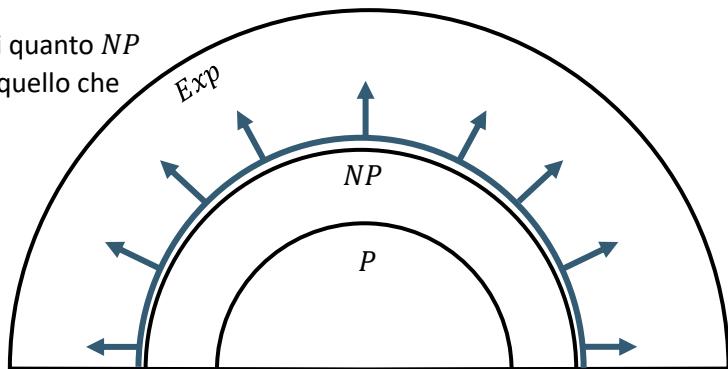
Immaginiamocelo così: che il bordo di *NP* più quello che c'è fuori. Pure i problemi indecidibili.

E i problemi *NP-completi*?

Sono i problemi sul bordo di *NP*.

Al di fuori c'è *Exponential Time*, che è ancora più grossa

Ma sono cose che *forse* vedremo.



Quello che andremo a fare è dimostrare un po' di problemi *NP-Hard* e un po' di problemi *NP-completi*. Il tutto partirà dal fatto che *Sat* è *NP-completo*.

Partiamo da *3Sat*, che è un problema intermedio che ci servirà poi per gli altri, perché molti saranno dimostrati partendo da *3Sat*.

Che differenza c'è tra *Sat* e

3Sat:

Sat
insieme delle formule
booleane in CNF

3Sat
formula in CNF in cui ogni clausola
contiene al più tre letterali = *3CNF*

Cos'è il problema *3Sat*, è l'insieme di tutte le formule booleane in *3CNF* soddisfacibili. Decidere *3Sat*, equivale a fare questa operazione "data in input una formula in *3CNF*, decidere se è soddisfacibile o no."

La cosa interessante è che si può dimostrare che *3Sat* ∈ *NP-complete*. Quindi, nonostante il vincolo sulla struttura della formula, noi abbiamo che questo problema, sebbene sia una variante di *Sat* semplificata in struttura, non è semplificata in complessità.

Per dimostrare che *3Sat* è *NP-completo* dobbiamo fare due passaggi:

1) *3Sat* ∈ *NP*

3Sat ∈ *NP* perché sottocaso di *Sat* che appartiene a *NP*

Se volessimo mostrarlo direttamente, costruiremmo una macchina non deterministica che lo calcola in tempo polinomiale. Questa macchina fa un guess della soluzione e poi verifica se quello che ha indovinato soddisfa la formula.

2) *3Sat* è *NP-Hard*

Mostriamo una riduzione:

$$\begin{array}{ccc} \textit{Sat} & \leq_P & \textit{3Sat} \\ \varphi & \rightarrow & \psi \\ (\textit{CNF}) & & (\textit{3CNF}) \end{array}$$

Ci dobbiamo inventare un modo di trasformare $\varphi \rightarrow \psi$

$$\begin{aligned} \varphi &= c_1 \wedge c_2 \wedge \dots \wedge c_n \\ c_i &= (l_1 \vee l_2 \vee \dots \vee l_k) \end{aligned}$$

$$\begin{aligned} \nearrow c'_i &= (l_1 \vee l_2 \vee h_i) \\ c_i & \searrow c''_i = (l_3 \vee \dots \vee l_k \vee \neg h_i) \end{aligned}$$

$$\|c'_i\| = 3 \quad \|c''_i\| = (\|c_i\| - 1)$$

Se noi vogliamo mostrare che sia possibile ridurre *Sat* a *3Sat*, dobbiamo far vedere che esiste una trasformazione polinomiale che prende φ , che è una formula in *CNF* generica, la trasforma in una formula ψ , in *3CNF*, tale che φ è soddisfacibile $\Leftrightarrow \psi$ è soddisfacibile.

Facciamo vedere come arrivare da φ a ψ : passiamo da $\varphi \rightarrow \varphi'$ che è una formula intermedia

Consideriamo le clausole $c_i \in \varphi$ che avranno dei letterali l_1, l_2, \dots, l_k e prendiamo una dopo l'altra; andiamo a guardare i letterali, se sono al più 3 $\Rightarrow c_i$ viene ricoppiata dentro φ' . Se, invece, sono più di 3 faccio un giochino: da c_i genero due clausole c'_i e c''_i ; dove h_i è una nuova variabile.

In questo modo abbiamo generato φ' . Male che ci va in φ' quante clausole avremo? $\boxed{2n}$

Nessuno ci vieta di reiterare il processo su φ' . Quindi se φ' , dopo questa prima riscrittura contiene ancora delle clausole con più di 3 letterali io ripeto il processo.

Se k è il numero di passaggi che dobbiamo fare per trasformare φ in ψ , cioè generiamo un φ' , poi un φ'' poi un φ''' e così via, fino a $\varphi^k = \psi$. Noi avremo ottenuto una formula il cui numero di clausole è $k \cdot m$, perché quando andiamo a dividere una clausola in due, è solo la seconda che può essere più lunga di 3, quindi ogni iterazione avremo: $2m$ clausole, $3m$ clausole, $4m$ clausole, ..., km clausole.

Questa cosa si fa in tempo polinomiale.

Ci rimane da verificare che una tale trasformazione trasformi istanze 'sì' di Sat in istanze 'sì' di $3Sat$ e istanze 'no' di Sat in istanze 'no' di $3Sat$.

La φ di partenza è soddisfacibile \Leftrightarrow la ψ di arrivo è soddisfacibile

$$\Rightarrow) \quad \varphi \text{ è soddisfacibile} \Rightarrow \psi \text{ è soddisfacibile}$$

Supponiamo che φ è soddisfacibile, $\Rightarrow \exists$ un'assegnamento σ per le variabili booleane di φ tale che φ viene valutata vera.

Mostrare ora che, se un tale σ esiste $\Rightarrow \exists \tau_\sigma$ che è un assegnamento di verità che soddisfa ψ .

τ_σ è ottenuta partendo da σ ricoppiando l'assegnamento che c'è in σ . Però τ_σ ha tutte quelle h_i aggiuntive che in σ non ci stavano; quindi, dobbiamo dargli un assegnamento di verità: consideriamo una generica clausola di φ :

$$c_i = (l_1 \vee \dots \vee l_k)$$

Mostreremo ora che se φ è soddisfacibile $\Rightarrow \varphi'$ è soddisfacibile, da cui poi per induzione tutto il resto rimane soddisfacibile.

Sappiamo che σ soddisfa φ , quindi soddisfa anche quella clausola c_i , e per farlo deve rendere almeno uno di quei letterali veri; quindi $\exists l_j \in c_i$ che è vero, causa σ .

Questa c_i noi l'abbiamo divisa in c'_i e c''_i . E quindi l_j appare (per costruzione) della riduzione, o in c'_i o in c''_i

$$\text{Se } l_j \in c'_i \Rightarrow \tau_\sigma \text{ rende vero } c'_i$$

$$\text{Se } l_j \in c''_i \Rightarrow \tau_\sigma \text{ rende vero } c''_i$$

Ma allora, l'altra clausola della coppia (c''_i se $l_j \in c'_i$ e c'_i se $l_j \in c''_i$) come facciamo a verificarla?

Abbiamo completo controllo su h_i (e quindi su $\neg h_i$) noi gli diamo il valore che ci serve e siamo in grado di verificare l'altra clausola della coppia.

\Rightarrow se φ è soddisfacibile noi siamo in grado di soddisfare anche φ'

$$\Leftarrow) \quad \psi \text{ è soddisfacibile} \Rightarrow \varphi \text{ è soddisfacibile}$$

Supponiamo che φ' sia soddisfacibile $\Rightarrow \exists \tau$ che soddisfa φ' \Rightarrow possiamo ottenere σ_τ per φ

τ deve dare un valore di verità a x_1, x_2, \dots e anche a h_1, h_2, \dots

Prendiamo una clausola di φ' che sono di due tipi:

- Quelle ricopiate da φ \Rightarrow le h non compaiono τ da un valore di verità a un letterale $\in \varphi$
- Quelle che provengono dalla divisione di c_i

$$\begin{array}{ll} c'_i = (l_1 \vee l_2 \vee [h_i]) & \Downarrow \\ c''_i = (l_3 \vee \dots \vee l_k \vee \boxed{\neg h_i}) & \nearrow c_i \end{array} \quad \text{Quindi se } c'_i \text{ e } c''_i \text{ sono verificate da } \tau \Rightarrow \text{non può essere } h_i \text{ che le verifica entrambe, in ogni caso significa che c'è un letterale } \in c_i \Rightarrow c_i \in \varphi = \text{Vera}$$

Abbiamo mostrato che questo processo di trasformazione trasforma φ soddisfacibile in una φ' soddisfacibile e che, se siamo arrivati ad una φ' soddisfacibile, era perché partivamo da una φ soddisfacibile. Qui abbiamo una catena di trasformazioni tale per cui da φ arriviamo a ψ .

Per questa ragione noi siamo in grado di trasformare istanze 'sì' di Sat in istanze 'sì' di $3Sat$ e istanze 'no' di Sat in istanze 'no' di $3Sat$. Da cui $3Sat$ è *NP-Hard*.

6.3 PROBLEMI NP-COMPLETI

6.3.1 Independent Set (IS)

Dato un grafo, un independent set per un grafo è un insieme di nodi tali per cui non esiste un arco fra nessuna coppia di nodi di questo set

$$S \subseteq V$$

Ovviamente ogni grafo ammette un independent set, il set vuoto. Quindi quelli che ci interessano sono independent set grandi perché sono quelli difficili da trovare.

Quindi ci serve una versione di decisione di questo problema che definiamo così:

$$IS = \{(G, k) \mid \exists IS \text{ di taglia almeno } k : IS \in G\}$$

Noi dimostreremo che questo problema è *NP-completo* \Rightarrow dimostreremo che:

$$\begin{array}{ll} 1) \quad IS \in NP & 2) \quad IS \in NP-Hard \end{array}$$

1. Membership: $IS \in NP$

Basta avere una macchina non deterministica che in tempo polinomiale è in grado di decidere un'istanza di questo problema, ce la facciamo?

Si, indoviniamo il risultato, l'insieme S , contiamo i suoi elementi, poi andiamo a coppie dentro S e vediamo se c'è un arco che li unisce. Se non c'è rispondiamo sì.

Questa cosa si fa in tempo polinomiale?

Il guess è polinomiale, perché dobbiamo decidere per ogni nodo se lo prendiamo o meno, il check è polinomiale.

Da questo abbiamo che $IS \in NP$.

2. Hardness: $IS \in NP-Hard$

Per dimostrarlo abbiamo due strade: o riduciamo tutti i problemi di NP a IS (che è una cosa che è meglio evitare) oppure ci scegliamo un problema NP"- completo o NP"- Hard da ridurre a IS

$$\begin{array}{ccc} \text{Vedremo ora una riduzione} & & \\ 3Sat & \leq_P & IS \\ \varphi & \xrightarrow{f} & \langle G, k \rangle \end{array}$$

Quindi noi dobbiamo trovare una funzione di trasformazione f che funziona in tempo polinomiale e trasforma formule *CNF* in coppie $\langle G, k \rangle$ tali che:

- Se la formula di partenza φ è soddisfacibile allora il grafo G ottenuto da φ ha $\|IS\| = k$.
- Se la formula è insoddisfacibile, allora il grafo G non deve avere un *IS* di taglia k .

Vedremo questa cosa un po' poco usuale in cui dobbiamo prendere una prima parte della formula e la trasformiamo in un grafo; prendiamo un esempio

$$\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$$

Che cos'è un'istanza per il problema *3CNF*?

Una formula.

Che cos'è un'istanza per *IS*?

Una coppia Grafo-numero

Noi vogliamo mappare *3CNF* su *IS* e, in particolare la difficoltà di *3Sat* sulla difficoltà di *IS*.

Che cos'è che rende difficile risolvere *3CNF*?

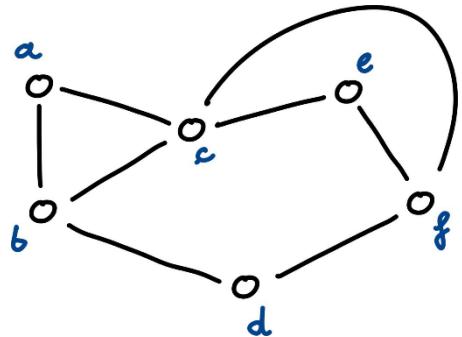
Per ogni variabile dobbiamo decidere se è *V/F*

Su *IS* che cos'è che dobbiamo decidere?

Se prendere un nodo o meno.

L'idea è inventare una trasformazione di una formula in un grafo, tale per cui, semanticamente, a variabile vera corrisponde un nodo nell'*IS* e a variabile falsa abbiamo nodo fuori dall'*IS*.

Ovviamente se vogliamo descrivere tramite l'appartenenza o meno di nodi all'*IS* se certe variabili booleane sono *V/F*, io dovrò evitare che nell'*IS* vadano a finire nodi corrispondenti a variabili opposte.



Cioè: io in S non voglio che vadano a finire nodi relativi a x_1 e $\neg x_1$, quindi mi dovrò inventare questa trasformazione tale per cui questa proprietà venga soddisfatta.

Guardiamo la definizione di IS , noi vogliamo ottenere che due nodi non possano stare in S se stanno "mimando" variabili opposte. Come facciamo a evitare che i nodi vadano a finire in S ?

Devono essere nodi congiunti

Quindi noi sfrutteremo questa idea di costruire un grafo in cui congiungiamo nodi relativi a variabili opposte, così quelli là in S non ci possono finire.

Per ognuno dei letterali, in ogni clausola, mettiamo un nodo nel nostro grafo:

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$$

- Collegiamo i nodi dei letterali opposti

Avremo tante triple di nodi quanti sono le clausole della nostra ϕ

Quello che dobbiamo verificare ora è che se:

- ϕ è soddisfacibile \Rightarrow otteniamo un insieme S di taglia k
- ϕ non è soddisfacibile $\Rightarrow \|S\| \neq k$

Supponiamo che ϕ sia soddisfacibile $\Rightarrow \exists \sigma : \sigma \models \phi$

σ assegnerà alle variabili V/F , σ soddisfa ϕ , significa che σ fa sì che in ogni clausola ci sia almeno un letterale vero.

Mostriamo che poiché ϕ è soddisfacibile $\Rightarrow \exists S : \|S\| = k$

$\forall c_i \in \phi$: prendiamo un l_j vero per σ (potrebbe essere che σ rende vero $\neg x_3$ e quindi per σ : $x_3 = F$).

Noi sappiamo che σ soddisfa tutte le clausole di ϕ , che vuol dire che ogni clausola ha almeno un letterale vero secondo σ . Prendiamo la prima clausola, becchiamo un suo letterale vero, il relativo nodo lo infiliamo dentro S , quindi costruiamo un insieme S_σ proveniente da σ .

Se il letterale di c_i è vero, prendiamo il relativo nodo di quella tripletta di nodi e lo infiliamo dentro S_σ .

Quantni nodi ci stanno in S_σ ?

Tante quante sono le clausole

$$\Rightarrow \|S_\sigma\| = m \Rightarrow m \text{ corrisponde a } k$$

I nodi che abbiamo infilato in S_σ sono o non sono un IS ? Supponiamo che non lo siano

Se non sono un IS ci devono essere due nodi dentro S_σ che sono collegati da un arco del grafo qui sopra.

Ma quali nodi sono collegati in questo grafo?

I nodi con letterali opposti.

Quindi se due nodi $\in \sigma$ fossero collegati $\Rightarrow \sigma$ starebbe valutando vero un letterale e il suo opposto.

Ma ciò non è possibile $\Rightarrow S_\sigma$ è un IS .

Quindi abbiamo dimostrato che questa trasformazione, se parte da una formula ϕ soddisfacibile, ottiene una coppia $\langle G, m \rangle$ in cui G è un IS di taglia m , dove m è il numero di clausole di ϕ .

Ora dobbiamo mostrare che se arriviamo a una coppia del genere era perché σ di partenza era soddisfacibile.

Supponiamo di essere arrivati a un'istanza 'sì', quindi il grafo qui sopra ha un IS di taglia m

Sia S un IS di taglia m su un grafo ottenuto in questo modo. Noi vorremmo che andando a guardare i nodi dentro S noi tiriamo fuori un assegnamento di verità per ϕ che ci certifichi che ϕ sia soddisfacibile.

Prendiamo S e costruiamo un σ_S . Quantni nodi ha S ?

Almeno m , per definizione.

Se ci sta un nodo dentro $S \Rightarrow$ assegniamo a σ il relativo assegnamento di verità coerente col fatto che quel letterale sia dentro S .

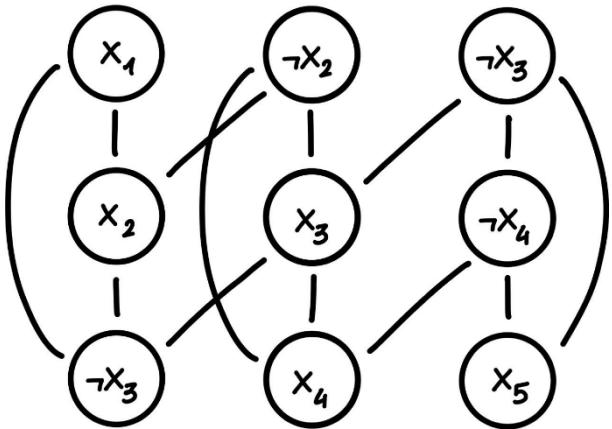
$$\text{Quindi se: } x_2 \in S \Rightarrow \sigma \text{ assegna Vero a } x_2 \\ \neg x_3 \in S \Rightarrow \sigma \text{ assegna Falso a } x_3$$

Possiamo essere certi che un assegnamento di verità σ_S soddisfi la formula di partenza? No.
Per soddisfare ϕ , σ_S dovrebbe soddisfare tutte le clausole, ma non siamo sicuri che σ_S le soddisfi tutte.

Questa riduzione non funziona perché S , a cui noi chiediamo di essere almeno m , (nell'esempio $m = 3$) potrebbe essere $S = (x_1 \ x_2 \ \neg x_3)$, cioè io riempio S con più nodi che vengono dalla stessa clausola; quindi, per formule più intricate non ho la garanzia che σ_S soddisfi ϕ .

Dobbiamo evitare che in S finiscano nodi provenienti da una stessa clausola, e per farlo li collegiamo:

Mostriamo quindi che questa riduzione è corretta:



Sia ϕ una formula in CNF, dobbiamo generare la coppia $\langle G, k \rangle$ per farlo dobbiamo dire chi sono i nodi e chi sono gli archi e chi è k .

Abbiamo un nodo per ognuno dei letterali che appare dentro ϕ . Gli archi di G : abbiamo archi che collegano i letterali di una stessa clausola, e archi che collegano nodi relativi a letterali opposti. k è il numero di clausole di ϕ .

Dobbiamo dimostrare che, se partiamo da ϕ soddisfacibile, $f(\phi)$ genera una coppia grafo-numero tale che il grafo ha un IS di taglia k con $k =$ numero di clausole di ϕ .

\Rightarrow Supponiamo che ϕ sia soddisfacibile, $\Rightarrow \exists \sigma$ che soddisfa ϕ , mostriamo che G ha un IS di taglia $k = m$.

Partiamo da σ e costruiamo S_σ , per ognuna delle triplettie di nodi prendiamo un nodo il cui letterale è Vero in σ , abbiamo garanzia che esista perché σ soddisfa ϕ .

In S_σ finiranno tanti nodi quante le clausole, in particolare un nodo per ogni clausola. $\|S_\sigma\| = m$

S_σ è un IS? Supponiamo per assurdo che non lo sia; se non lo è devono esserci due nodi dentro G che sono collegati da un arco. Quali archi stanno in G ? Quelli delle triplettie o quelli tra le triplettie.

È possibile che ci siano due nodi in S_σ provenienti dalla stessa triplettie e quindi collegati da un arco?

No, perché ne abbiamo preso uno a triplettia.

È possibile che ci siano due nodi in S_σ relativi a letterali opposti?

No, perché partiamo da un assegnamento σ che è consistente.

Quindi S_σ è un IS di taglia k e se ϕ è soddisfacibile otteniamo un'istanza 'sì' del problema IS.

\Leftarrow Supponiamo che siamo arrivati ad un'istanza 'sì' dell'IS; sia S un IS di taglia k ,

Vogliamo mostrare che da S riusciamo ad ottenere un assegnamento σ_S che ci certifichi che la formula iniziale di partenza è soddisfacibile. $\|S\| = m$

È possibile che dentro S abbiamo più nodi provenienti dalla stessa triplettia? No. Quindi in S abbiamo esattamente 1 nodo $\forall c_i$.

Come generiamo σ_S ? Per ogni nodo $\in S$ generiamo, se ci sta un letterale tipo x_2 , in σ_S mettiamo $x_2 = V$, se dentro S ci sta un nodo di un letterale negativo, in σ_S mettiamo $x_3 = F$. E lo facciamo per tutte le c_i .

σ_S soddisfa ϕ ? Supponiamo per assurdo che non sia così, di conseguenza $\exists c_n \in \phi$ i cui letterali sono tutti F secondo σ_S , ma ciò equivale al fatto che in S non avremmo preso alcun nodo di una certa triplettia. Ma ciò non è possibile $\Rightarrow \sigma_S$ dev'essere un assegnamento di verità che certifichi che ϕ è soddisfacibile.

Quindi noi otteniamo che questa riduzione trasforma istanze 'sì' di CNF in istanze 'sì' di IS e istanze 'no' di CNF in istanze 'no' di IS $\Rightarrow IS \in NP-Hard$ e siccome $IS \in NP \Rightarrow IS \in NP-completo$.

6.3.2 Vertex Cover (VC)

Un vertex cover di un grafo è un sottoinsieme dei suoi vertici C tale per cui copriamo tutti gli archi.

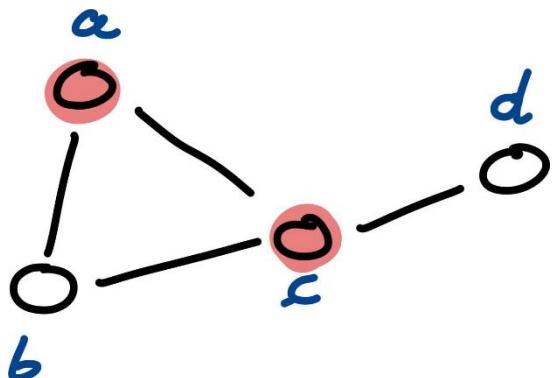
Significa che noi siamo in grado di vedere tutti gli archi.

$$\text{i.e. } C = \{a, c\}$$

Formuliamo il problema di decisione corrispondente:

Dato un grafo, è vero o no che un grafo ha un vertex cover?

Sì, tutti i nodi. Abbiamo quindi un problema banale.



Problema più difficile è:

$$VC = \left\{ \langle G, k \rangle \middle| \begin{array}{l} G \text{ ha un vertex} \\ \text{cover di taglia} \leq k \end{array} \right\}$$

Dimostriamo che $VC \in NP$ -completo:

1) $VC \in NP$? Sì.

Fissiamo i nodi e controlliamo che siano al più k e che ricopriano tutto; questo guess & check può essere fatto in tempo polinomiale? Si $\Rightarrow VC \in NP$

2) $VC \in NP\text{-Hard}$

$$\begin{array}{ccc} IS & \xrightarrow[f]{\leq_P} & VC \\ \langle G, k \rangle & \xrightarrow{\text{Grafo-numero}} & \langle H, l \rangle \\ \text{Grafo-numero} & & \text{Grafo-numero} \end{array}$$

Una volta che iniziamo ad avere problemi *NP-completi* li possiamo usare come “sorgenti di complessità”; ora, per esempio, sebbene sia possibile ridurre *Sat* da *VC* utilizziamo *IS* perché è già un problema su grafi che ci rende la riduzione molto più semplice.

Quello che noi dovremmo fare è trovare una funzione di trasformazione, calcolabile in tempo polinomiale, che data una coppia $\langle G, k \rangle$ Grafo-numero tira fuori una coppia $\langle H, l \rangle$ tale per cui il grafo $S \in G : \|S\| \geq k \Leftrightarrow \|VC\| \leq l$.

La riduzione si basa su questo lemma:

Lemma:

Sia $G = \langle V, E \rangle$ un grafo non orientato, e sia $S \subseteq V$ un insieme di nodi.

Allora S è un Independent Set $\Leftrightarrow V \setminus S$ è un vertex cover di G .

Proof: \Rightarrow S è un Independent Set, assumiamo per assurdo che $V \setminus S$ non sia un vertex cover di G .

Se $C = V \setminus S$ non è un vertex cover $\Rightarrow \exists$ un arco $\in G$ i cui end-point (entrambi) $\notin C$, ma siccome C è il complemento di S allora gli end-point di questo arco stanno tutti e due in $S \Rightarrow$ in S ci stanno due nodi agganciati e quindi S non è un independent set. Assurdo, perciò C dev'essere un vertex cover.

\Leftarrow Supponiamo che $C = V \setminus S$ sia un vertex cover, assumiamo per assurdo che S non sia un IS

Se S non è un independent set $\Rightarrow \exists$ due nodi in S collegati da un arco, ma se ci sono due nodi in S collegati da un arco, quello è un arco i cui due end-point $\notin C \Rightarrow C$ non è un vertex cover. Assurdo, quindi S dev'essere per forza un independent set.

Quindi possiamo ottenere la trasformazione in questo modo

$$\langle G, k \rangle \longrightarrow \langle H, l \rangle, \quad H = G, \quad l = |v| - k$$

Dobbiamo ora dimostrare che se G ha un *IS* di taglia almeno $k \Rightarrow H$ ha un vertex cover di taglia al più l

Supponiamo che G abbia un independent set S di taglia k . Il complemento di S è un vertex cover di G di $G = H \Rightarrow$ il complemento di S è un vertex cover di H . La taglia del complemento di S è $l = |v| - k$.

Supponiamo di essere arrivati a un'istanza 'sì' di vertex cover; quindi, in H esiste un vertex cover di taglia l . Chiamiamo questo vertex cover C ; consideriamo il complemento di C , per il lemma il complemento di C è un independent set di H , ma $H = G \Rightarrow$ il complemento di C è un independent set di G .

Il complemento di G è un independent set sufficientemente grande? Per forza! Siccome $|C| \leq l \Rightarrow$ il suo complemento ha una taglia $\leq k$, per definizione della trasformazione.

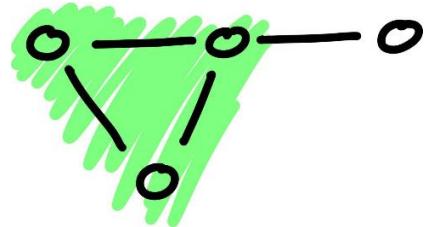
Di conseguenza, se siamo arrivati a un'istanza 'sì' di vertex cover era perché partivamo da un'istanza sì di independent set.

$$\Rightarrow VC \in NP\text{-Hard} \Rightarrow VC \in NP\text{-complete}$$

■

6.3.3 Clique

Una Clique su grafo è un sottografo completamente connesso. Quindi avremo un grafo come in figura, e la parte evidenziata in verde è una clique di taglia 3, perché i nodi sono collegati a coppie.



Dato un grafo, è vero o no che questo grafo ammette una clique?

L'insieme vuoto o l'insieme di un nodo. Quindi il problema è banale, ci serve la sua variante utile

$$\text{Clique} = \{\langle G, k \rangle \mid G \text{ ha una clique di taglia } \geq k\}$$

Dimostriamo che $\text{Clique} \in \text{NP-completo}$:

1) $\text{Clique} \in \text{NP}$? Si.

Indoviniamo la clique, controlliamo la clique. Si fa in tempo polinomiale $\Rightarrow \in \text{NP}$

2) $\text{Clique} \in \text{NP-Hard}$

$$\begin{array}{ccc} IS & \xrightarrow[f]{\leq_P} & \text{Clique} \\ \langle G, k \rangle & \xrightarrow{\quad \text{Grafo-numero} \quad} & \langle H, l \rangle \\ & \xrightarrow{\quad \text{Grafo-numero} \quad} & \end{array}$$

Sia $G = \langle V, E \rangle$ un grafo, definiamo $\bar{G} = \langle V, \bar{E} \rangle$ come il grafo in cui i nodi sono gli stessi e gli archi sono flippati, dove non ci stava lo mettiamo, dove ci stava ce lo togliamo.

Lemma:

Sia G un grafo, allora S è un independent set di $G \Leftrightarrow S$ è una clique in \bar{G}

Proof: \Rightarrow Sia S un independent set di G ; assumiamo per assurdo che S non sia una clique in \bar{G}

Se S non è una clique in \bar{G} allora vuol dire che in S ci stanno due nodi che in \bar{G} non sono collegati. Ma allora quei due nodi in G erano una collegati e allora S non era un independent set, *assurdo*.

\Leftarrow Sia S una clique di \bar{G} ; assumiamo per assurdo che S non sia un independent set in G

Se S non è un independent set in G allora ci sono due nodi di S che sono collegati dentro G .

Ma allora, per def. di \bar{G} , quei due nodi in \bar{G} erano sganciati da cui S non era una clique, *assurdo*.

Sfruttiamo questa proprietà per la riduzione: partendo da

$$\langle G, k \rangle \rightarrow \langle H, l \rangle$$

$$H = \bar{G} \wedge l = k$$

Dobbiamo dimostrare che questa trasformazione trasforma istanze 'sì' di IS in istanze 'sì' di $Clique$ e istanze 'no' di IS in istanze 'no' di $Clique$.

Supponiamo che $\langle G, k \rangle$ sia un'istanza 'sì' di IS , ciò vuol dire che G ammette un independent set S di taglia k ; per il lemma precedente, S è una clique di \bar{G} di taglia k , ma per definizione di $\langle H, l \rangle$, S sarà una clique di H di taglia l .

Quindi, partendo da un'istanza 'sì' di IS siamo arrivati a un'istanza 'sì' di $Clique$.

Supponiamo di essere arrivati a un'istanza 'sì' di $Clique$, quindi H ha una clique S di taglia l . Per il lemma precedente, S è un independent set di taglia l di \bar{H} , ma per definizione di $\langle H, l \rangle$, $\bar{H} = G$ e $l = k$ e quindi S è un independent set di G di taglia k .

Quindi se siamo arrivati a un'istanza 'sì' di era perché partivamo da un'istanza 'sì'.

Da cui IS si riduce a $Clique \Rightarrow Clique \in \text{NP-Hard}$

$$\Rightarrow Clique \in \text{NP-complete}$$

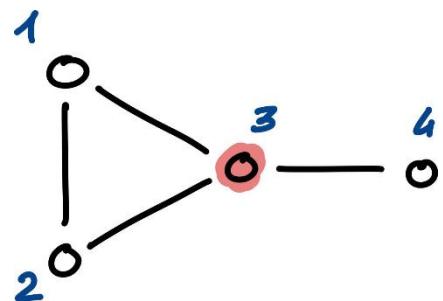


6.3.4 Dominating set (DS)

Dominating set è un altro problema su grafo, simile a vertex cover ma non completamente. Un dominating set di un grafo è un insieme di nodi, tale per cui tutti i nodi fuori il dominating set sono agganciati a qualche nodo del dominating set.

Può essere anche un solo nodo (ie. $D = \{3\}$); però D non è un vertex cover perché c'è l'arco ab che non è coperto da niente.

Quindi per il dominating set noi vogliamo essere in grado di raggiungere da D tutti gli altri nodi; invece, nel vertex cover noi vogliamo essere in grado di coprire tutti gli archi.



Un vertex cover è sempre un dominating set, ma non viceversa.

$$DS = \{(G, k) \mid G \text{ ha un dominating set di taglia } \leq k\}$$

Dimostriamo che $DS \in NP$ -completo:

1) $DS \in NP$

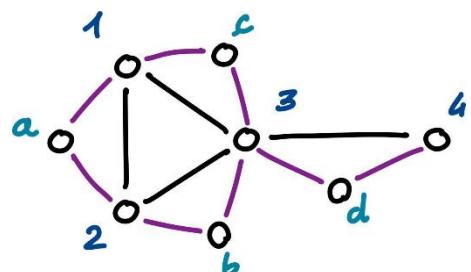
Si, perché indovino un insieme e poi faccio il check, il tutto in tempo polinomiale.

2) $DS \in NP$ -Hard

$$\begin{array}{ccc} VC & \xrightarrow[f]{\leq_P} & DS \\ \langle G, k \rangle & & \langle H, l \rangle \\ \text{Grafo-numero} & & \text{Grafo-numero} \end{array}$$

H non può essere la semplice copia di G , perché un VC è sempre un DS ma non è certo l'opposto.

Allora, supponiamo di star trasformando il grafo qua sopra, noi aggiungiamo dei nodi: nello specifico, un vertice in H per ognuno degli archi di $G \Rightarrow \{a, b, c, d\}$. Poi li agganciamo ai rispettivi nodi dell'arco. E questo è come si ottiene H , $l = k$.



Mostriamo che se $\langle G, k \rangle$ è un'istanza 'si' di vertex cover, allora $\langle H, l \rangle$ è un'istanza 'si' di dominating set, e, se $\langle H, l \rangle$ è un'istanza 'si' di dominating set allora era perché $\langle G, k \rangle$ è un'istanza 'si' di vertex cover.

Supponiamo che $\langle G, k \rangle$ sia un'istanza 'si' di $VC \Rightarrow \exists$ un insieme C di taglia k che è un vertex cover di G . Siccome C era un vertex cover di G da C in H riusciamo sicuramente a raggiungere tutti i nodi con un nome *numerico*, perché C è un vertex cover di G . Però per costruzione di H da C raggiungiamo pure i nodi con un nome *letterale* perché il grafo H è stato costruito così.

Quindi se C è un vertex cover di G di taglia $k \Rightarrow C$ sarà un dominating set di H di taglia l .

Supponiamo che $\langle H, l \rangle$ sia un'istanza 'si' di dominating set. Quindi esiste un dominating set D di taglia l . Questo insieme D , per com'è costruito H può contenere nodi *letterali* o nodi *numerici*.

Prendiamo D e facciamo questa operazione: se compare un nodo *lettera* dentro D lo sostituiamo con uno dei due nodi numero a cui è agganciato. Quindi otteniamo un \tilde{D} i cui nodi sono tutti *numerici*.

Ma allora, siccome \tilde{D} è un dominating set di H deve raggiungere da \tilde{D} non solo i nodi *numerici*, ma anche i nodi *letterali*. Di conseguenza avremo che i nodi di \tilde{D} riescono a prendere i nodi che mimano gli archi. Quindi \tilde{D} su G è un vertex cover di taglia k .

Quindi se siamo arrivati a un'istanza 'si' era perché partivamo da un'istanza 'si'.

Da cui VC si riduce a $DS \Rightarrow DS \in NP$ -Hard

$$\Rightarrow DS \in NP\text{-complete}$$

6.3.5 Definizione differente della classe NP

Da quello che abbiamo visto dagli esercizi appena fatti, non è difficile dimostrare che un problema stia in NP , sostanzialmente data un'istanza per rispondere 'si' facevamo guess & check entrambi in tempo polinomiale. Praticamente tutti i problemi in NP li abbiamo sempre risolti nello stesso modo, guess & check polinomiali.

Ricordiamo la definizione di NP :

$$NP = \bigcup_{c \geq 1} NTime(n^c)$$

Dove $NTime(n^c)$ è l'insieme dei linguaggi per cui esiste una macchina non deterministica che li decide in tempo $O(n^c)$.

❓ In questa definizione la MdT che decide un certo linguaggio, quando fa i guess ?

Quando vuole.

Eppure, in tutti gli esempi che abbiamo visto, abbiamo sempre notato che questi linguaggi si risolvono sempre facendo un guess preliminare e un check successivo.

❓ Ma è solo un caso ? O tutti i linguaggi di NP possono essere decisi in questo modo ?

DEF: (relazione binaria)

Una relazione binaria R su Σ^* è definita come:

$$R \subseteq \Sigma^* \times \Sigma^*$$

Nei problemi che abbiamo visto della classe NP noi facevamo un guess e poi un check, e che cosa indovinavamo? Indovinavamo la **prova** che l'istanza che avevamo in input era un'istanza 'si'. Questa cosa che indoviniamo, questa **prova** che guessavamo noi la chiamiamo **certificato** è un certificato che ci certifica che una certa istanza è un'istanza 'si'.

Combineremo ora questa idea del certificato all'interno della nozione di relazione binaria e vedremo che NP si può definire anche in un altro modo.

La nostra macchina indovinava un certificato, questo certificato non può essere "grosso" perché non abbiamo tempo di indovinare un certificato più grande di polinomiale; quindi, diciamo che questo certificato è **conciso**. La macchina poi farà un test per verificare la correttezza del certificato, mettendoci tempo polinomiale, e diciamo che questo certificato è **polynomialmente verificabile**.

- Guessiamo un certificato conciso
- Verifichiamo in tempo polinomiale la veridicità del certificato

Quello che noi vogliamo andare a mostrare è che i linguaggi NP , a parte essere quelli per i quali esiste una macchina non deterministica che li decide in tempo polinomiale, sono quei linguaggi le cui istanze 'si' ammettono certificati concisi e polynomialmente verificabili.

Vogliamo spostare la nostra attenzione a un tipo di computazione di macchina non deterministica in cui la fase di guess avviene tutta all'inizio.

DEF: (polynomialmente bilanciata)

$$R \subseteq \Sigma^* \times \Sigma^*, \quad \langle x, y \rangle \in R$$

Noi diciamo che R è polynomialmente bilanciata, se $\|y\| \leq \|x\|^c$, con c fissato

Cioè R è una relazione polynomialmente bilanciata se le stringhe nella parte destra della coppia non sono troppo più grosse delle stringhe in prima posizione.

DEF: (polynomialmente decidibile)

Noi diciamo che R è polynomialmente decidibile, se R si può decidere in tempo polinomiale deterministico.

Quando diciamo che R si decide in tempo *polinomiale deterministico*, significa che data una coppia $\langle x, y \rangle$ codificata in un alfabeto opportuno noi in tempo polinomiale deterministico siamo in grado di dire se quella coppia appartiene alla relazione o meno. R è semplicemente un sottoinsieme di tutte le possibili coppie di stringhe.

Un collegamento con basi di dati: le tabelle sono relazioni; quindi, noi diciamo che sono relazioni n -arie mentre noi qua ci interessiamo di relazioni binarie, cioè tabelle con due colonne. Una relazione binaria è una tabella con due colonne.

Una relazione è polinomialmente bilanciata se le stringhe in seconda colonna sono limitate da un polinomio delle stringhe in prima colonna.

La relazione è polinomialmente decidibile se esiste una MdT deterministica che in tempo polinomiale è in grado di stabilire se questa coppia fa parte di questa tabella; se questa tabella è di taglia finita, il gioco è facile, noi dobbiamo pensarla in maniera un po' più estesa, cioè R è una tabella infinita, e a noi serve un algoritmo per stabilire se una certa coppia sta in quella tabella o meno.

THM:

Sia L un linguaggio, $L \in NP \Leftrightarrow \exists$ una relazione binaria R_L polinomialmente bilanciata e decidibile tale che:

$$L = \{x \mid \langle x, y \rangle \in R_L\}$$

In prima colonna ci stanno tutte e solo le stringhe del linguaggio.

$L \in NP$ se noi siamo in grado di inventarci una tabella tale per cui le stringhe in prima colonna sono le stringhe del linguaggio, e in seconda colonna abbiamo i certificati.

Un esempio:

$$IS = \{x \mid \langle x, y \rangle \in R_{IS}\}$$

R_{IS} è una relazione binaria in cui x è una coppia $\langle G, k \rangle$ e y è un independent set di taglia k .

Proof:

\Leftarrow Sia L un linguaggio tale che esiste una relazione R_L polinomialmente decidibile e bilanciata tale che

$$L = \{x \mid \langle x, y \rangle \in R_L\}$$

Vogliamo mostrare che $L \in NP$

Questo linguaggio può essere deciso in tempo polinomiale da una MdT non deterministica e siccome M_L è polinomialmente decidibile $\Rightarrow \exists$ una MdT deterministica che in tempo polinomiale è in grado di decidere R_L . Come fa una MdT non deterministica a decidere L , dato x in input indovina y , scrive y su nastro di lunghezza $\|x^c\|$. Sul nastro avremo poi la coppia $\langle x, y \rangle$ quindi deve essere lanciata una procedura che data la coppia $\langle x, y \rangle$ stabilisca se questa coppia appartiene a R_L o meno, siccome R_L è polinomialmente decidibile \Rightarrow questa MdT non deterministica fa quello che farebbe il riconoscitore di R_L .

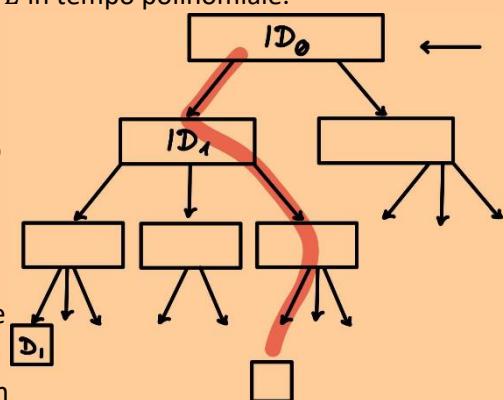
Quindi in tempo polinomiale non deterministico siamo in grado di riconoscere R_L , da cui se L è caratterizzato da una relazione polinomialmente decidibile e bilanciata $\Rightarrow L \in NP$.

\Rightarrow Sia L un linguaggio in NP , vogliamo mostrare che esiste una relazione R_L polinomialmente decidibile e bilanciata tale che $L = \{x \mid \langle x, y \rangle \in R_L\}$

Se $L \in NP \Rightarrow \exists$ una MdT non deterministica M che decide L in tempo polinomiale.

La computazione di questa macchina è caratterizzata da questo computation tree. Se la macchina lavora in tempo polinomiale, possiamo dire che il branch di computazione più lungo è polinomiale nella taglia dell'input. Supponiamo che questo polinomio sia $P_1(n)$ se la macchina lavora in tempo $P_1 \Rightarrow$ non potrà scrivere qualcosa polinomialmente più grande perché le manca il tempo.

\Rightarrow ogni instantaneous description sarà di taglia polinomiale. Supponiamo che quello rosso sia il path accettante, noi Possiamo listare la sequenza dell'instantaneous description



Questa sequenza di instantaneous description della macchina è sostanzialmente una prova del fatto che la stringa in input debba essere accettata. La nostra idea sarà usarla come certificato.



Perché costituisce la prova che la macchina accetti.

Le instantaneous description di questa sequenza sono $P_1(n)$ e ogni instantaneous Description è lunga P_2 , quindi avremo un certificato costituito da un numero polinomiale di instantaneous desc. ognuna delle quali ha lunghezza polinomiale.

Quindi overall questo certificato ha taglia $P_1(n) \cdot P_2(n)$

Noi dobbiamo andare a costruire un R_L in cui x sono le istanze del linguaggio e y è il certificato. Se in y ci andiamo a mettere il certificato qui sopra, è vero o no che solamente le istanze 'si' di L hanno un certificato di questo tipo? Si, quindi il certificato è sensato.

Quindi in linea di principio noi potremmo caratterizzare l' R_L del linguaggio L in questo modo in cui x è l'istanza del linguaggio e y è la sequenza di passi che la MdTnD fa per dire 'si' sulla stringa x

Il certificato è conciso? Si, ogni instant. desc. è di taglia polinomiale.

$\Rightarrow R_L$ è polinomialmente bilanciata

Il certificato è polinomialmente verificabile? Si, noi verifichiamo a coppie, con la ID successiva e questo check si fa in tempo polinomiale det. perché verifichiamo che questa cosa sia legale per una funzione di transizione fissata.

Quindi noi abbiamo che un linguaggio appartiene a NP e solo se è caratterizzabile da una relazione binaria polinomialmente decidibile e polinomialmente bilanciata.

■

Ma quindi questa nuova caratterizzazione di NP dove sta?

I linguaggi di NP , poiché sono caratterizzati dalla relazione polinomialmente decidibile e polinomialmente bilanciata, sono quei linguaggi le cui soluzioni possono essere verificate in tempo polinomiale deterministico.

Quindi, in soldoni, abbiamo che in P ci stanno i problemi la cui soluzione si calcola in tempo polinomiale; in NP abbiamo quei linguaggi che una volta che ci danno un certificato, quello lo verifichiamo in tempo polinomiale.

Quindi la questione P vs NP si annoda su questo punto cruciale:

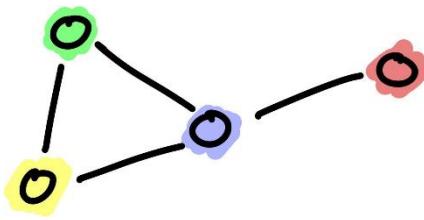
"è vero o no che tutti i problemi la cui soluzione possa essere verificata facilmente, sono anche problemi la cui soluzione può essere calcolata facilmente?"

6.3.6 Colorabilità dei grafi (Col)

La colorazione di un grafo è una funzione che assegna un colore a ogni nodo. Una colorazione dev'essere una funzione tal per cui nodi agganciati da un arco hanno colori distinti.

Dato un grafo, è vero o no che ci sta una colorazione per quel grafo?

Certo! Usiamo tanti colori quanti nodi.



Allora il problema è colorare il grafo con una tavolozza limitata.

$$\text{Col} = \{\langle G, k \rangle \mid \text{È possibile colorare i nodi di } G \text{ con al più } k \text{ colori}\}$$

Dimostriamo che $\text{Col} \in \text{NP-completo}$:

1) $\text{Col} \in \text{NP}$

Appartiene a NP , perché indoviniamo una colorazione in tempo polinomiale, verifichiamo che il guess sia corretto in tempo polinomiale.

2) $\text{Col} \in \text{NP-Hard}$

$$\begin{array}{ccc} 3\text{Sat} & \xrightarrow[f]{\leq_P} & \text{Col} \\ \phi & & \langle G, k \rangle \\ 3\text{CNF} & & \text{Grafo-numero} \end{array}$$

Un'istanza sì per 3Sat è una formula soddisfacibile per la quale esiste un assegnamento di verità per le variabili tale che la formula sia soddisfatta.

Un'istanza 'si' per Col è una coppia grafo-numero tale che quel grafo ammette una colorazione dei suoi nodi con al più k colori tale per cui quella colorazione assegna colori diversi a nodi adiacenti.

Supponiamo di avere:

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$$

Da questa formula generiamo un grafo che sia k -colorabile. Partiamo con questi nodi T, F, B dei quali ne facciamo una clique.

Da un lato dobbiamo decidere se una variabile è vera o falsa

Dall'altro dobbiamo decidere che colore assume un certo nodo.

La nostra riduzione dovrà mappare assegnamenti di verità variabili in colori da dare ai nodi.

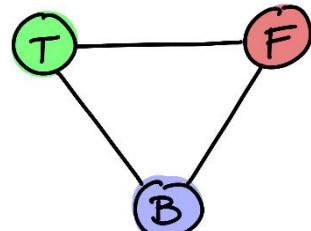
Per colorare questa clique ci servono almeno tre colori. Colore del **Vero**, del **Falso** e **neutro (o Base)**.

Quindi nel nostro grafo i nodi che avranno lo stesso colore del nodo T è come se gli stessimo assegnando il valore **Vero**, stessa cosa per F e B .

Siccome noi dobbiamo mappare l'assegnamento di verità su una colorazione, noi per ogni variabile x_i avremo due nodi x_i e $\neg x_i$; siccome dovranno avere colori diversi li colleghiamo tra loro. Siccome noi dobbiamo mimare che possano essere veri o falsi, li agganciamo tutti al nodo **base**, in modo che possano assumere solo i colori **verde** e **rosso**.

Quindi x_i e $\neg x_i$ potranno essere uno **verde** e l'altro **rosso** o al contrario, però non potranno essere **blu**.

Così questo pezzo di grafo ci codifica l'assegnamento di verità delle variabili booleane.



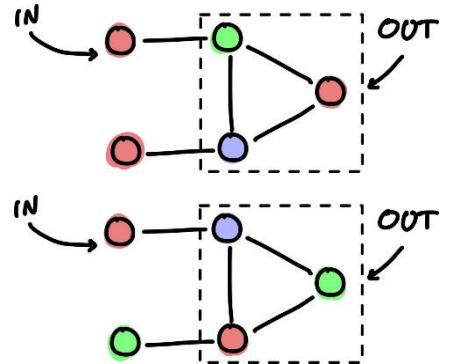
Dobbiamo adesso inventarci altri pezzi di grafo che vadano a codificare se una clausola sia stata soddisfatta da un certo assegnamento o meno.

Simuliamo tramite grafi e colorazione un "OR"; in questo modo:

Supponiamo di dare il colore **falso** ai due input; se io li coloro entrambi **rossi**, quelli subito successivi dovranno essere uno **verde** e uno **blu**, quale non importa perché sono entrambi collegati.

Quindi questo pezzo di grafo è interessante perché se ha in input due nodi **rossi**, in output ha un nodo **rosso**.

Supponiamo che invece siano uno **rosso** e l'altro **verde**, i due nodi intermedi dovranno essere uno **blu** e uno **rosso**, di conseguenza l'uscita potrà essere colorata **verde**.



Quindi questo pezzettino di grafo sta codificando la funzione logica "OR". Quindi noi utilizziamo questo *sottografo* per codificare le clausole, c_1 e c_2 hanno 3 elementi e noi metteremo in fila due sottografi OR.

Quindi noi abbiamo che nei due riquadri tratteggiati stiamo andando a codificare le clausole della formula, noi vogliamo che l'output sia vero, quindi **verde**, per imporre che gli ultimi due nodi dei sottografi OR siano **verdi** li colleghiamo sia a **B** che a **F**.

Questo grafo quindi è particolare, è un grafo ottenuto dalla formula G , con $k = 3$.

Vogliamo mostrare è che ϕ è soddisfacibile \Leftrightarrow il grafo così generato è un grafo 3-colorabile.

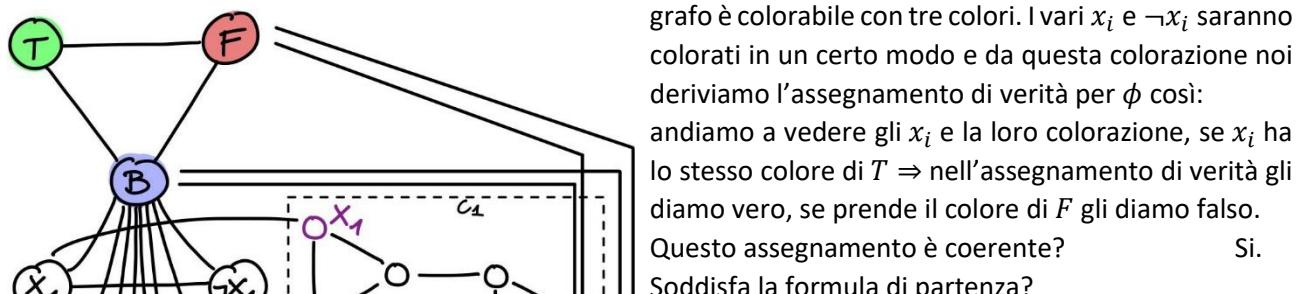
\Rightarrow) Supponiamo che la formula di partenza ϕ sia soddisfacibile. Questo significa che ammette un assegnamento di verità σ per le sue variabili tale che σ soddisfa tutte le clausole di ϕ . Allora noi coloriamo il grafo G ottenuto da ϕ con tre colori in questo modo: i tre nodi come in figura.

Se x_i è Vero in $\sigma \Rightarrow$ verde

Se x_i è Falso in $\sigma \Rightarrow$ rosso

E a $\neg x_i$ gli diamo l'opposto. Quindi noi stiamo prendendo l'assegnamento di verità che soddisfa ϕ e lo stiamo trasformando in una colorazione per questo grafo. Dopodiché dobbiamo colorare la parte rimanente del grafo, che è colorabile perché per le proprietà dei sottografi "OR" se questi sottografi ricevono in input dei nodi verdi allora l'output può essere colorato verde. E siccome ciò è vero per tutte le clausole, perché σ soddisfa tutte le clausole di ϕ , noi avremo un modo di colorare in modo coerente i sottografi.

\Leftarrow) Supponiamo di essere arrivati a un'istanza 'si' del problema della colorabilità, allora vuol dire che questo grafo è colorabile con tre colori. I vari x_i e $\neg x_i$ saranno colorati in un certo modo e da questa colorazione noi deriviamo l'assegnamento di verità per ϕ così:



andiamo a vedere gli x_i e la loro colorazione, se x_i ha lo stesso colore di $T \Rightarrow$ nell'assegnamento di verità gli diamo vero, se prende il colore di F gli diamo falso.

Questo assegnamento è coerente? Si.

Soddisfa la formula di partenza?

Supponiamo per assurdo che non la soddisfi e che ci sia una clausola in ϕ che sia falsa.

Se è falsa, vuol dire che tutti i letterali sono falsi, ma se sono tutti falsi allora il sottografo "OR" non darebbe output rosso ma ciò non può essere perché sono entrambi agganciati a F .

Quindi l'assegnamento di verità ottenuto dalla colorazione di questo grafo soddisfa la formula ϕ .

6.3.7 Exact cover (EC)

L'input per Exact cover sono due insiemi:

$$U = \{u_1, \dots, u_n\} \quad F = \{s_1, \dots, s_m\} \quad s_i \subseteq U$$

Un insieme U di oggetti che chiamiamo *universo* e una famiglia F di sottoinsiemi di U .

L'output, essendo un problema di decisione, è una risposta 'sì'/no'.

Rispondiamo sì, quindi un'istanza $\langle U, F \rangle$ di exact cover è un'istanza sì, se esiste tra gli insiemi di F una partizione di U . Per partizione intendiamo che questa selezione degli insiemi s_i sono insiemi disgiunti e la loro unione somma a U .

Dimostriamo che $EC \in NP$ -completo:

1) $EC \in NP$

Indoviniamo la soluzione, e poi check in tempo polinomiale.

2) $EC \in NP$ -Hard

$$\begin{array}{ccc} 3Sat & \xrightarrow[f]{\leq_P} & EC \\ \phi & \longrightarrow & \langle U_\phi, F_\phi \rangle \\ 3CNF & & \text{insieme di oggetti,} \\ & & \text{Famiglia di sottoinsiemi di } U \end{array}$$

Noi abbiamo che ϕ ha questa forma:

$$\phi = c_1 \wedge \dots \wedge c_l, \quad c_i = \lambda_{i_1} \vee \lambda_{i_2} \vee \lambda_{i_3}$$

Noi prendiamo e trasformiamo le cose in questo modo:

$$U_\phi = \{x_i \mid 1 \leq i \leq n\} \cup \{c_j \mid 1 \leq j \leq l\} \cup \{P_{j,k} \mid \forall \lambda_{j_k}\}$$

Abbiamo un oggetto x_i per ognuna delle variabili booleane che appare dentro ϕ .

Poi, siccome noi vogliamo simulare se la formula sia soddisfacibile o meno tramite un partizionamento, ci serviranno oggetti dentro U che ci simulino se una certa clausola è stata soddisfatta o meno.

Quindi l'universo U_ϕ ha oggetti di tre tipi:

Un oggetto per ogni variabile, un oggetto per ogni clausola, un oggetto per ogni letterale.

Definiamo F_ϕ che contiene:

- > un insieme $\{P_{j,k}\}$ per ogni oggetto $P_{j,k} \in U_\phi$
- > un insieme di altri insiemi che ci servono a simulare l'assegnamento dei valori di verità:
 \forall variabile booleana $x_i \in \phi$ abbiamo due insiemi:
 - $T_{i,T} = \{x_i\} \cup \{P_{j,k} \mid \lambda_{j_k} = \neg x_i\}$ Mettiamo l'oggetto x_i più tutti gli oggetti associati a quei letterali che quando $x_i = V$ loro vengono falsificati
 - $T_{i,F} = \{x_i\} \cup \{P_{j,k} \mid \lambda_{j_k} = x_i\}$ Mettiamo l'oggetto x_i più tutti gli oggetti letterali tali per cui quando $x_i = F$ il letterale è falsificato.
 - $\forall c_j \in \phi$ abbiamo le coppie $\{c_j, p_{j_1}\}, \{c_j, p_{j_2}\}, \{c_j, p_{j_3}\}$; Questi insiemi della partizione possono essere selezionati quando la clausola c_j viene verificata dal letterale corrispondente.

ie:

$$\phi = \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{c_1} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_4)}_{c_2}$$

$$U_\phi = \{x_1, x_2, x_3, x_4, c_1, c_2, P_{1,1}, P_{1,2}, P_{1,3}, P_{2,1}, P_{2,2}, P_{2,3}\}$$

$$F_\phi = \{\{P_{1,1}\}, \{P_{1,2}\}, \{P_{1,3}\}, \{P_{2,1}\}, \{P_{2,2}\}, \{P_{2,3}\}, \leftarrow \text{singleton}\}$$

$$\begin{array}{ll} T_{1,T} = \{x_1, P_{2,1}\} & T_{2,T} = \{x_2, P_{1,2}\} \\ T_{1,F} = \{x_1, P_{1,1}\} & T_{2,F} = \{x_2, P_{2,2}\} \{c_1, P_{1,1}\} \{c_2, P_{1,2}\} \{c_1, P_{1,3}\} \\ T_{3,T} = \{x_3\} & T_{4,T} = \{x_4\} \quad \{c_2, P_{2,1}\} \{c_2, P_{2,2}\} \{c_2, P_{2,3}\} \\ T_{3,F} = \{x_3, P_{1,3}\} & T_{4,F} = \{x_4, P_{2,3}\} \end{array}$$

x_3 e x_4 non comparendo mai in forma negata, dentro le clausole, compariranno nelle T da soli.

Mostriamo che la formula ϕ è soddisfacibile \Leftrightarrow dentro F ci sta un partizionamento di U .

\Rightarrow) Se ϕ è soddisfacibile, allora in F_ϕ c'è una partizione di U_ϕ

Supponiamo che ϕ sia soddisfacibile, allora esiste un assegnamento di verità σ per le variabili di ϕ tale che $\sigma \models \phi$. Per mostrare che in F_ϕ c'è una partizione di U_ϕ la dobbiamo costruire e la costruiamo partendo da σ :

Per la partizione di U_ϕ prendiamo questo:

- | | |
|--------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Se $\sigma[x_i] = V \Rightarrow$ prendiamo $T_{i,T}$
2. Se $\sigma[x_i] = F \Rightarrow$ prendiamo $T_{i,F}$ | 3. Prendiamo anche le $\{c_j, P_{j,k}\}$ tali che il letterale $\lambda_{j,k}$ è reso vero in σ e non appare nei $T_{i,T}$ o $T_{i,F}$.
4. Prendiamo infine i $\{P_{j,k}\}$ rimasti fuori. |
|--------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Tramite 1. e 2. andiamo a coprire gli oggetti x_i più la parte dei $\{P_{j,k}\}$ resi falsi dall'assegnamento.

Tramite 3. copriamo gli oggetti c_j più alcuni oggetti $P_{j,k}$ che non erano entrati tramite 1. e 2.

Quello che può accadere è che altri oggetti $P_{j,k}$ siano rimasti fuori dalla copertura allora noi ce li prendiamo dai singleton uno per uno.

Quindi se la formula è soddisfacibile allora F_ϕ contiene una partizione di U_ϕ .

\Leftarrow) Se F_ϕ contiene una partizione di U_ϕ allora ϕ è soddisfacibile.

Supponiamo che $P \subseteq F_\phi$ sia una partizione di U_ϕ . Quindi dentro a P ci saranno un po' di insiemi che provengono da F_ϕ tra i quali quelli che noi abbiamo chiamato $T_{i,T}$ o $T_{i,F}$, ci saranno un po' di coppie $\{c_j, P_{j,k}\}$, un po' di singleton $P_{j,k}$; insomma c'è qualcosa di vario.

Quello che sappiamo è che P costituisce una partizione di U_ϕ . Per mostrare che ϕ è soddisfacibile mostreremo che è possibile costruire un assegnamento σ che parte da P che soddisfa la formula ϕ :

$$\begin{aligned} \sigma_P[x_i] &= V \text{ se } T_{i,T} \in P \\ \sigma_P[x_i] &= F \text{ se } T_{i,F} \in P \end{aligned}$$

σ soddisfa ϕ ? Supponiamo, per assurdo, che σ_P non soddisfi $\phi \Rightarrow \exists$ una clausola $c_j \in \phi$ in cui tutti i letterali sono falsi in σ_P , ma ciò implicherebbe che in P noi non stiamo coprendo l'oggetto c_j , ma ciò non può essere perché P è una partizione di $U_\phi \Rightarrow$ assurdo $\Rightarrow \sigma \models \phi$.

Da cui VC si riduce a $DS \Rightarrow DS \in NP-Hard$

$$\Rightarrow DS \in NP-complete$$

■

6.3.8 Knapsack (Kn)

Sia $\{1, \dots, n\}$ un insieme di oggetti, per ognuno di loro abbiamo un peso w_i e un valore v_i . Abbiamo poi una soglia W . Vogliamo trovare un sottoinsieme $S \subseteq \{1, \dots, n\}$ tale che:

$$\sum_{i \in S} w_i \leq W, \quad \sum_{i \in S} v_i \text{ è massima}$$

Questo problema sta in NP ? No, perché non è un problema di decisione, la variante di decisione è:

Dato un insieme di oggetti $\{1, \dots, n\}$, per ognuno dei quali abbiamo un peso w_i e un valore v_i . E dati due interi W e K , vogliamo trovare un sottoinsieme $S \subseteq \{1, \dots, n\}$ tale per cui:

$$\sum_{i \in S} w_i \leq W, \quad \sum_{i \in S} v_i \geq K$$

Dimostriamo che $Kn \in NP$ -completo:

3) $Kn \in NP$

Indoviniamo la soluzione polinomialmente, e poi check in tempo polinomiale.

4) $Kn \in NP$ -Hard

$$\begin{array}{ccc}
 EC & \xrightarrow[f]{\leq_P} & Kn \\
 \langle U, F \rangle & & \{1, \dots, n\} \quad \{v_1, \dots, v_n\} \\
 & & \{w_1, \dots, w_n\} \quad W, K \\
 \text{insieme di oggetti,} & & \text{insieme di oggetti} \\
 \text{Famiglia di sottoinsiemi di } U & & \text{relativi pesi e valori} \\
 & & \text{due interi}
 \end{array}$$

L'istanza di Kn che andremo a costruire è particolare e ci permette di semplificarci le cose; nonostante questa riduzione costruisca un'istanza di Kn molto specifica, ciò non toglierà nulla alla difficoltà di Kn se noi siamo, in ogni caso, in grado di mappare EC su Kn .

Nello specifico, nella riduzione avremo che $w_i = v_i$, $\forall i$ e inoltre avremo che $W = K$.

Siccome $W = K$ e $w_i = v_i$ e siccome noi dobbiamo verificare le due sommatorie, il tutto si riduce a:

Ma esiste un insieme delle w_i che, se sommate tra loro, da esattamente W ?

Il Tuttto può partire dalla rappresentazione binaria dei numeri:

Supponiamo $U = \{1, 2, 3, 4\}$ ed $F = \{\{3, 4\}, \{2, 4\}, \{2, 3, 4\}\}$.

L'idea, siccome U ed F devono essere mappati verso una lista di interi e un numero W , è che otterrò:

$$\begin{array}{c}
 F = \underbrace{\{3, 4\}}_{F_1}, \underbrace{\{2, 4\}}_{F_1}, \underbrace{\{2, 3, 4\}}_{F_1} \\
 \downarrow \quad \downarrow \quad \downarrow \\
 w_1 = (0011)_2 \quad w_2 = (0101)_2 \quad w_3 = (0111)_2
 \end{array}$$

Questa scritta è un'istanza 'no' di EC , perché 1 non è coperto. Però l'idea è questa, ovvero che noi partiamo da F_i e in base agli oggetti che ci stanno, costruiamo il peso w_i in cui, se c'è l'oggetto j , mettiamo la cifra 1 nella rappresentazione binaria di w_i , mettiamo 0 altrimenti. Otteremo quindi:

$$\begin{array}{ccc}
 w_1 = (0011)_2 & w_2 = (0101)_2 & w_3 = (0111)_2 \\
 \parallel & \parallel & \parallel \\
 3 & 5 & 7
 \end{array}$$

Per W , noi lo otteniamo da U :

$$\begin{array}{c}
 U = \underbrace{\{1, 2, 3, 4\}}_{W=(1111)_2=15}
 \end{array}$$

L'idea di fondo è trasformare gli insiemi dentro F verso numeri che in qualche modo codifichino cosa sta e cosa non sta dentro F_i ; e lo facciamo andando a guardare la rappresentazione binaria dei w_i , se l'oggetto j c'è usiamo la corrispondente cifra 1, se non c'è usiamo la corrispondente cifra 0.

Se noi invece quelle stringhe binarie le interpretassimo semplicemente come stringhe di booleani, staremmo riconducendo l'OR logico di quelle stringhe di booleani a fare la somma di numeri rappresentati in binario, e questo è il mapping che stiamo cercando di ottenere.

Noi però stiamo partendo da un'istanza 'no', e se andiamo a fare la somma:

L'or logico non da la stringa 1111, ma la somma si.

Il problema è il riporto, quindi noi ci dobbiamo inventare dei numeri tali per cui il riporto non ci dia dei problemi.

$$\begin{array}{r} 0 & 0 & 1 & 1 & + \\ 0 & 1 & 0 & 1 & + \\ \hline 0 & 1 & 1 & 1 & = \\ 1 & 1 & 1 & 1 & \leftarrow \end{array}$$

Quindi noi abbiamo un'istanza $U = \{1, \dots, n\}$ e un insieme $F = \{F_1, \dots, F_m\}$, questa cosa dev'essere trasformata in un insieme di numeri w_1, \dots, w_m e in un W . Questo in maniera tale che è possibile selezionare nella lista w_1, \dots, w_m dei numeri la cui somma sia W . Abbiamo visto che ottenere i pesi tramite la codifica binaria dell'appartenenza o meno degli oggetti a F_j è una buona idea, il problema è il riporto. Dobbiamo quindi inventarci dei w_i tali che il riporto non ci da problemi.

Per risolvere questo problema partiamo dall'esempio *fallimentare* di prima, notiamo che in F ci stanno 3 F_i , quindi nella peggiore delle ipotesi avremo tre numeri da sommare. Per essere sicuri che la somma di 3 unità non generi riporto noi interpretiamo questi w_i in base 4, generalizzando li interpretiamo in base $m + 1$, così avremo la certezza di non generare mai riporto. Quindi noi abbvremo che:

$$w_i = \sum_{j \in F_i} 1 \cdot (m+1)^{n-j} \quad w_i \text{ è il numero le cui cifre sono } 0 \text{ e } 1, \text{ e gli } 1 \text{ stanno in corrispondenza degli oggetti che stanno dentro l'insieme } F_i, \text{ però interpretati in base } m+1.$$

$$W = K = \sum_{j=1}^n (m+1)^{n-j} \quad \text{Questo } W = K \text{ è semplicemente il numero in base } m+1 \text{ fatto di tutte cifre } 1$$

Dobbiamo ora dimostrare che un'istanza 'si' di EC viene in questo modo trasformata in un'istanza 'si' di Kn e le istanze 'si' di Kn ottenute in questo modo partivano da istanze 'si' di EC .

\Rightarrow Se $\langle U, F \rangle$ è un'istanza 'si' di EC , \Rightarrow l'istanza $f(\langle U, F \rangle)$ di Kn è un'istanza 'si'.

Supponiamo di star partendo da un'istanza 'si' di EC , l'istanza di Kn non è un'istanza di Kn a caso, è un'istanza di Kn ottenuta con questi numeri molto specifici.

Se partiamo da un'istanza 'si' di $EC \Rightarrow$ dentro F ci stanno dei sottoinsiemi di U che costituiscono una partizione di U . Se prendo le stringhe booleane che rappresentano questi insiemi, quali oggetti ci stanno o no, avremo che queste stringhe booleane, non avranno '1' in comune, sulla stessa colonna e sulla stessa colonna ci sarà almeno un '1'; ciò significa che i rispettivi numeri w_i ottenuti come sopra, sono numeri in base $m+1$ la cui somma è esattamente W perché è il numero in base $m+1$ dove c'è la cifra 1 su tutte le colonne.

Quindi se parto da un'istanza 'si' di EC ottengo un'istanza 'si' di Kn .

\Leftarrow Se $f(\langle U, F \rangle)$ è un'istanza 'si' di Kn \Rightarrow l'istanza $\langle U, F \rangle$ di EC è un'istanza 'si'.

Supponiamo di essere arrivati a un'istanza 'si' di Kn , allora vuol dire che tra tutti i w_i ne esistono alcuni che se sommani fanno W , che è il numero che in base $m+1$ è formato da tutti '1'. Siccome ho al più m w e le sto considerando come numeri in base $m+1$ e sono numeri le cui cifre sono '0' e '1' ma in base $m+1$, io non potrò mai generare un riporto nella somma di m numeri.

Quindi se la loro somma produce un numero $(1 \dots 1)_{m+1}$ e questa somma non ha mai generato riporto, allora vuol dire che in tutte le colonne compare uno e un solo '1'.

Quindi se sono arrivato a un'istanza 'si' di Kn era perché partivo da un'istanza 'si' di EC .

Da cui EC si riduce a $Kn \Rightarrow Kn \in NP\text{-Hard}$

$\Rightarrow Kn \in NP\text{-complete}$ ■

E poiché Kn si può esprimere come un problema di programmazione lineare intera, allora tutta la $PLI \in NP\text{-Hard}$.

6.4 TEOREMA DI COOK

Abbiamo detto che un linguaggio L è *NP-Hard* se $\forall L' \in NP, L' \leq_P L$. Però non è quello che abbiamo fatto finora, per mostrare che *Knapsack*, *Exact Cover*, Colorabilità, vertex cover, *Clique*, *3Sat*, ... noi abbiamo usato una proprietà delle riduzioni, ovvero la transitività delle riduzioni polinomiali che ci diceva che se A e B sono due problemi con $A \in NP\text{-Hard}$ e, inoltre, $A \leq_P B \Rightarrow B \in NP\text{-Hard}$.

L'inghippo di questa proprietà, che ci è stata comodissima perché è stata alla base di tutte quelle dimostrazioni di complessità viste finora, è che noi abbiamo bisogno di un problema $A \in NP\text{-Hard}$, ...

$$Kn \geq_P EC \geq_P 3Sat \geq_P Sat \geq_P ???$$

Per dimostrare che $Sat \in NP\text{-Hard}$ non possiamo usare questo trucco, perché non abbiamo altri linguaggi *NP-Hard*. Potremmo pure ridurre $Sat \leq_P Kn$ ma avremmo poi una riduzione circolare; quindi, avremmo che la complessità di questi due problemi è equivalente ma niente più, non avremmo la *hardness*.

Dobbiamo quindi usare la definizione iniziale, cioè, per dimostrare che $Sat \in NP\text{-Hard}$ dobbiamo dimostrare che ogni linguaggio $L \in NP$ si riduce a Sat . Che è una bella rogna, perché ci dobbiamo inventare una riduzione da tutti i linguaggi $\in NP$ che mappino su Sat .

Ad occhio e croce quanti sono i linguaggi $\in NP$? Sono infiniti, quindi trovare infinite riduzioni non è banale. Ci dobbiamo inventare un *trucco*, dobbiamo far vedere che, preso qualsiasi linguaggio $\in NP$ (qualsiasi: *Kn*, *EC*, *VC*, *Col*, *3Sat*, *Clique*, *IS*, problemi sulla teoria dei giochi, ...) si può mappare su *Sat*.

Questo risultato è dovuto a Stephen Cook e prende il nome di **Teorema di Cook**:

Cerchiamo prima di focalizzare la nostra attenzione su "come" debba essere organizzata questa riduzione:

$$\begin{array}{ccc} L & \xrightarrow[f]{\leq_P} & Sat \\ w & \xrightarrow{\phi_w} & CNF \\ Stringa & & \\ generica & & \end{array}$$

Ci chiediamo *che cos'è un'istanza...* di L ? È una stringa, possiamo dire qualcosa di questa stringa? È un grafo? È un'istanza di knapsack? Potrebbe essere qualsiasi cosa, non lo sappiamo, è una stringa sui simboli del linguaggio.

Dobbiamo tradurre questa cosa su di un'istanza di *Sat* che è una formula booleana in formato *CNF*.

Dobbiamo inventarci una funzione f calcolabile in tempo polinomiale, che trasforma w (di natura sconosciuta) in una formula ϕ_w

Che cos'è un'istanza 'sì' del linguaggio L ? È una stringa di simboli che appartiene al linguaggio.

Noi dobbiamo mostrare che, preso un linguaggio L a caso che proviene da NP , esiste un modo di trasformare le istanze di L , qualsiasi, in formule. Prendiamo delle stringhe w , di cui non sappiamo niente, tali per cui, se $w \in L \Rightarrow \phi_w \models Sat$ e se $w \notin L \Rightarrow \phi_w \not\models Sat$.

Finora son tutte cose che abbiamo già visto, c'è un pezzo strano, ovvero che noi non sappiamo cosa sia L .

Allora, quello che noi ci proponiamo di far vedere è che se L è un qualsiasi linguaggio di $NP \Rightarrow \exists$ una riduzione. Non saremo in grado di dare il dettagli della riduzione, perché non sappiamo chi è L . Però siamo in grado di mostrare che una tale riduzione c'è; daremo quindi uno scheletro di riduzione abbastanza preciso che ci permetta di trasformare una qualsiasi istanza di un qualsiasi linguaggio in NP verso *Sat* e che quindi ci mostra che basta che $L \in NP \Rightarrow L \leq_P Sat$.

Possiamo fare un'assunzione su L , ovvero che siccome $L \in NP \Rightarrow \exists$ una macchina M non deterministica che in tempo polinomiale decide L .

Assumiamo che il *running time* di M sia $p(n)$.

La nostra riduzione mirerà a trasformare w in una formula ϕ_w che mima il comportamento di M su w e ϕ_w sarà soddisfacibile $\Leftrightarrow M$ accetta w . (NB f non prende in input M , ma f è progettata su M)

La funzione che dobbiamo tirare fuori è in *CNF* e per semplificarcici il compito, siccome le formule in *CNF* hanno una struttura abbastanza limitata, ci servono un po'di regole di trasformazione di funzioni booleane, in maniera tale che sappiamo che quel tipo di funzione può essere facilmente trasformata in *CNF*.

> Distributività

$$(A_1 \wedge \dots \wedge A_n) \vee (B_1 \wedge \dots \wedge B_m) \equiv (A_1 \vee B_1) \wedge \dots \wedge (A_1 \vee B_m) \wedge (A_2 \vee B_1) \wedge \dots \wedge (A_n \vee B_m)$$

Nota: questa formula è in *CNF*

> De Morgan

$$\neg(A_1 \wedge \dots \wedge A_n) \equiv \neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n$$

Proprietà delle implicazioni:

$$\circ \quad \phi \Rightarrow \psi \equiv \neg \phi \vee \psi$$

$$\circ \quad \phi \Rightarrow \psi_1 \wedge \psi_2 \equiv (\phi \Rightarrow \psi_1) \wedge (\phi \Rightarrow \psi_2)$$

Esempio:

$$\begin{aligned} A \wedge B &\Rightarrow (C \wedge D) \vee (E \wedge F) \equiv \\ A \wedge B &\Rightarrow (C \vee E) \wedge (C \vee F) \wedge (D \vee E) \wedge (D \vee F) \equiv \\ (A \wedge B \Rightarrow C \vee E) \wedge (A \wedge B \Rightarrow C \vee F) \wedge (A \wedge B \Rightarrow D \vee E) \wedge (A \wedge B \Rightarrow D \vee F) \equiv \\ (\neg A \vee \neg B \vee C \vee E) \wedge (\neg A \vee \neg B \vee C \vee F) \wedge (\neg A \vee \neg B \vee D \vee E) \wedge (\neg A \vee \neg B \vee D \vee F) \end{aligned}$$

Questa qua è tutta una trasformazione, ad esempio, che ci permette di listare le funzioni come implicazioni e quindi ci occuperemo di trascrivere la riduzione come implicazione.

$$\begin{array}{ccc} L & \xrightarrow[L \leq_P Sat]{f} & \phi_w \\ w & \xrightarrow{f} & \end{array}$$

Quindi noi dobbiamo andare a trasformare w in ϕ_w in cui andremo a mimare il funzionamento di M su w , che vuol dire che dentro ϕ_w ci saranno delle proposizioni booleane che in dipendenza se sono *V/F* ci dicono cosa sta succedendo nella computazione di M su w .

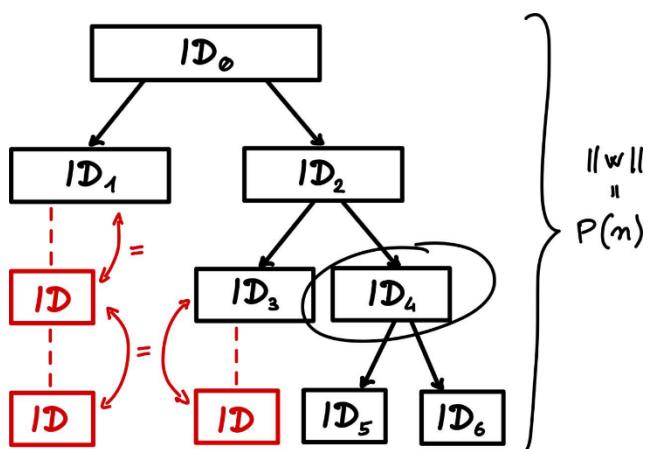
Concentriamoci prima sul funzionamento della macchina M :

I rami di computazione di questa macchina possono avere lunghezze differenti, cioè per alcune scelte la macchina potrebbe impiegare più tempo che non su altre scelte, l'unico vincolo che sappiamo è che la macchina ci mette al più $p(n)$ passi. Dove $n = \|w\|$.

Faremo una serie di assunzioni su M così da semplificare la traduzione del suo funzionamento all'interno di ϕ_w , le assunzioni che faremo sono:

- M non scrive mai \emptyset
- M ha un nastro semi-infinito (ha un punto di partenza, la cella n°0)
- M ci mette esattamente $p(n)$ passi su tutti i rami; cioè l'albero ha delle ID_{extra} così che tutti i branch abbiano la stessa lunghezza, queste config aggiuntive sono copie delle precedenti.

La macchina ha un atteggiamento particolare, una volta che entra in una configurazione finale M continua a ciclare fino a quando fa esattamente $p(n)$ passi.



Le variabili booleane che noi utilizzeremo sono queste qua:

- › $q_{i,k}$ che vale $V \Leftrightarrow$ al passo di computazione i la macchina M si trova allo stato q_k .
 $q_{3,8} = V$ sse al 3° passo la macchina sarà nello stato 8.
 - › $h_{i,j}$ che vale $V \Leftrightarrow$ al passo di computazione i la testina di M si trova sulla j -esima cella del nastro.
 $h_{3,6} = V$ sse al 3° passo la macchina avrà la testina alla cella n°6 (cosa impossibile in 3 passi cella 6)
 - › $t_{i,j,l}$ che vale $V \Leftrightarrow$ al passo i la cella j -esima del nastro contiene il simbolo $\alpha_l \in \Gamma$
 $t_{3,6,2} = V$ quando al terzo passo, in sesta posizione del nastro ci sta il simbolo α_2

Quante sono queste variabili?

$$\rightarrow q_{i,k}, \begin{cases} 0 \leq i \leq p(n) \\ 0 \leq k \leq r \end{cases} \quad \rightarrow h_{i,j}, \quad 0 \leq i, j \leq p(n) \quad \rightarrow t_{i,j,l}, \begin{cases} 0 \leq i, j \leq p(n) \\ 0 \leq l \leq S \end{cases}$$

dove r è il numero fissato degli stati
dove $S = \|\Gamma\|$ fissato.

Queste variabili le useremo per anadre a inventarci la formula ϕ_w :

Siccome ϕ_w deve mimare il funzionamento di M su w tramite queste variabili proposizionali, M sarà fatta da una serie di pezzi:

- il primo pezzo, che chiameremo C “Consistency” sarà il pezzo che si occupa della consistenza dell’assegnamento di verità alle variabili. Cioè noi vogliamo essere sicuri che non può accadere che $q_{2,3} = V$ e $q_{2,4} = V$, che starebbe a significare che al secondo passo la macchina sta nello stato 3 e nello stato 4, ci sarà quindi un pezzo di formula che si assicurerà che:
 - fissato un passo la macchina sta in un solo stato,
 - fissato il passo la testina sta in una sola cella
 - fissato passo e numero di cella sul nastro ci sta un solo simbolo.Quindi ci sarà un pezzo della formula che codificherà questo; cioè, imporrà questi vincoli a come possono essere assegnati i valori di verità alle variabili.
 - Un pezzo S “Start”, si deve occupare di codificare il fatto che la macchina viene accesa nello stato iniziale e sul nastro ci sia la stringa w .
 - Avremo un pezzo N “Next step” che si occuperà di codificare che la formula descrive dei passi sensati, che quindi i valori di verità assegnati a tutte quelle variabili che ci siamo inventati servano a replicare che la macchina veramente funziona come dovrebbe funzionare.
 - Infine, la macchina transisce in una configurazione accettante, F “Final step”.

$$\phi_w = C \wedge S \wedge N \wedge F$$

Consistency C:

$$C = \bigwedge_{\substack{i,k,k' \\ : k \neq k'}} (q_{i,k} \Rightarrow \neg q_{i,k'}) \wedge \bigwedge_{\substack{i,j,j' \\ : j \neq j'}} (h_{i,j} \Rightarrow \neg h_{i,j'}) \wedge \bigwedge_{\substack{i,j,l,l' \\ : l \neq l'}} (t_{i,j,l} \Rightarrow \neg t_{i,j,l'})$$

Se $q_{i,k}$ è vero, allora $q_{i,k'}$ dev'essere falso. (e questa cosa vale per tutte, $q_{i,k}$, $h_{i,j}$ e $t_{i,j,l}$). Queste clausole sono tutte polinomiali.

C sta imponendo che la macchina a ogni passo sta in al più uno stato, la testina ogni passo sta in al più una posizione e che ogni cella del nastro, per ogni passo, contiene al più un simbolo.

Start S: S deve codificare che all'inizio la macchina si trova nello stato iniziale, che sul nastro ci sta la stringa w , per la lunghezza di w , e poi notiamo in $t_{i,j,l}$ che j è compreso tra 0 e $p(n)$, ciò significa che noi stiamo andando a guardare un nastro la cui lunghezza è $p(n)$, e questo è sufficiente perché la macchina compie $p(n)$ passi.

Quindi tutto quello che ci interessa della computazione della macchina sta tra la cella 0 e $p(n)$, perciò quando codifichiamo la configurazione iniziale della macchina nella porzione iniziale di nastro ci sta w e in tutto il resto che ci sta? Ci sta λ e lo scriviamo così:

$$\begin{aligned} S = q_{0,0} \wedge h_{0,0} \wedge t_{0,0,w_0} \wedge t_{0,1,w_1} \wedge t_{0,1,w_2} \wedge \dots \\ \wedge t_{0,n-1,w_{n-1}} \\ \wedge t_{0,n,\lambda} \wedge t_{0,n+1,\lambda} \wedge \dots \wedge t_{0,p(n),\lambda} \end{aligned}$$

In ordine: $q_{0,0}$ per indicare che la macchina a passo 0 sta nello stato 0; la testina a passo 0 sta su cella 0;

a passo 0 le celle del nastro da 0 a w_{n-1} (dove $n = \|w\|$) e tutte le altre, fino a $p(n)$, mettiamo λ .

Next Step N: Dobbiamo codificare nella funzione che i next passi seguono la funzione di transizione:

Sul nastro possiamo distinguere due aree:

1) Tutte quelle celle in cui non c'è la testina:

Da un passo all'altro, il contenuto di queste celle non varia

2) La cella dove c'è la testina:

Invece qua l'pezzo di nastro dove c'è la testina cambia secondo la funzione di transizione.

Inerzia)

Le parti di nastro lontane dalla testina rimangono invariate da un passo all'altro.

$$N^I = \bigwedge_{i,j,l} (\neg h_{i,j} \wedge t_{i,j,l} \Rightarrow t_{i+1,j,l})$$

Per tutte le triple i,j,l se la testina al passo i non sta in posizione j e, inoltre, al passo i in posizione j abbiamo il simbolo l , allora al passo $i+1$ in posizione j il simbolo dev'essere lo stesso.

Testina)

Quello che metteremo in N^H è una generalizzazione di quanto segue: Per ogni pezzo della funzione di transizione del tipo:

$$\delta(q_k, \alpha_l) = \{(q_{k'}, \alpha_{l'}, \rightarrow) \vee (q_{k''}, \alpha_{l''}, \leftarrow)\}$$

δ ha questa struttura, per ogni coppia stato-simbolo la macchina andrà in uno stato, scriverà un certo simbolo e muove la testina; quindi, noi dobbiamo andare a codificare dentro un pezzo di formula questa cosa qua:

$$N^H = \bigwedge_{i,j,l} \left(q_{i,k} \wedge h_{i,j} \wedge t_{i,j,l} \Rightarrow \left(\begin{array}{l} (q_{i+1,k'} \wedge h_{i+1,j+1} \wedge t_{i+1,j,l'}) \vee \\ (q_{i+1,k''} \wedge h_{i+1,j-1} \wedge t_{i+1,j,l''}) \end{array} \right) \right)$$

Se al passo i -esimo la macchina sta nello stato k , la testina sta nella cella j e in quella cella c'è il simbolo l , allora andiamo nello stato k' , scriviamo il simbolo l' e spostiamo a destra la testina, oppure andiamo nello stato k'' , scriviamo il simbolo l'' e spostiamo la testina a sinistra ($j-1$).

(questo dalla δ di cui sopra)

Padding

Ovvero tutti quei passi finti che la macchina deve compiere per fare esattamente $p(n)$ passi.

$$N^P = \bigwedge_{i,j,l} (q_{i,F} \wedge h_{i,j} \wedge t_{i,j,l} \Rightarrow q_{i+1,F} \wedge h_{i+1,j} \wedge t_{i+1,j,l})$$

Se la macchina al passo i entra nello stato finale F e la testina è in posizione j e sul nastro c'è l , allora al passo $i + 1$ la macchina sta nello stesso stato, la testina non cambia di posizione, e il contenuto del nastro è lo stesso, questo per ogni i, j, l . Questo però gestisce solo gli stati accettanti vediamo la formula per gli altri:

$$\bigwedge_{i,j,l} \bigwedge_{\substack{q_k \in Q, \alpha_l \in \Gamma: \\ \delta(q_k, \alpha_l) = \emptyset}} (q_{i,k} \wedge h_{i,j} \wedge t_{i,j,l} \Rightarrow q_{i+1} \wedge h_{i+1,j} \wedge t_{i+1,j,l})$$

È uguale a quella sopra ma è definita per tutte le coppie stato-simbolo in cui la macchina si blocca.

$$N = N^I \wedge N^H \wedge N^P$$

Final Step F : all'ultimo passo (è il $p(n)$ -esimo) lo stato è accettante:

$$F = q_{p(n),F}$$

Quindi la formula ϕ_w è la congiunzione di tutta questa roba che abbiamo scritto, C, S, N e F .

Ultime considerazioni:

f prende w e sputa fuori ϕ_w che è questa formula mostruosa che, sebbene sia enorme, è comunque di taglia polinomiale.

Questa formula descrive, tramite delle variabili proposizionali opportune, il comportamento di una macchina M fissata, che decide se w appartiene a L o meno.

Per come è stata definita la formula, in cui stiamo andando a imporre dei vincoli a queste variabili proposizionali tali per cui gli assegnamenti V/F deve per forza replicare cosa fa la macchina M su w .

Di conseguenza, per come è stato definito ϕ_w , $w \in L \Leftrightarrow M \models w \Rightarrow \phi_w$ è soddisfacibile.

Questa formula è in CNF , perché abbiamo usato solo implicazioni, e quindi un qualsiasi linguaggio di $L \in NP$ che non conosciamo ammette una riduzione di questo tipo verso Sat ; da cui $Sat \in NP$ -Hard.

Abbiamo già visto che $Sat \in NP$, perché basta indovinare l'assegnamento di verità

$$\Rightarrow Sat \in NP\text{-complete}$$

■

6.5 COMPLEMENTO DI NP

Abbiamo visto nelle lezioni precedenti che la classe NP può essere definita in diversi modi equivalenti. Uno di questi è che un linguaggio $L \in NP$ se l'appartenenza di una stringa w a L è testimoniata dall'esistenza di un "piccolo" certificato verificabile in tempo polinomiale.

In termini di MdTN, ciò significa che esiste un MdTN M , che richiede un numero polinomiale di passi, tale che con input $w, w \in L \Leftrightarrow \exists$ un cammino nell'albero di calcolo M che termina nello stato accettante.

Consideriamo ora il seguente linguaggio, che è il complemento di Sat :

$$UnSat = \{\phi \mid \phi \text{ è una formula booleana in CNF che non è soddisfacibile}\}$$

$UnSat \in NP$? per stare in NP deve esistere una MdTN in grado di decidere il problema in tempo polinomiale

Noi indoviniamo un assegnamento di verità per le variabili nella formula, verifichiamo poi che *non* soddisfa la formula e rispondiamo 'sì'. Questa macchina decide $UnSat$?

Se noi definiamo:

$$Taut = \{\phi \mid \phi \text{ in CNF che sono tautologie}\}$$

Una tautologia è una formula per la quale qualsiasi assegnamento di verità la soddisfa. Quindi una macchina che faccia un guess di un assegnamento per la formula ϕ , verifica che questo assegnamento non soddisfa ϕ , in realtà sta decidendo \overline{Taut} .

In effetti questo problema è particolare, noi non è che possiamo rispondere 'sì' se troviamo un assegnamento di verità che non soddisfa la formula, noi possiamo rispondere "si questa formula ϕ non è soddisfacibile" se tutti gli assegnamenti alle variabili booleane che costituiscono ϕ non soddisfano la formula, quindi in qualche modo noi vorremmo che tutti i path di computazione di questa MdTN siano rifiutanti, per poter dire di 'sì'. Si, questa formula non è soddisfacibile quando sappiamo che in qualsiasi modo noi proviamo a guessare, non siamo mai in grado di soddisfarlo.

Questo problema, che è un po' particolare, è un esempio di una classe diversa da NP , che è la classe

$$CO-NP = \{L \mid \overline{L} \in NP\}$$

Cioè, è la classe dei linguaggi L i cui complementi stanno in NP . Ci stanno tutta una serie di problemi interessanti, però sostanzialmente questa classe è definita come i complementi dei linguaggi in NP .

Quindi ad esempio, $UnSat \in CO-NP$ perché $\overline{UnSat} = Sat \in NP$.

Una cosa importante da specificare è che:

$$CO-NP \neq \overline{NP}$$

Perché \overline{NP} sarebbe tutto ciò che sta fuori NP , tra cui RE , quello fuori RE ...

Quindi $CO-NP$ non è il complemento di NP , esattamente come $CO-RE$ non è il complemento di RE .

$CO-NP$ è definito in maniera similare a $CO-RE$ ed è l'insieme dei linguaggi i cui complementi stanno in NP .

L'intuizione che possiamo avere, per i linguaggi in NP è sostanzialmente speculare a quello che abbiamo per NP ; un linguaggio sta in $CO-NP$ se le sue istanze 'no' sono caratterizzate da un certificato.

In NP abbiamo i linguaggi le cui istanze 'sì' ammettono un certificato per dire di 'sì'

In $CO-NP$ ci stanno i linguaggi le cui istanze 'no' ammettono un certificato per dire di 'no', cioè noi siamo in grado di rispondere 'no' guessando qualcosa

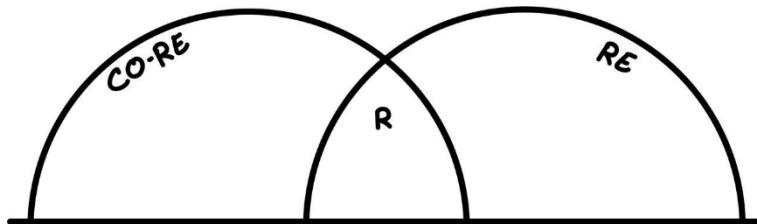
Prendiamo $UnSat$, data una formula booleana ϕ in CNF dobbiamo stabilire se sia non soddisfacibile o meno. Per rispondere "no, non è non soddisfacibile" indoviniamo un assegnamento che la soddisfa.

Quindi questo è il senso, poiché $CO-NP$ è definita in questo modo particolare, noi possiamo rimodulare la definizione intuitiva dei linguaggi in $CO-NP$, come quei linguaggi per i quali abbiamo un certificato conciso per rispondere di no.

Quindi in NP ci stanno quei linguaggi che ammettono un certificato conciso per rispondere di 'si', in $CO-NP$ ci stanno quei linguaggi che ammettono un certificato conciso per rispondere di 'no'.

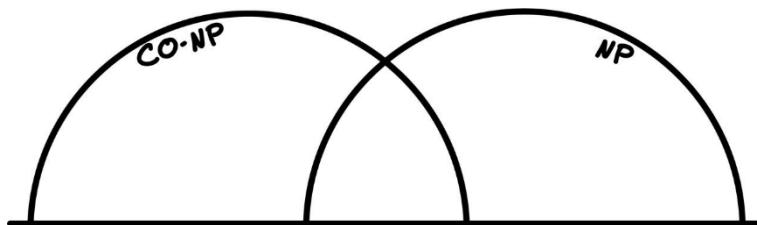
Vediamo la relazione tra $CO-NP$ e NP , un po' allo stesso modo di quando vedemmo la relazione tra $CO-RE$ ed RE , ricordiamo che riuscimmo a dire che:

- $CO-RE \neq RE$
- $CO-RE \cap RE = R$



Ora ci chiediamo che relazione c'è tra $CO-NP$, NP e P .

Ora abbiamo quest'altra situazione:



E vogliamo stabilire qual è la loro relazione ed eventualmente dove stia P .

A differenza delle relazioni tra $CO-RE$, RE ed R che noi conosciamo, le relazioni tra $CO-NP$, NP e P sono aperte, cioè nessuno sa niente. Noi non sappiamo se NP e $CO-NP$ siano due classi distinte o meno, ovvero

$$\text{?} \\ CO-NP = NP$$

Questo è un problema aperto, esattamente come P vs NP , esattamente come noi non sappiamo nulla riguardo l'inclusione stretta tra P ed NP , noi non sappiamo nulla nella relazione fra $CO-NP$ ed NP , cioè noi non lo sappiamo se queste classi sono distinte o meno, nessuno non è mai riuscito a dimostrare che queste due classi siano uguali ma nessuno non è mai riuscito a dimostrare che siano distinte. Abbiamo dei risultati che mostrano un po' come siano legate $CO-NP$ ed NP però più di questo non siamo in grado di dire.

THM: $NP = CO-NP \Leftrightarrow \exists L \in NP\text{-complete} : L \in CO-NP$

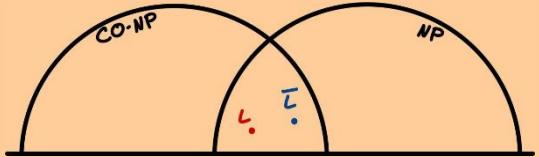
Proof:

$\Rightarrow)$ se $NP = CO-NP$ $\Rightarrow \exists L : L \in NP\text{-complete} \wedge L \in CO-NP$
Se L è $NP\text{-complete}$, per definizione $L \in NP$, ma se $NP = CO-NP \Rightarrow L \in CO-NP$

$\Leftarrow)$ se $\exists L : L \in NP\text{-complete} \wedge L \in CO-NP$ $\Rightarrow NP = CO-NP$

Facciamo due considerazioni

- > Se $L \in NP\text{-complete}$ $= \boxed{L \in NP \Rightarrow \bar{L} \in CO-NP}$
- > Se $\boxed{L \in CO-NP}$ $= \boxed{\bar{L} \in NP}$



Ci troviamo nella situazione dello schema qui sopra

Mostriamo che $NP \subseteq CO-NP$

Sia $L' \in NP$ un linguaggio qualsiasi, mostriamo che $L' \in CO-NP$ e mostriamolo facendo Vedere $\bar{L}' \in NP$
Siccome

$$L' \in NP \wedge L \in NP\text{-complete} \Rightarrow L' \leq_P L$$

Se $L' \leq_P L$ possiamo dire

$$L' \leq_P L \Rightarrow \exists f : \Sigma^* \rightarrow \Sigma^* \text{ t.c. } \forall w, w \in L' \Leftrightarrow f(w) \in L$$

Siccome questa relazione vale come un sse, noi la possiamo scrivere:

$$\forall w, w \notin L' \Leftrightarrow f(w) \notin L$$

Da cui

$$\forall w, w \in \bar{L}' \Leftrightarrow f(w) \in \bar{L}$$

Questa cosa qua ci dice che

$$\exists \bar{L}' \leq_P \bar{L}$$

Noi abbiamo che $\bar{L} \in NP$, da cui abbiamo che

$$\bar{L}' \leq \bar{L} \in NP \Rightarrow \bar{L}' \in NP \Rightarrow L' \in CO-NP \Rightarrow \boxed{NP \subseteq CO-NP}$$

Mostriamo che $CO-NP \subseteq NP$

Sia $L' \in CO-NP$ un linguaggio arbitrario, mostriamo che $L' \in NP$

Abbiamo che

$$L' \in CO-NP \Rightarrow \bar{L}' \in NP$$

Siccome

$$L \in NP\text{-complete} \wedge \bar{L}' \in NP \Rightarrow \bar{L}' \leq_P L$$

Per quello che abbiamo detto prima, se $\bar{L}' \leq_P L$ allora noi possiamo considerare anche i complementi
cioè: $L' \leq_P \bar{L}$

Ma

$$\bar{L} \in NP \Rightarrow L' \in NP \Rightarrow CO-NP \subseteq NP$$

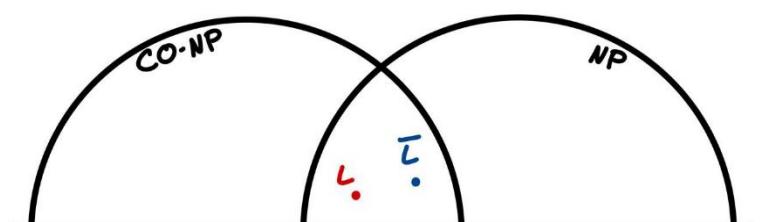
$$\Rightarrow NP = CO-NP$$

■

Abbiamo quindi la caratterizzazione di questo teorema che in realtà non ci dice moltissimo, L'intuizione è

*"se mai avessimo un problema $NP\text{-complete}$ che fa parte di $CO-NP$,
allora queste due classi sarebbero la stessa cosa"*

Però nessuno è mai riuscito a collocare un
problema $NP\text{-complete}$ dentro $CO-NP$,
per questa ragione noi supponiamo che le
due classi siano distinte, e che la situazione
sia quella qui a fianco.

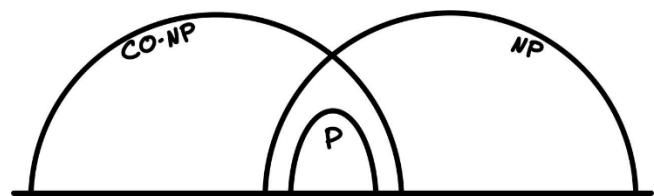


Quale relazione c'è tra P , $CO-NP$ e NP ?

Il risultato che abbiamo è questo in figura, noi non sappiamo se l'intersezione tra $CO-NP$ e NP sia esattamente P o un sottoinsieme di P .

Siccome noi abbiamo che se ci fosse un problema

NP -completo in NP allora le due classi sarebbero equivalenti e nessuno ha mai dimostrato che questo sia il caso, allora noi abbiamo un insieme di problemi che per essere in NP e in $CO-NP$ non possono essere NP -completi, quindi noi ci attendiamo che nell'intersezione tra $CO-NP$ e NP ci siano linguaggi "non tosti" e in effetti si, ci sono tutti i linguaggi P , ovvero linguaggi non NP -completi che stanno in $CO-NP$ e in NP .



THM: $P \subseteq NP \cap CO-NP$

Proof:

Sia L un linguaggio generico di P , poiché $L \in P$ e $P \subseteq NP \Rightarrow L \in NP$.

Consideriamo il linguaggio \bar{L} . Siccome $L \in P \Rightarrow \exists$ MdT deterministica M che in tempo polinomiale decide L , se noi prendiamo M , e gli andiamo a scambiare gli stati accettanti e quelli non accettanti, allora M andrà ad accettare $\bar{L} \Rightarrow \bar{L} \in P$

(per questa ragione P , che è una classe deterministica, come tutte le classi deterministiche, sono chiuse su complemento)

Da $\bar{L} \in P \Rightarrow \bar{L} \in NP \Rightarrow L \in CO-NP$

E questo ci permette di dimostrare che $P \subseteq NP \cap CO-NP$,

■

Però sul fatto se $NP \cap CO-NP$ sia esattamente P , questo è ancora un problema aperto.

Quindi a differenza di RE , $CO-RE$ e R , per i quali noi siamo in grado di stabilire delle relazioni precise tra queste classi, tra NP , $CO-NP$ e P non lo sappiamo.

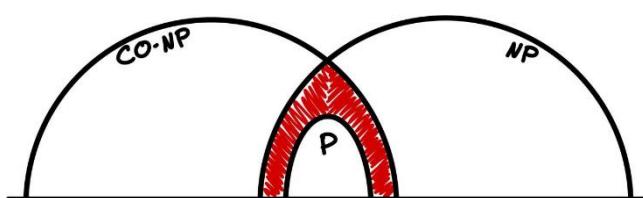
Cosa succederebbe se $P = NP$? Cosa potremmo dire della loro relazione con $CO-NP$?

Allora $CO-NP = NP = P$ tutto collasserebbe su P .

Però un'altra cosa che potrebbe accadere è avere $NP = CO-NP$ che però non coincidono a P .

Quello che **si ritiene** sia lo stato dei fatti, è che P sia un sottoinsieme stretto di $NP \cap CO-NP$ (è un'ipotesi) E un'altra cosa che ipotizziamo è che NP sia distinto da $CO-NP$ (ma anche questa la ipotizziamo)

Nella classe $CO-NP$ ci stanno i problemi tosti in $CO-NP$, quindi esistono i problemi $CO-NP-Hard \ni UnSat$. I problemi $CO-NP-Hard$, similmente ai problemi $NP-Hard$ sono i problemi almeno difficili quanto tutti i problemi di $CO-NP$; e un certo problema L è $CO-NP-Hard$ se \bar{L} è $NP-Hard$.



Noi crediamo che la relazione tra le classi $CO-NP$, NP e P sia questa in figura, di conseguenza c'è tutta l'area dell'intersezione che non è P nella quale si trovano problemi che non sono né difficili per quello che abbiamo detto; quindi, non siamo riusciti a

mostrare che siano problemi NP -completi e reputiamo che nemmeno lo siano, perché stando lì in mezzo, stanno sia in NP che in $CO-NP$, di conseguenza noi non ci attendiamo che siano NP -completi.

Però sono dei problemi per i quali non siamo nemmeno riusciti a trovare degli algoritmi polinomiali; quindi, c'è un limbo di problemi che non sono né tosti né semplici; e su questi problemi si basano i sistemi di crittografia corrente, uno di questi è il problema della fattorizzazione:

Il problema della fattorizzazione “*dato un numero, produrre una fattorizzazione in numeri primi*” sta in *NP*?
 No, perché non è un problema di decisione.

Formuliamone la sua versione decisionale:

$$\text{Factor} = \{\langle n, k \rangle | n \text{ è un intero naturale che ha almeno un fattore primo } p \leq k\}$$

Ad esempio:

$$\begin{aligned} 175 &= 5 \cdot 6 \cdot 7 \\ \langle 175, 6 \rangle &\in \text{Factor} \\ \langle 175, 4 \rangle &\notin \text{Factor} \end{aligned}$$

THM:	$\text{Factor} \in NP \cap CO-NP$
<i>Proof:</i>	
• $\text{Factor} \in NP$	<p>Guess di un fattore primo $p \leq k$ e poi check che p sia primo e che p divida n. Questa cosa si può fare in tempo polinomiale perché p è minore sia di n che di k, poi n si può dividere per p in tempo polinomiale, e il test che p sia primo si fa in tempo polinomiale deterministico. (n^{12}).</p>
• $\text{Factor} \in CO-NP$	<p>Lo mostriamo facendo vedere che $\overline{\text{Factor}} \in NP$,</p> $\overline{\text{Factor}} = \{\langle n, k \rangle n \text{ è un naturale i cui fattori primi sono tutti più grandi di } k\}$ $\overline{\text{Factor}} = NP$ <p>Guessiamo tutta la fattorizzazione di n (p_1, p_2, \dots, p_m), ma questi che stiamo guessando sono polinomiali sulla taglia di n, quindi il numero massimo di fattori è 2^n, quindi male che ci va il numero di fattori primi che guessiamo è polinomiale. Quindi sappiamo che m è bounded da un polinomio nella taglia di n. Quanto ci costa testare che siano tutti primi? Ci costa polinomiale perché dobbiamo fare un numero polinomiale di test polinomiali, quindi il check si fa in tempo polinomiale. Poi dobbiamo controllare che il loro prodotto sia n e la moltiplicazione si può fare in tempo polinomiale nella taglia degli operandi. E poi dobbiamo verificare che siano tutti più grandi di k e si fa anche questo in tempo polinomiale. Da ciò abbiamo che $\overline{\text{Factor}} \in NP \Rightarrow \text{Factor} \in CO-NP$</p> $\Rightarrow \text{Factor} \in NP \cap CO-NP$

La cosa interessante è che buona parte dei sistemi crittografici a chiave pubblica che usiamo in questo momento, tipo RSA, si basano sul problema della fattorizzazione e si basano sul fatto che noi assumiamo che fattorizzare dei numeri sia un problema tosto, in realtà noi non lo sappiamo, se sia un problema dentro *P* o no, siccome nessuno ci è mai riuscito, ipotizziamo che non lo sia.

Quello che sappiamo è che, con altissima probabilità, il problema della fattorizzazione non è *NP-completo*, quindi non è un problema tosto di *NP* perché *Factor* sta sia in *NP* che in *CO-NP* di conseguenza se *Factor* fosse uno dei problemi tosti di *NP*, se fosse un problema *NP-completo*, allora avremo che *NP* = *CO-NP* e questo non ci aspettiamo che accada.

6.6 EXP E NEXP

Vediamo altre classi di complessità, le introdurremo soltanto, non vedremo cose troppo dettagliate perché richiederebbe del tempo.

Abbiamo finora parlato dei problemi *NP*, che non è la classe di problemi *Non Polinomiali*, ma sono i problemi **Nondeterministic Polinomiali**. Però non è che le classi di complessità si esauriscano qua, sopra *NP* ci stanno tante classi, alcune che vedremo e altre delle quali ne daremo l'intuizione.

Ci concentriamo sui problemi che possono essere risolti in tempo esponenziale, i problemi Exponential time.

DEF: classe *Exp*

$$\text{Exp} = \bigcup_{c \geq 1} \text{DTIME}(2^{n^c})$$

Sono quei problemi che possono essere risolti da MdTD il cui running time è 2 elevato a un polinomio.

?

Quindi che relazione c'è tra *P* ed *Exp*? Chi sta dentro chi?

$P \subseteq \text{Exp}$, perché quello che sappiamo fare in tempo polinomiale lo sappiamo fare anche in tempo esponenziale.

Ricordiamo che i linguaggi di *P* stanno dentro *Exp*, perché non è che un linguaggio per stare dentro *Exp* si debba risolvere in tempo esponenziale e non di meno, basta che sia risolvibile in tempo esponenziale.

?

Che relazione c'è tra *Exp* e *P*?

Sappiamo dire che **sono diversi**. Non vediamo il THM.

?

Qual è la relazione tra *NP* ed *Exp*?

$NP \subseteq \text{Exp}$, Perché siccome noi possiamo simulare il funzionamento di una MdTN con una crescita temporale di tipo esponenziale, se la MdTN esegue in tempo polinomiale, come le macchine in *NP*, la loro simulazione si fa in tempo esponenziale.

?

Cosa sappiamo dire tra *Exp* e *NP*?

Non sappiamo dire nulla.

?

Similmente noi abbiamo $\text{CO-NP} \subseteq \text{Exp}$, perché?

$\text{CO-NP} \subseteq \text{Exp}$ perché in CO-NP abbiamo i linguaggi i cui complementi stanno in *NP* ma per la relazione di prima $NP \subseteq \text{Exp}$, *Exp* è una **classe deterministica** che è chiusa sotto complemento.

Sia $L \in \text{CO-NP}$, mostriamo che $L \in \text{Exp}$: Se

$$L \in \text{CO-NP} \Rightarrow \bar{L} \in \text{NP} \Rightarrow \bar{L} \in \text{Exp} \Rightarrow L \in \text{Exp}$$

perché *Exp* è chiuso sotto complemento.

DEF: classe *NExp*

$$\text{NExp} = \bigcup_{c \geq 1} \text{NTIME}(2^{n^c})$$

NExp conterrà tutto, perché tutto ciò che può essere fatto con una macchina deterministica può essere fatto con una non-deterministica, quindi:

$$\text{NExp} \supseteq \text{Exp}$$

?

Cosa siamo in grado di dire tra *Exp* e *NExp*?

Non sappiamo dire niente.

Noi abbiamo che $\text{Exp} \subseteq \text{NExp}$, ma non sappiamo se $\text{Exp} = \text{NExp}$,

Sappiamo dire però che $NExp \neq NP$

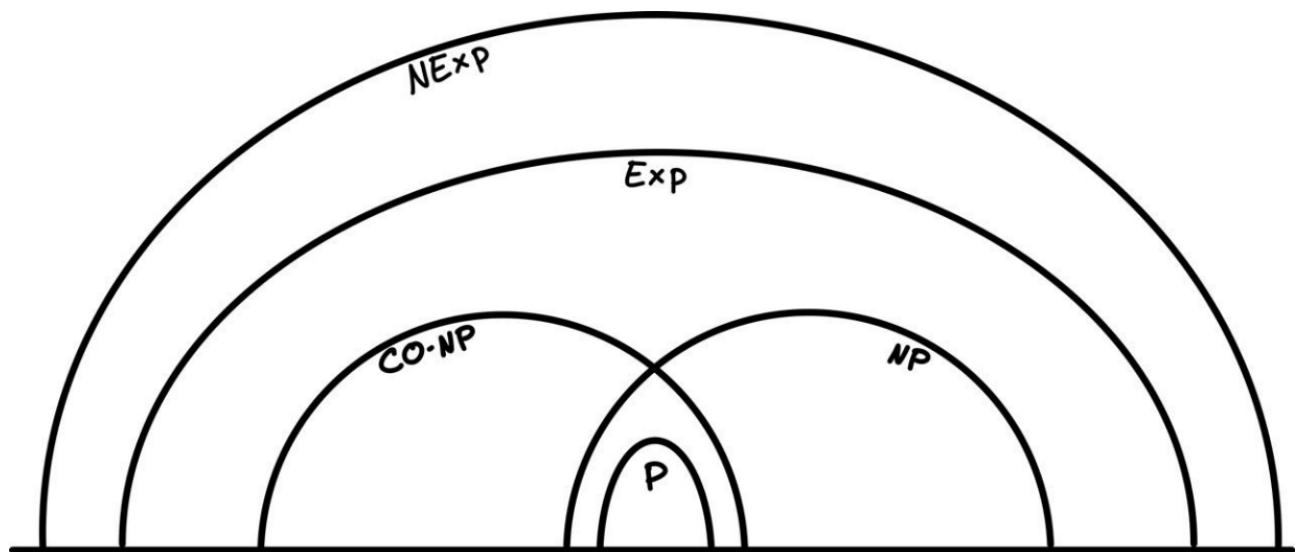
Caratterizzazione di $NExp$:

"è l'insieme dei problemi di decisione che possono essere decisi in tempo esponenziale da macchine deterministiche"

oppure (definito tramite certificati)

"un linguaggio appartiene a $NExp$ se le sue istanze 'si' sono caratterizzate da certificati di taglia esponenziale che sono verificabili da macchine deterministiche in tempo polinomiale nella taglia combinata dell'input e del certificato"

Quindi abbiamo un input con la sua taglia, facciamo un guess di taglia esponenziale, la verifica si fa in tempo polinomiale nella taglia dell'input + la taglia del certificato. Perché non diciamo che la verifica è in tempo esponenziale nella taglia del certificato? Perché, se no la verifica sarebbe in tempo doppiamente esponenziale nella taglia dell'input.



7 COMPLESSITÀ SPAZIALE

? Abbiamo due macchine, una macchina M e una macchina N . La macchina M può usare tempo polinomiale, mentre l'altra macchina N può usare spazio polinomiale. Intuitivamente ce n'è una più potente dell'altra?

Si, N perché la M sarà per forza bounded da spazio polinomiale, quindi ha tempo polinomiale e spazio polinomiale, la seconda invece, che può usare spazio polinomiale, non è limitata dal tempo polinomiale.

Quindi dato un certo vincolo, quel vincolo se dato sul tempo o sullo spazio fa la differenza.

Per prima cosa dobbiamo introdurre un modello differente di MdT, altrimenti definire la complessità spaziale viene un po' complicato.

Parliamo sempre di decisori, cioè MdT che rispondono si/no. Consideriamo una MdT fatta in questo modo: ha un nastro di input sul quale la testina può andare avanti/indietro ma è un nastro di sola lettura; abbiamo poi un nastro aggiuntivo che è il nastro di lavoro ("worktape"), è un nastro infinito, la testina può andare avanti/indietro ed è un nastro di lettura/scrittura. Quindi la brutta copia la MdT la fa sul worktape.

Facciamo tutta questa assunzione perché il vincolo dello spazio usato lo metteremo sul nastro di lavoro.

Questo perché andremo a vedere una classe di complessità in cui imporremo un ridottissimo spazio, gli algoritmi che funzionano in spazio logaritmico, però se noi andassimo a contare anche lo spazio sul nastro di input noi occuperemo sempre spazio lineare. Allora per poter categorizzare questa classe piccolissima che è la classe dei problemi in spazio logaritmico ci serve il nastro aggiuntivo in maniera tale che il bound logaritmico viene dato "sul foglio di brutta".

Introduciamo i concetti di *computation space*, *running space*, ... che avevamo visto per il time:

DEF: (Computation space)

Il **computation space** della macchina M su input w è il numero di celle distinte viste da M sul worktape mentre processa w .

Se M è non-deterministica, definiamo il computation space come il massimo numero di celle visitate dalla macchina su worktape in qualsiasi dei suoi branch di computazione.

DEF: (Space function)

Una funzione $S: \mathbb{N} \rightarrow \mathbb{N}$ è una **space function** se S è strettamente positiva e non decrescente.

DEF: (Running space)

Sia $s(n)$ una space function, il **running space** di M è $s(n)$ se $\forall w$, a parte un numero finito, si ha che il computation space di M su w è bounded da $S(\|w\|)$

Esattamente come avevamo fatto per il tempo, che avevamo definito le classi temporali, possiamo definire le classi di complessità spaziale:

7.1 LOGSPACE L E NL

DEF:

Sia $s(n)$ una space function, denotiamo $D\text{Space}(s(n)) = \left\{ L \middle| \begin{array}{l} \exists M \text{ deterministica t.c. } \mathcal{L}(M) = L \\ \text{e il running space di } M \text{ è } O(s(n)) \end{array} \right\}$

> Definiamo formalmente la classe L (Logspace)

$$L = D\text{Space}(\log_2 n)$$

DEF:

Sia $s(n)$ una space function, denotiamo $D\text{Space}(s(n)) = \left\{ L \middle| \begin{array}{l} \exists N \text{ non deterministica t.c. } \mathcal{L}(N) = L \\ \text{e il running space di } M \text{ è } O(s(n)) \end{array} \right\}$

> Definiamo formalmente la classe P (Non-det. Logspace)

$$NL = D\text{Space}(\log_2 n)$$

Tanto per cambiare noi sappiamo che $L \subseteq NL$ ma non sappiamo se $L = NL$.

Vediamo un esempio di un linguaggio in *Logspace*, i.e., un linguaggio che possiamo decidere con una MdT deterministica e che abbiamo già visto all'inizio del corso

$$\mathcal{L}_{01} = \{0^n 1^n \mid n \geq 0\}$$

?

Come l'avevamo riconosciuto?

Facevamo questo gioco: prendevamo la testina e andavamo sullo 0, lo marcavamo o lo cancellavamo e andavamo in fondo, tornavamo indietro e continuavamo come una pallina da pingpong.

Assumiamo di fare questo genere di lavoro sulla macchina che abbiamo definito prima, quella col worktape. Quello che dovremmo fare sarà sostanzialmente: prendere l'input, ricopiarlo su worktape e iniziare a fare il gioco, perché non possiamo modificare il nastro di input.

?

Quanto spazio occupiamo a fare questo lavoretto?

Lineare, quindi questa macchina usa tanto spazio. Non ci mostra che $\mathcal{L}_{01} \in L$

Possiamo usare una nuova MdT che utilizzi spazio logaritmico, consideriamo una stringa in input w .

1. Se w è vuota \Rightarrow accetta, altrimenti
2. Leggiamo la stringa in input da sinistra a destra, e incrementiamo un contatore nel worktape per ogni 0 incontrato, finché non troviamo un 1.
3. Leggiamo il resto dell'input sinistra a destra e incrementiamo un secondo contatore (memorizzato subito dopo il primo), per ogni 1 incontrato finché non troviamo blank.
4. Confrontiamo i contatori, se sono uguali accettiamo, rifiutiamo altrimenti.

Notiamo che il worktape memorizza soltanto due contatori, nel caso peggiore ogni contatore deve rappresentare il numero $n = |w|$, e se scriviamo i contatori in binario sono necessarie solamente $O(\log_2 n)$ celle per ogni contatore. Poiché utilizziamo un numero costante di contatori (2 in questo caso) la MdT di cui sopra M è tale che $S_M(n) \in O(\log_2 n) \Rightarrow \mathcal{L}_{01} \in \text{Logspace}$

Vediamo ora un esempio di un linguaggio in NL :

$$\text{Reach} = \{(G, s, t) | G \text{ è un grafo orientato tale che esiste un percorso da } s \text{ a } t \text{ in } G\}$$

Abbiamo già visto un algoritmo che decide questo linguaggio, questo algoritmo è basato su di un approccio in ampiezza, e richiede tempo polinomiale. Comunque, questo algoritmo usa una *coda* di nodi, che nel caso pessimo conteneva tutti i nodi del grafo. Quindi lo spazio richiesto dall'algoritmo è lineare sulla taglia dell'input.

Dobbiamo inventarci un algoritmo non-deterministico per risolvere *Reach*

L'idea è che noi *guessiamo*, di volta in volta, solo il passo successivo. Quindi sono in s , *guesso* il nodo dopo. Come faccio a stabilire che non sto *guessando* spazzatura? Verifico che il *guess* sia relativo a un arco presente nel grafo. Una volta che ho questo qua (e sto usando poco spazio perché uso l'identificativo o un puntatore al nodo), cosa faccio? Sono in s , *guesso* il suo successivo, (solo il nome, utilizzo spazio logaritmico per rappresentarlo) dopodiché quando devo *guessare* il passo successivo, saremo in un s' , prendiamo un successivo di s' e vado in s'' . E lo vado a sovrascrivere, di volta in volta (sfruttando questa sovrascrittura ottimizziamo lo spazio). Se esisterà un path, la macchina lo trova, però noi dobbiamo garantirci che questa macchina non entri in loop; perché qua stiamo solo *guessando* il successivo.

Per risolvere questo problema mettiamo un bound sui nodi visti, quindi in memoria noi teniamo: nodo corrente, più un contatore "quanti ne ho visti finora". Se mentre *guesso* i passi successivi il contatore supera il numero dei nodi del grafo allora sono in un loop e quel branch di computazione lo tronco.

Per accettare *guesso* i passi successivi, conto quanti passi faccio e accetto se arrivo a t prima della "tagliola".

```

1. Reach (G,s,t) {
2.   p ← s;
3.   n ← 1;
4.   if (p == t) then ACCEPT;
5.   p' ← Guess di un vertice di G;
6.   if ((p,p')∉ A) then REJECT;
7.   p ← p';
8.   n ← n + 1;
9.   if (n ≤ |v|) then goto 4.
10. else REJECT
11. }
```

Quindi questo algoritmo decide un linguaggio in spazio logaritmico perché abbiamo tre elementi, ognuno dei quali prende spazio logaritmico. Quindi questa è una procedura *Logspace* da parte di una macchina non-deterministica che decide la raggiungibilità in un grafo orientato, da cui $\text{Reach} \in NL$. ■

7.1.1 NL-completezza

❓ Per risolvere *Reach* il non-determinismo è determinante? Possiamo farlo in *Logspace*?

Non si sa. È una di quelle questioni della complessità nella quale algoritmi deterministicici *Logspace* per *Reach* non ce ne stanno, però nessuno ha mai dimostrato che non sia possibile.

Sono stati fatti dei ragionamenti simili alla questione *P vs NP*, cioè nel momento in cui noi sappiamo che $L \subseteq NL$, non sappiamo se $L = NL$. E quindi sono stati inventati i “*problemi più difficili della classe NL*”

Come abbiamo i problemi *NP-completi* abbiamo i problemi *NL-completi* che sono i problemi più difficili della classe *NL*. E *Reach* ∈ *NL-complete* è un problema tosto di *NL* e per questo crediamo che *Reach* ∉ *L*.

Riprendiamo la definizione di *NP-completezza*:

un linguaggio è *NP-completo* se appartiene a *NP* e qualsiasi problema appartenente a *NP* è riducibile polinomialmente a quel linguaggio.

Non possiamo usare la stessa definizione, ne dobbiamo usare una similare; in particolare il problema si annida nella riduzione. Ridurre polinomialmente problemi dentro *NL* è troppo potente, cioè noi potremmo ridurre problemi difficili a problemi facili, se avessimo la possibilità di fare riduzioni in tempo polinomiale.

Lo vedremo più avanti ma $NL \subseteq P$. Quindi fare riduzioni polinomiali per problemi che stanno dentro a *NL* sono riduzioni un po' troppo forti, allora ci si è inventati tutta una serie di riduzioni, le cui più semplici sono le riduzioni *logspace*. Una riduzione *logspace* è una funzione calcolabile in spazio logaritmico. Quindi ci dobbiamo inventare che cosa significhi che una funzione sia calcolabile in spazio logaritmico.

Le funzioni calcolabili in tempo polinomiale erano definite tramite il concetto di trasduttore, ovvero una MdT stramba che aveva un nastro di output. Quindi le funzioni sono calcolate da MdT che hanno un nastro particolare sola scrittura e con la testina che va in un solo verso, nel quale la macchina sputa fuori il proprio output questo perché, quando vogliamo imporre un vincolo spaziale a un trasduttore, se non avesse limiti su quello che può fare sul nastro di output è come se fosse un *foglio di brutta* ma noi vogliamo che abbia il *foglio di bella* in output.

Quindi in un trasduttore così definito, in cui il nastro di input lo possiamo solo leggere, il nastro di output può essere solo scritto, diventa sensato imporre il vincolo di spazio sul nastro di lavoro.

Un tale trasduttore calcola una certa funzione f se per un qualsiasi input che gli diamo, su quella stringa w il trasduttore lascerà a fine calcolo sulla stringa $f(w)$.

DEF: (riduzione logspace)

Siano A e B due linguaggi, esiste una riduzione logspace da A a B , denotato con $[A \leq_L B]$ se esiste una funzione f che mappa stringhe su stringhe, che sia f calcolabile in *logspace* deterministico e inoltre,

$$\forall w, w \in A \Leftrightarrow f(w) \in B$$

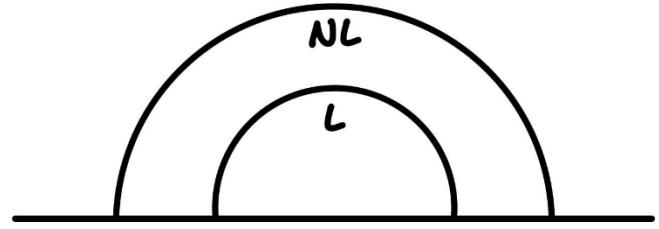
Possiamo ora definire i linguaggi *NL-completi*

DEF: (*NL-completeness*)

Un linguaggio L è *NL-completo* se:

- 1) $L \in NL$ (Membership in *NL*)
- 2) $\forall L' \in NL \quad L' \leq_L L$ (*NL-Hardness*)

Per le riduzioni *Logspace* valgono tutte le proprietà che valevano per le riduzioni polinomiali, *transitività*, ecc. Quindi faremo le riduzioni logspace allo stesso modo di quelle polinomiali



Come per L'*NP-completezza* avevamo *Sat*, anche per *NL-Hard* ci serve un problema di partenza e ci servirà qualcosa che abbia la valenza del problema di Cook però su *NL*.

7.1.2 Reach è NL-complete

Per mostrare che *Reach* è *NL-completo* dobbiamo far vedere che preso un linguaggio L generico questo si riduce in logspace deterministico a *Reach*.

$$\begin{array}{ccc} L & \xrightarrow[f]{\leq_L} & \text{Reach} \\ w & & f(w) = \langle G, s, t \rangle \\ \text{Stringa} & & \text{Tripla: Grafo,} \\ \text{generica} & & \text{sorgente, target} \end{array}$$

Avevamo osservato nel teorema di Cook che L è un linguaggio generico, un linguaggio che non sappiamo cosa sia, quindi non sappiamo se sono formule, non sappiamo se sono grafi, non sappiamo niente, sappiamo che le istanze di L sono stringhe. E queste stringhe vanno trasformate in grafi, tali per cui se $w \in L \Rightarrow$ nel grafo G c'è un path da s a t ; se $w \notin L \Rightarrow$ in G non c'è un path.

Quindi di L non sappiamo niente, come con Cook, ma lì sapevamo che esiste una MdTN che in tempo polinomiale decide L , e lì ci inventammo tutto un gran casino per mostrare che quella MdTN poteva essere codificata in *CNF*. Qua dobbiamo fare una cosa simile, il funzionamento di una macchina M_L che decide L va trasformato in un grafo.

Dobbiamo fare una cosa simile a Cook, li noi avevamo che nella formula booleana dovevamo codificare le configurazioni della macchina; qui facciamo una cosa simile, però sfruttiamo il fatto che l'istanza di arrivo ha un grafo.

I nodi del grafo G rappresenteranno le configurazioni della macchina M sulla stringa w . Quindi i nodi di G ci diranno a che punto della computazione sta la macchina M_L mentre esegue su w . I nodi verranno collegati da archi se uno stato è il legal successor del precedente. Dopodiché dentro questo grafo avremo la foto dello stato iniziale e la foto del target e potremmo andare dallo stato iniziale allo stato finale solo se quello è accettante.

La parte *tricky* di tutto ciò è che questo vada fatto in spazio logaritmico deterministico; quindi, non possiamo generare tutto il grafo e tenerlo su memoria di lavoro, perché occuperebbe troppo spazio.

Quindi f che sputa sul nastro di output la tripla $\langle G, s, t \rangle$ non può mantenere nella propria memoria di lavoro pezzi di grafo intero, occupa troppo spazio. Possiamo tenere cose intermedie e dobbiamo sputare in output qualcosa che sappiamo essere corretto nel momento in cui lo scriviamo.

Ci servono tanti nodi quanti sono le possibili configurazioni dell'albero di computazione di M_L su w . Il nastro della macchina non può essere scritto, di conseguenza la caratterizzazione delle possibili condizioni in cui si trovi la computazione di M su w sono caratterizzate dalla posizione della testina (ma non il suo contenuto), quindi non è necessario generare nodi che prendono in considerazione il fatto che il contenuto del nostro input cambi, dobbiamo considerare che la testina sul nostro input può essere in posizioni diverse ma non che il contenuto cambi.

Quindi per stabilire a che punto la macchina sia ci serve sapere dove stia la testina sull'input, il contenuto del worktape, dove sia la testina sul worktape, lo stato corrente di computazione (q_4, q_{11}, \dots).

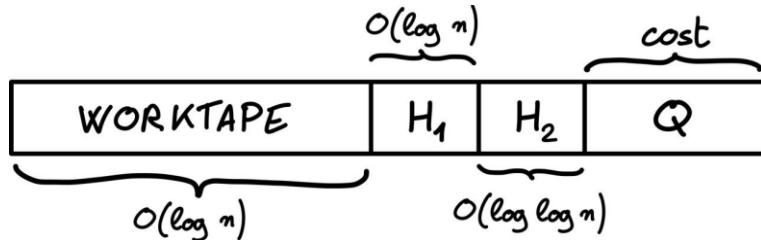
Questa "foto" dello stato corrente della computazione potrebbe essere scritto su una stringa di questo tipo: H_1 e H_2 : numeri che dicono "la testina sul 1° nastro sta alla cella xx", "la testina sul 2° nastro sta alla cella yz" Q per dirci in quale stato siamo.

Noi così siamo in grado di stabilire a che punto del calcolo la macchina M è. Questa stringa è una stringa di simboli alla quale possiamo dare il significato che vogliamo, tipo il nome dei nodi in G , la utilizzeremo come la label dei nodi in G .

WORKTAPE	H_1	H_2	Q
----------	-------	-------	-----

La macchina $M_L \in NL$, quindi usa spazio logaritmico, quindi $|worktape| = O(\log n)$.

H_1 occuperà anche lui $O(\log n)$, H_2 deve indicizzare un nastro di taglia logaritmica quindi $O(\log \log n)$, per Q invece avremo costo costante.



Quindi per scrivere questa stringa rappresentativa ci servirà spazio $O(\log n)$.

Dobbiamo far vedere come f tira fuori questo grafo, il principio è questo:

i nodi sono tanti quanti queste label, gli archi verranno mesi fra nodi che sono uno il legal successor dell'altro secondo la funzione di transizione, i nodi relativi a configurazioni accettanti verranno agganciati a un nodo extra t , il nodo s è il nodo relativo alla configurazione iniziale della macchina. Questa cosa però va fatta in spazio logaritmico, quindi dobbiamo ricilare della roba.

Il trasduttore f per generare il grafo G i cui nodi hanno ognuno dimensione come la label qui sopra,
 Quanti sono? $2^{\log n} = n$.

Allora facciamo in questo modo: il trasduttore si prende la label, istanzia una di queste sul worktape e fa un check "questa stringa è qualcosa di sensato?" (posizione delle testine e nome dello stato)

Inizia a ciclare questo check per ogni simbolo che può comparirci, se ha senso la sputa fori e genera un nodo. In questo modo genero tutti i nodi di G . Notiamo che i nodi di G vengono sputati in output senza essere tutti presenti allo stesso momento sul worktape, questo è il trucco. Cioè io non posso tenerli tutti assieme, quindi ho questa stringa e la ciclo e sputo fuori. Finito questo giro, prendo una stringa e aggiungo un simbolo e sputo quest'ultimo nodo v^* questo sarà t , che è un nodo speciale perché ha un simbolo in più. Cancelliamo tutto e ripuliamo il worktape e ripartiamo con due stringhe e le contiamo entrambe facendo lo stesso giochino per considerare tutte le coppie, che ci faccio con queste coppie:

*"la prima è sensata? Si, la seconda è sensata? Si, la seconda è un
legal successor della prima? Si, sputo in output."*

Quando non verifico questa condizione vado alla successiva, notiamo che il grafo non è mai presente completamente sul worktape. Noi stiamo avanzando contatori.

Dopo il ciclo delle coppie cancelliamo tutto e rifacciamo il giochino dell'esplorazione per andare a vedere quali sono le configurazioni con uno stato accettante, quelle le agganciamo a v^* .

Abbiamo generato G , dentro a G avremo tanti nodi relativi a configurazioni accettanti, ma noi il path dev'essere da s verso un nodo specifico, quindi noi ne mettiamo uno fittizio in maniera tale che il path sia sempre verso il nodo fittizio.

Generiamo il grafo, s è il nodo che rappresenta la configurazione iniziale, $t = v^*$.

Per come è costruito questo grafo ha un path da s a $t \Leftrightarrow M \models w$ e quindi, siccome L è un linguaggio generico che si riduce a *Reach* allora avremo che

$$Reach \in NL\text{-Hard} \Rightarrow Reach \in NL\text{-complete}.$$

7.2 TEOREMA DI SAVITCH

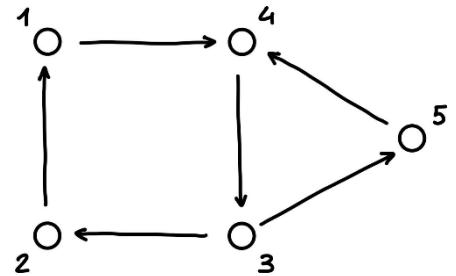
7.2.1 Reach $\in DSpace(\log^2 n)$

Sebbene noi non siamo in grado di dimostrare che

$$\text{Reach} \in L = DSpace(\log n)$$

Siamo però in grado di dimostrare che

$$\text{Reach} \in DSpace \frac{(\log^2 n)}{(\log n)^2}$$



Focalizziamoci su un esempio: Abbiamo questo grafo orientato

Supponiamo che vogliamo andare da $s = 2$ a $t = 5$

Quindi abbiamo un'istanza 'si' di Reach. In particolare: avremo che $s, 1, 4, 3, t$ questo il path

Il trucco per riuscire a risolverlo in spazio $\log^2 n$ è praticamente fare una specie di ricerca binaria.

"se esiste un path per andare da s a t $\Rightarrow \exists$ un nodo intermedio u i.e.: $u = 4$ e ci deve stare un path da $s \rightarrow u$ e da $u \rightarrow t$ ".

Abbiamo quindi un approccio di tipo **ricorsivo** quindi ci inventeremo un algoritmo ricorsivo di questo tipo

Se voglio andare $s \rightarrow t$ ho tre casi: due casi base e uno in cui applichiamo la ricorsione

1. $s = t \Rightarrow$ rispondiamo 'si'
2. t successivo $s \Rightarrow$ rispondiamo 'si'
3. Else \Rightarrow proviamo tutte le u per vedere se $\exists s \rightarrow u \wedge u \rightarrow t$

```

1. Exists-path(G=<V,A>, s, t, k){
2.   if k == 0 {
3.     if s == t { ACCEPT; }
4.     else { REJECT; }
5.   } if k == 1 {
6.     if (s,t) ∈ A { ACCEPT; }
7.     else { REJECT; }
8.   } for each u ∈ V {
9.     if Exists-path(G, s, u, [k/2]) and
10.      if Exists-path(G, s, u, [k/2]) {
11.        ACCEPT;
12.      }
13.    }
14.  REJECT;
15. }
```

?

Qual è la complessità spaziale di questo algoritmo? Di quanta memoria necessita?

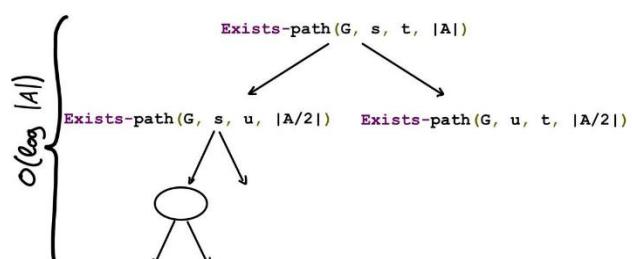
Ci servirà dello spazio in memoria per sapere chi sono s e t , quindi o dei puntatori verso il nastro di input dove appaiono s e t nella descrizione del grafo, oppure un Id di taglia logaritmica. A noi servono due registri di taglia logaritmica su worktape, uno per s e uno per t che ci servono nei test a riga 3. e a riga 6. Dopodiché ci serve un altro spazio di taglia logaritmica per tenere traccia di chi è l' u corrente.

Quindi fino a riga 7. utilizziamo tre pezzettini di taglia logaritmica.

Per la parte ricorsiva andiamo a vederne l'albero di computazione ricorsiva.

La chiamata iniziale è sul numero degli archi e genereremo, per ogni u , due chiamate ricorsive.

Le due chiamate per u non vengono eseguite in parallelo, prima una e poi l'altra \Rightarrow esploreremo prima la prima chiamata che a sua volta ne genererebbe altre due chiamate ricorsive e l'ovale ne genera altre due, e così via...



Ma noi di queste chiamate ricorsive quando si biforca ne abbiamo sempre una in memoria, non ne facciamo mai due contemporaneamente.

?

Quanto è profondo però l'albero? Perché noi avremo spazio logaritmico per il numero di chiamate su stack.

Le chiamate sono logaritmiche perché ogni volta dimezziamo k . Quindi la profondità è $O(\log|A|)$

$$\Rightarrow O(\log|A|) \cdot O(\log n) = O(\log^2 n)$$

7.2.2 Proof Savitch

THM: Savitch

Sia $s(n)$ una space function tale che $s(n) \in \Omega(\log n)$. Allora $N\text{Space}(s(n)) \subseteq D\text{Space}(s(n)^2)$

Proof:

Per dimostrare questo teorema noi dobbiamo sfruttare due elementi:

la riduzione verso *Reach* e l'algoritmo a pagina precedente.

Sia $L \in N\text{Space}(s(n)) \Rightarrow \exists$ una MdTN M tale che $\mathcal{L}(M) = L$ e il suo running space è $s(n)$.

Di questa macchina M noi possiamo andare ad analizzare il suo computation graph, ovvero un grafo i cui nodi sono relativi ai vari stati di computazione della macchina ed esiste un arco che collega due nodi se uno è il legal successor del precedente rispetto alla funzione di transizione di M .

Sebbene L sia riconoscibile in spazio non-det. $s(n)$, per riconoscerlo in spazio deterministico $s(n)^2$ noi:

- › Costruiamo il suo computation graph
- › lanciamo l'algoritmo di *Reach* sul grafo
- › Verifichiamo se esiste un percorso dalla configurazione iniziale a quel nodo v^*

Per fare questo però ci dobbiamo chiedere quanto grosso sia questo grafo, perché questo grafo lo dobbiamo rappresentare prima di lanciarci sopra l'algoritmo di *Reach*.

Noi abbiamo che i nodi possono essere distinti, guardando a quattro elementi:

1. Il contenuto del worktape
2. La posizione della testina sul nastro di input
3. La posizione della testina sul nastro di lavoro
4. Lo stato corrente di computazione Q

E avremo le seguenti dimensioni:

- 1) $O(s(n))$ perché lo spazio che noi diamo disponibile a M per eseguire è $s(n)$
- 2) $O(\log n)$ perché i puntatori alle celle del nastro di input hanno taglia $\log n$
- 3) $O(\log s(n))$ perché
- 4) $cost$ perché il numero degli stati non dipende dalla lunghezza dell'input

Siccome $s(n) \in \Omega(\log n) \Rightarrow$ tutta la label richiede $O(s(n))$ simboli

A questo punto possiamo fare delle considerazioni sul grafo:

Quanti sono i nodi?

Su worktape ci possono stare i vari simboli di nastro, il tutto è bounded dalla quantità di nodi che si possono generare se usassimo tutti i simboli di nastro in tutte le posizioni.

Sia $l = |\Gamma|$ allora il numero di nodi è $O(l^{s(n)})$

Quando rappresentiamo il grafo facciamo solo la lista dei nodi? No, ci servono gli archi che sono

$$O((l^{s(n)})^2) = O(l^{2s(n)})$$

Fra i due abbiamo più archi, quindi la taglia del computation graph è $O(l^{2s(n)})$

Abbiamo

$$O((\log l^{2s(n)})^2) \Rightarrow O\left(\left(\frac{2}{\text{cost}} \cdot s(n) \cdot \log l \right)^2\right) \Rightarrow O(s(n)^2)$$

Possiamo quindi definire $PSpace$ e $NPSpace$, che sono l'equivalente di P e NP ma invece di essere sul tempo possono essere sullo spazio

$$PSpace = \bigcup_{c \geq 1} DSpace(n^c), \quad NPSpace = \bigcup_{c \geq 1} NSpace(n^c)$$

$$PSpace \subseteq NPSpace$$

Per Savitch abbiamo che

$$PSpace = NPSpace$$

Savitch ci dice che tutte le classi di complessità spaziali da *Polinomial* a salire, la classe deterministica eguaglia la classe non-deterministica.

L'intuizione è che, siccome abbiamo vincoli sullo spazio ma non sul tempo, il tempo una volta che lo usiamo è andato, lo spazio invece possiamo riutilizzarlo. Ecco perché $PSpace = NPSpace$

Una macchina a $PSpace$ può simulare una macchina $NPSpace$ semplicemente riprovando, lo spazio di uso è sempre lo stesso.

Abbiamo detto che $Logspace \subseteq NL$. Ma che relazione esiste fra $Logspace$ e $PTime$?

Supponiamo di avere una macchina in spazio logaritmico, deterministica, che si arresti sempre (risponde sempre sì/no). Quanto tempo ha a disposizione? Polinomiale! Infatti, tutte le possibili configurazioni attraversate dalla macchina sono $2^{\text{dimensione della configurazione generica}}$, dunque nel caso di macchina in $Logspace$ è $2^{\log n}$, che complessivamente è una quantità polinomiale. Si arresta sempre \Rightarrow non passa mai per 2 volte nella stessa configurazione (altrimenti ci sarebbe un loop, ma sappiamo che si arresta sempre). Perciò, avendo un numero polinomiale di configurazioni, anche il tempo sarà polinomiale! Da cui $L \subseteq PTime$.

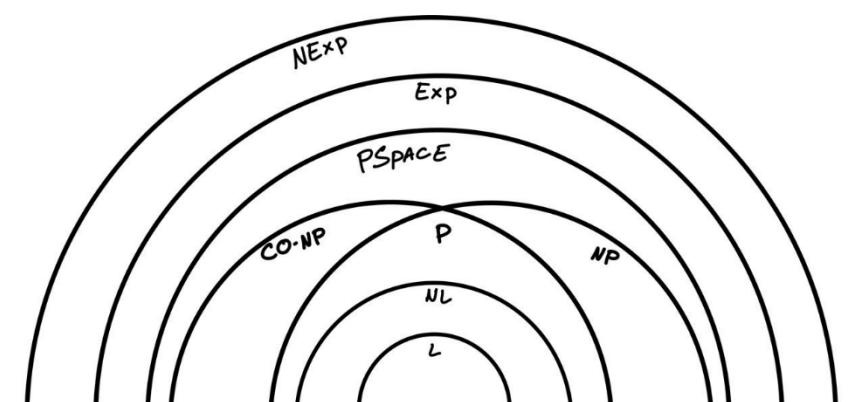
Che relazione c'è invece fra NL e P ? *Reach* è *NL-completo*; questo implica che tutti i problemi di NL possano essere trasformati in tempo polinomiale in *Reach*, che è a sua volta risolvibile in tempo polinomiale tramite breadth-first search. Da cui $NL \subseteq PTime$.

Che relazione fra NP e $NPSpace$? Supponiamo MdT non deterministica in tempo polinomiale. In tempo polinomiale non può scrivere più di una quantità polinomiale, perciò $NP \subseteq NPSpace = PSpace$ (Savitch).

Sia L un linguaggio $\in CO-NP$. Per definizione di *CO-NP*, $\bar{L} \in NP \Rightarrow \bar{L} \in PSpace$. $PSpace$ è una classe deterministica, dunque è chiusa sotto complemento! Da cui $L \in PSpace$, perciò $CO-NP \subseteq PSpace$.

Consideriamo macchina in $PSpace$; quanto tempo ha a disposizione? Esponenziale! (stessa logica di 2^{roba} vista sopra). $PSpace \subseteq EXPTime \subseteq NEXPTime$.

Complessivamente:



Problemi *P-complete* = fra i più difficili in P . Riteniamo che non siano in *Logspace*, ma non dimostrato.

Sono i problemi inerentemente sequenziali, nel senso di non parallelizzabili.

I problemi in *Logspace* sono invece parallelizzabili.

8 MDT CON ORACOLI

Introduciamo gli ultimi due concetti; non andremo troppo nei dettagli ma è fornire un'idea. Vedremo Oracoli, macchine a oracolo e un'introduzione alle classi funzionali.

Ci focalizziamo sul problema del *Vertex Cover*, ovvero “*dato un grafo, un VC è un insieme dei suoi nodi, tale per cui, per ogni arco, almeno uno degli end-point è coperto da uno dei vertici nel VC.*”

Avevamo definito il problema di decisione per *VC*:

$$VC = \{\langle G, k \rangle | G \text{ è un grafo che ammette un vertex cover di taglia } \leq k\}$$

$$VC \in NP\text{-complete}$$

VC ∈ *NP* perché per rispondere sì guessiamo un vertex cover, verifichiamo che la sua taglia sia ≤ *k*, verifichiamo che il set di nodi guessati è un vertex cover e rispondiamo di sì. Abbiamo quindi un certificato conciso per rispondere di ‘sì’.

Definiamo ora questo problema:

$$Min\text{-}cover = \{\langle G, k \rangle | G \text{ è un grafo i cui vertex cover di taglia minima hanno taglia } k\}$$

Dato questo grafo e un intero k, è vero o no che i suoi vertex cover di taglia minima contengono esattamente k nodi?

❓ In quale classe di complessità potrebbe stare *Min-cover*?

- NP?

Abbiamo una procedura che ci dice un grafo e ci dà 15, se noi guessiamo un *VC* di taglia 15 possiamo dire di ‘sì’, che quello è il *VC* è di taglia minima?

No.

Il problema, quindi, è questo noi possiamo guessare una cosa per **tentare** di rispondere di ‘sì’. Cioè, noi guessiamo un *VC* di taglia *k*, quel set di nodi di taglia *k* è un *VC* però noi stiamo semplicemente dicendo che **esiste** un *VC* di quella taglia, non abbiamo la garanzia che sia minimo.

❓ Possiamo guessare qualcosa che ci dia la garanzia che sia minimo?

Non si può fare.

- CO-NP? Quindi devo poter guessare qualcosa per rispondere di ‘no’

Intuitivamente non si può nemmeno in *CO-NP*. Perché nel momento in cui guessiamo qualcosa per rispondere di no, noi non lo sappiamo se quella cosa che abbiamo guessato è un *VC* di taglia minima o meno.

Quindi pare che questo problema non stia né in *NP* né in *CO-NP*. Sta in *ExpTime* perché possiamo testare tutto, possiamo dare un bound migliore di *ExpTime* tra le classi che abbiamo visto?

Quello che si può fare, per risolvere questo problema in *PSpace* è testare tutti i *VC* candidati, quindi, dobbiamo testare tutti i possibili sottoinsiemi di *V* (dell’insieme dei nodi) e ne testiamo uno per volta.

Ogni volta che facciamo il test lo scriviamo su nastro, facciamo il test se quello è un *VC* o meno. Se lo troviamo ci appuntiamo la sua taglia ... insomma andiamo alla ricerca della taglia più piccola.

Il trucco sta che ogni volta che noi facciamo un tentativo poi cancelliamo tutto, manteniamo giusto il valore temporaneo nel minimo trovato e rigeneriamo un altro *VC*. Quindi noi con spazio polinomiale, se lo riutilizziamo siamo in grado di rispondere a questo problema.

$$\Rightarrow Min\text{-}cover \in PSpace$$

Quello che vogliamo fare ora è dare un bound migliore a questo genere di problemi. Perché questo problema, sebbene sia risolvibile in *PSpace*, non è uno dei problemi più difficili di *PSpace*. In realtà questo problema sta in delle classi al di sotto di *PSpace* ma al di sopra di *NP* che finora non abbiamo visto.

Andremo ora a introdurre queste classi intermedie che stanno tra *NP* e *PSpace* e caratterizzano problemi di questo tipo, nelle quali un guess soltanto non ci basta e *PSpace* sembra “sparare con un cannone a una mosca”, cioè in *PSpace* potremmo fare una marea di cose ma in realtà questo problema è molto più semplice.

Notiamo che a noi bastano due test, ci basta testare che esiste un *VC* di taglia k e che non esiste un *VC* di taglia $k - 1$, quindi possiamo cambiare la definizione del problema in questo modo:

$$\text{Min-cover} = \{\langle G, k \rangle | \langle G, k \rangle \in \text{VC} \wedge \langle G, k - 1 \rangle \notin \text{VC}\}$$

Abbiamo quindi così definito lo stesso problema.

Se noi abbiamo un decisore del problema di *VC* noi possiamo uscirne fuori *facendo un paio di domande* a questo decisore di *VC*; quindi, se noi avessimo una sub-routine capace di risolvere *VC*, noi avremmo semplicemente una cosa di questo tipo:

```
1. min-cover(G, k) {
2.   result1 = check-VC(G, k);
3.   result2 = check-VC(G, k-1);
4.   return: result1 AND NOT result2;
5. }
```

❓ Noi stiamo chiedendo “è vero o no che esiste un *VC* di taglia l più k ?”

se ci dice di **sì** sappiamo che ci sono *VC* di G di taglia al più k .

se ci risponde di **no**, allora vuol dire che il *VC* più piccoli che G ammette hanno taglia esattamente k .

Quindi in qualche modo noi vorremmo legare la complessità di questo problema al fatto che noi possiamo risolverlo in tempo polinomiale deterministico, nel momento in cui disponessimo di una procedura che è in grado di risolvere il problema del Vertex cover.

Per questo introduciamo un modello di Macchina di Turing differente che chiamiamo **macchine ad oracolo**, il modello delle macchine ad oracolo serve a catturare questa cosa del chiamare le sub-routine.

L'intuizione è che sia una MdT in grado di fare query ad un oracolo che gli risponde qualcosa.

le denotiamo: $M^?$

Quindi nell'esempio di sopra noi avremo una MdT che fa due domande a un oracolo per *VC*: “scusa mi sai dire se questa è un'istanza si?” poi dice “te ne faccio un'altra, mi puoi dire se questa è un'istanza si?” e gli dà una risposta, in base alle due risposte la macchina chiamante è in grado di rispondere si/no.

Le macchine ad oracolo (deterministiche o non-det, perché possono essere o l'una o l'altra) hanno:

- Oracle-tape (write-only):
Nastro di sola scrittura in cui la macchina scrive la propria domanda per l'oracolo.
- Tre stati particolari:
 - > $q_?$ Il chiamante scrive sull'oracle-tape la domanda, dopodiché transisce nello stato $q_?$. In quel momento l'oracolo decide l'appartenenza della stringa che appare in quel momento sull'oracle tape e in un solo passo della macchina chiamante avviene la transizione verso q_{yes} o q_{no} in dipendenza se l'oracolo sta rispondendo di sì o di no.
 - > q_{yes}
 - > q_{no}
- Nel momento in cui riceviamo la risposta dall'oracolo magicamente il contenuto dell'oracle-tape si cancella; quindi, non dovremo perdere tempo a cancellarlo.

Quindi come potremmo implementare la procedura di sopra con una macchina ad oracolo?

1. Scrive sul nastro il proprio input $\langle G, k \rangle$
2. L'oracolo dirà sì/no
3. Se dice no, fatta, non vado oltre, se dice sì
4. Prendo dall'input k e sottraggo 1
5. Scriviamo sul nastro dell'oracolo $\langle G, k - 1 \rangle$
6. L'oracolo dirà sì/no
7. In base alle risposte che riceviamo, le combiniamo e rispondiamo sì k è la taglia minima o no, non lo è.

La macchina la denotiamo $M^?$ perché la funzione di transizione di M è indipendente dalla funzione di transizione dell'oracolo.

Quando vogliamo dire che la macchina sta facendo domande ad un oracolo per un linguaggio L scriviamo:

$$M^L$$

Che vuol dire che la macchina ha come oracolo un decisore per il linguaggio L

8.1.1 Classi ad oracolo

Possiamo quindi definire le seguenti classi ad oracolo:

DEF: (classe

Sia C una classe di complessità (i.e.: P , NP , etc...);

$$P^C = \left\{ L \left| \begin{array}{l} L \text{ può essere deciso in un tempo polinomiale da} \\ \text{una macchina ad oracolo deterministica che} \\ \text{interpella un oracolo per un linguaggio } L' \in C \end{array} \right. \right\}$$

E similmente possiamo definire:

$$NP^C = \left\{ L \left| \begin{array}{l} L \text{ può essere deciso in un tempo polinomiale da} \\ \text{una macchina ad oracolo non deterministica che} \\ \text{interpella un oracolo per un linguaggio } L' \in C \end{array} \right. \right\}$$

?

Min-cover in che classe sta?

In P^{NP} \exists *Min-cover* perché è un problema che può essere deciso da un decisore deterministico in tempo polinomiale che interpreta un oracolo di un linguaggio che sta in NP .

Relazione tra NP e P^{NP}

?

Come sono relazionate queste due classi di complessità?

$$NP \subseteq P^{NP}$$

Sia $L \in NP$, per dimostrare che $L \in P^{NP}$ noi dobbiamo trovare una macchina ad oracolo che in tempo polinomiale, utilizzando un oracolo per NP , è in grado di rispondere a L .

L'oracolo della macchina M è per L ; quindi, noi prendiamo l'input lo copiamo sull'oracle-tape, gli facciamo la domanda, prendiamo la risposta e rispondiamo la stessa cosa.

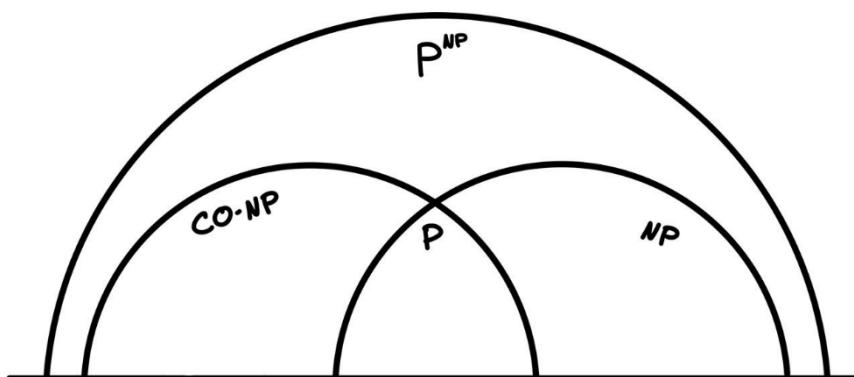
Relazione tra $CO-NP$ e P^{NP}

?

Come sono relazionate queste due classi di complessità?

$$CO-NP \subseteq P^{NP}$$

Sia $L \in CO-NP \Rightarrow \bar{L} \in NP$, noi diamo alla macchina M l'oracolo \bar{L} , M fa la domanda all'oracolo lui risponde, M inverte la risposta e restituisce il risultato.



8.1.2 Gerarchia polinomiale

Abbiamo quindi introdotto le classi P^{NP} , NP^{NP}

Questa cosa può essere generalizzata e potrebbe avere cose ancora più sofisticate ad esempio

$$NP^{NP^{NP}}$$

Questa sarebbe una macchina NP che ha come oracolo linguaggi in NP^{NP} .

Quindi noi possiamo fare delle catene molto lunghe di tanti oracoli, però il significato è che il chiamante ha come linguaggio dell'oracolo un linguaggio che sta in NP^{NP} . Questa cosa si può generalizzare, avere tre, quattro, cinque, mille livelli ... e da origine a quella che viene chiamata **gerarchia polinomiale**.

DEF: (gerarchia polinomiale)

La gerarchia polinomiale è un insieme di classi di complessità al di sopra di NP che hanno questi nomi:

$$\Sigma_i^P \quad \Pi_i^P \quad \Delta_i^P \quad i \geq 0$$

Dove

$$\Sigma_0^P = P$$

Per $i \geq 1$

$$\Sigma_i^P = NP^{\Sigma_{i-1}^P}$$

Per $i \geq 0$

$$\begin{aligned} \Pi_i^P &= CO\text{-}\Sigma_i^P \\ \Delta_{i+1}^P &= P^{\Sigma_i^P} \end{aligned}$$

Quindi

$$\Delta_2^P = P^{NP} \Rightarrow \text{Min-cover} \in \Delta_2^P$$

?

Consideriamo un generico Σ_i^P e un generico $\Delta_{i+1}^P = P^{\Sigma_i^P}$, che relazione c'è tra loro?

È contenuto dentro Δ , (similmente a $NP \subseteq P^{NP}$) noi basta che usiamo una macchina a oracolo che chiede al proprio oracolo di decidere il linguaggio in Σ_i^P , prende la risposta e la dà in output.

$$\Sigma_i^P \subseteq \Delta_{i+1}^P$$

?

Cosa possiamo dire di $\Pi_i^P = CO\text{-}\Sigma_i^P$ e di Δ_{i+1}^P ?

Per un ragionamento simile a quello di prima possiamo dire che (similmente a $CO\text{-}NP \subseteq P^{NP}$)

$$\Pi_i^P \subseteq \Delta_{i+1}^P$$

?

Cosa possiamo dire di Δ_i^P e di Σ_i^P ?

Perché entrambe sono classi il cui oracolo è Σ_{i-1}^P però Δ_i^P il chiamante è deterministico, in Σ_i^P il chiamante è non-deterministico e perciò è più potente.

$$\begin{array}{ccc} \Delta_i^P & \subseteq & \Sigma_i^P \\ || & & || \\ P^{\Sigma_{i-1}^P} & & NP^{\Sigma_{i-1}^P} \end{array}$$

?

Cosa possiamo dire di Δ_i^P e di Π_i^P ?

Consideriamo un linguaggio $L \in \Delta_i^P = P^{\Sigma_{i-1}^P}$, sappiamo che esiste una macchina $M^?$ deterministica che lavora in tempo polinomiale che, facendo domande ad un oracolo per un linguaggio $L \in \Sigma_{i-1}^P$, è in grado di decidere L .

Ma questa macchina M , il chiamante dell'oracolo, è deterministica; quindi, possiamo trasformare $M^?$ e scambiare gli stati di accettazione e rifiuto; chiamiamola $\overline{M}^?$ lei andrà a decidere $\overline{L} \Rightarrow \overline{L} \in P^{\Sigma_{i-1}^P} = \Delta_i^P$.

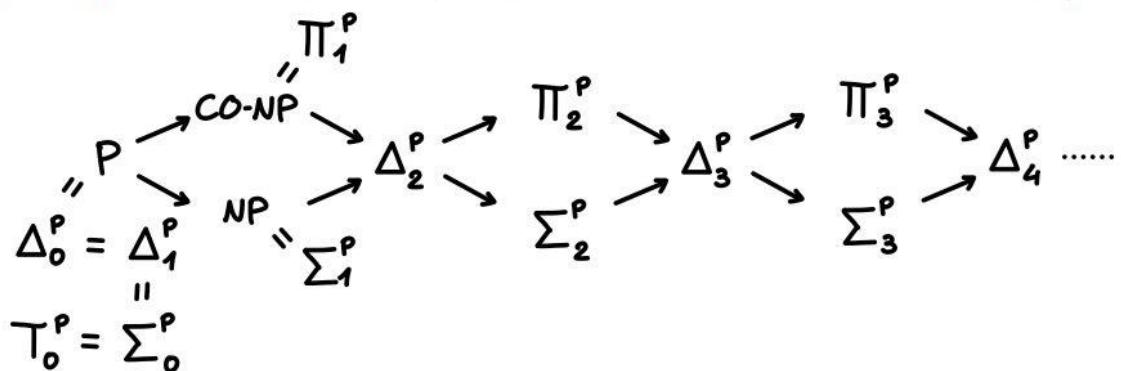
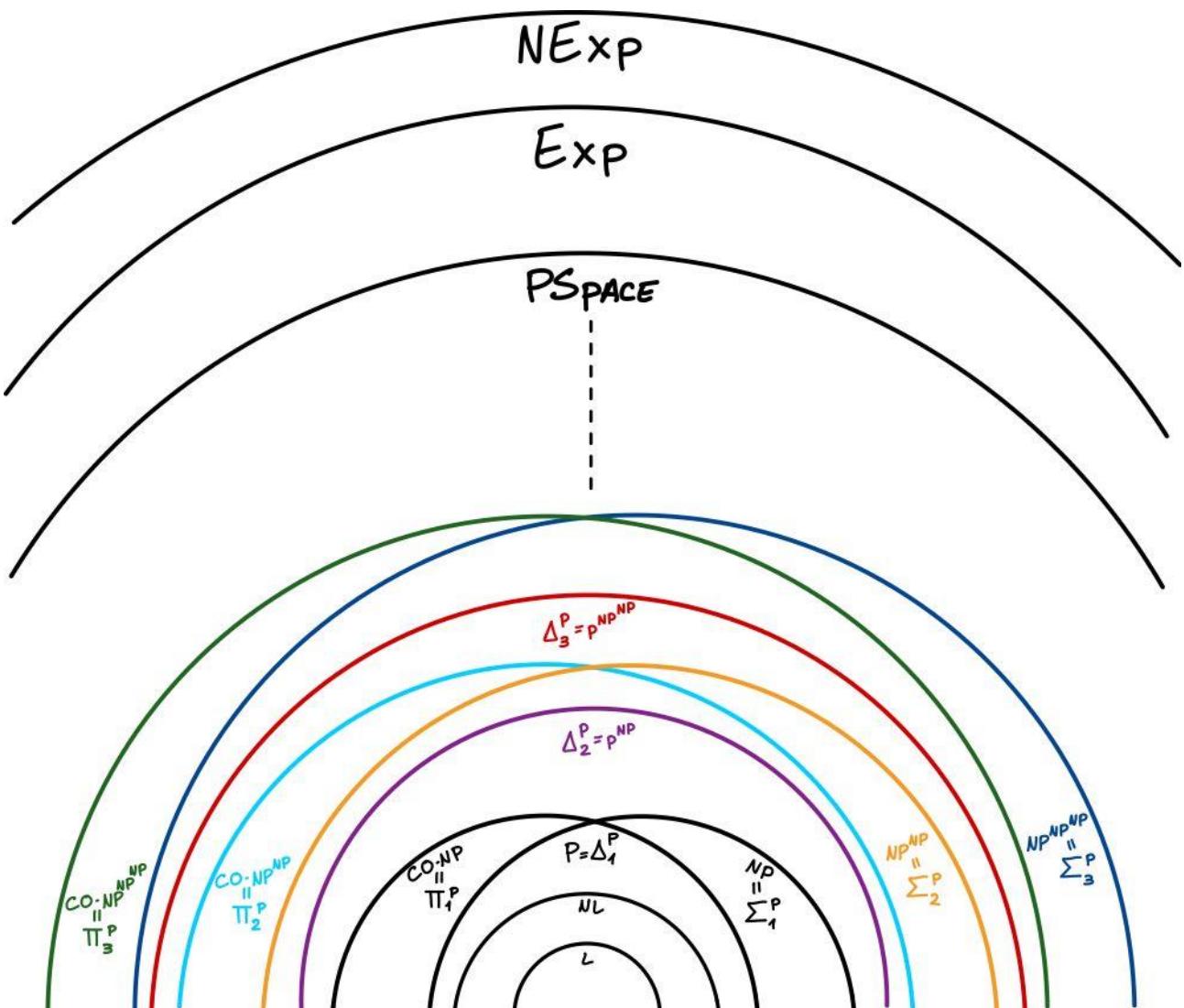
Per il risultato di prima $\Delta_i^P \subseteq \Sigma_i^P \Rightarrow \overline{L} \in \Sigma_i^P$, da cui $L \in \Pi_i^P \Rightarrow \boxed{\Delta_i^P \subseteq \Pi_i^P}$.

P sta dentro a $\frac{CO-NP}{NP}$ che stanno dentro a Δ_2^P .

Δ_2^P sta dentro a $\frac{\Pi_2^P}{\Sigma_2^P}$ che stanno dentro a Δ_3^P e così via ...

Q Quanti livelli ci sono di questa gerarchia?

Pensiamo siano infiniti; Infatti, uno dei problemi aperti è stabilire se il numero dei livelli della gerarchia polinomiale sia infinito o finito; quello che riteniamo è che sia infinito.



Tutte queste sembrano classi un po' strane, e di fatti lo sono. Cerchiamo di dare un'intuizione su quali problemi stanno lì dentro.

Abbiamo visto *Sat*, come possiamo scriverlo?

$$Sat \rightarrow \exists \bar{x}, \quad \phi(\bar{x}) \quad \exists\text{-}Sat$$

“È vero o no che esiste un modo di assegnare un valore di verità alle variabili booleane in x tale che la formula $\phi(x)$ è vera?”

Prima non avevamo il simbolo di esiste perché lo dicevamo a parole. Quindi la formula verificata potremmo avere:

$$(\exists x_1, x_2, x_3) : ((x_1 \vee x_2) \wedge (x_3 \Rightarrow \neg x_1) \vee \dots)$$

Questo è il *Sat* standard e per questa ragione lo chiamiamo $\exists\text{-}Sat$

Noi possiamo generalizzarla a questa cosa; consideriamo questa formula:

$$(\exists x_1, x_2) : (\forall y_1, y_2, y_3) \text{ abbiamo che } (x_1 \vee \neg y_1 \vee x_2) \wedge (\neg y_1 \vee \neg x_2 \vee y_3) \dots$$

Questa formula booleana complessa e il significato è: ci chiediamo se esista un assegnamento di verità per le variabili x_1 e x_2 tale che, qualsiasi sia l'assegnamento di verità che diamo alle variabili y_1, y_2, y_3 la formula è soddisfatta.

“È vero che esiste un modo di assegnare valori booleani alle variabili x tale per cui qualsiasi cosa facciamo sulle variabili y la formula è soddisfatta?”

Questo è un problema di soddisfacibilità diverso e lo chiamiamo $\exists\forall\text{-}Sat$

Si può mostrare, noi non lo vedremo che $\exists\forall\text{-}Sat \in \Sigma_2^P$ -completo, cioè è un problema tosto della classe Σ_2^P

Similmente si può definire $\exists\forall\exists\text{-}Sat \stackrel{\epsilon}{\Rightarrow} \Sigma_3^P$ -completo.

Notiamo che questa cosa è meno alieno di quanto possa sembrare:

Consideriamo gli scacchi, *“è vero che esiste una mossa che io posso fare, tale per cui qualsiasi sia la risposta del mio avversario, esiste una risposta mia tale che qualsiasi cosa faccia il mio avversario esiste una risposta mia tale che bla bla bla... e alla fine vinco?”* Questi sono i problemi di questa gerarchia, cioè un'alternanza di giocatori in cui *“esiste una strategia vincente per me tale per cui qualsiasi cosa faccia l'altro ho sempre il modo di vincere?”* questa cosa ovviamente va avanti all'infinito perché dipende da quanto si va avanti...

Quindi, giusto per avere un'intuizione, questo problema stranissimo delle formule booleane quantificate in realtà, per esempio, è la complessità dei giochi. I giochi hanno quel tipo di complessità perché io mi chiedo *“se io ho un avversario intelligente, esiste una mossa che io posso fare ora che qualsiasi cosa faccia l'altro per me esiste un modo bla bla bla... e alla fine vinco?”* Questa è la complessità di questo genere di problemi

8.2 PROBLEMI DI OTTIMIZZAZIONE E CLASSI FUNZIONALI

Definiamo questo problema: (attenzione, è il primo problema di calcolo che vediamo)

$$FMin-cover(G) = \min\{|v| \mid v \text{ è un vertex cover per } G\}$$

Functional Min-cover: dato un grafo G noi vogliamo calcolare la taglia del più piccolo vertex cover.

DEF: (Functional Deterministic Polynomial)

FP è la classe delle funzioni calcolate da trasduttori deterministici in tempo polinomiale.

FP è la classe cugina di 109 . P è l'insieme dei problemi di decisione, decidibili da MdT deterministiche in tempo polinomiale, il suo cuggino FP è l'insieme delle funzioni che possono essere calcolate da trasduttori deterministici in tempo polinomiale.

Notiamo che le riduzioni polinomiali sono funzioni che stanno in FP .

FNP , che sarebbe quello legato a NP non lo vediamo perché sarebbe “un macello”, quindi ci focalizziamo su FP che è facile e lineare.

Quindi, in soldoni, FP sono quelle funzioni per le quali abbiamo algoritmi deterministici che le calcolano.

Supponiamo di voler risolvere $FMin-cover(G)$ e abbiamo a disposizione un oracolo per VC (decisionale). Quindi abbiamo una bella procedura in una libreria che, data una coppia Grafo-numero, è in grado di dirci se quel grafo ha un vertex cover di taglia al più quel numero.

Noi vogliamo risolvere il problema di calcolare, dato un grafo, la taglia del vertex cover minimo di quel grafo.

?

Se sappiamo che abbiamo una procedura che risolve il problema di decisione, che algoritmo ci possiamo inventare per riuscire a calcolare il vertex cover di taglia minima?

Iniziamo da $k = 1$, e iniziamo a chiedere all'oracolo “è vero che questo grafo ha un VC di taglia 1?”

E lui dice “no”. Ok, andiamo avanti; $k = 2$:

“è vero che questo grafo ha un VC di taglia 2?” no. $k = 3$ e poi 4 e 5 ... al primo che ci dice ‘si’ ci fermiamo e restituiamo il k a cui eravamo.

Per questa ragione, noi possiamo dire che $FMin-cover \in FP^{NP}$

Scriviamo questo perché un trasduttore deterministico in tempo polinomiale, è in grado di calcolare il risultato se nella sua esecuzione chiede l'aiuto di un oracolo in NP .

?

Perché prende tempo polinomiale?

Perché noi ciclamo su tutti i $k = 1, 2, 3, \dots$ male che va noi dobbiamo ciclare fino al limite del numero dei nodi, perché al numero dei nodi riceveremo sicuramente risposta ‘si’ dall'oracolo.

Quindi noi ci prendiamo tempo polinomiale perché facciamo, male che ci va, un numero di domande pari al numero di nodi del grafo, e poiché sono esplicitamente rappresentati nell'input, quello è un numero polinomiale di chiamate.

?

C'è una procedura più efficiente. Quale può essere?

Una ricerca binaria: stesso oracolo, prendiamo la taglia del numero dei nodi n , e iniziamo con $n/2$ se ci dice ‘si’ allora il minimo sta tra 0 e $n/2$, se ci dice ‘no’ allora il minimo sta tra $n/2$ ed n . E ripetiamo...

?

Quante domande facciamo all'oracolo?

Un numero logaritmico, quindi questa procedura è più efficiente, e lo scriviamo:

$$FMin-cover \in FP^{NP} [O(\log n)]$$

A questo punto siamo in grado di stabilire una relazione detta all'inizio del corso:

avevamo detto “ci focalizziamo sui problemi di decisione perché più facili, ma i problemi di decisione sono legati ai problemi di ottimizzazione e di ricerca”

? Secondo voi, *FMin-cover*, quindi calcolare la taglia del vertex cover minimo di un grafo G può stare in FP ?

Se noi fossimo in grado di calcolare in tempo polinomiale deterministico la taglia del vertex cover minimo di un grafo, il problema del vertex cover decisionale starebbe in P , perché calcoleremmo la taglia minima, lo confrontiamo col k passato in input, vediamo chi è più grande e rispondiamo sì/no. Ma noi sappiamo che $VC \in NP\text{-complete}$ e ciò comporterebbe che $P = NP \Rightarrow$ calcolare la taglia minima del vertex cover di un grafo riteniamo non sia possibile farlo in tempo polinomiale perché collasserebbe NP su P .

Quindi sapere che la versione decisionale di VC è un problema $NP\text{-completo}$, ci dà un bound stretto forte su qual è la complessità del calcolo.

Se noi fossimo in grado di calcolare in tempo polinomiale anche il problema di decisione sarebbe polinomiale deterministico e a quel punto avremmo $P = NP$.

Ed ecco dov'è la connessione tra problemi di ricerca/ottimizzazione e le loro varianti di decisione, se un problema di decisione è tosto \Rightarrow la sua variante di calcolo non può essere semplice.

8.2.1 Travelling Salesman Problem (TSP)

Vediamo ora un altro esempio di problema di ricerca, la cui complessità può essere compresa analizzando la complessità della sua versione decisionale. Per definire questo problema introduciamo alcune nozioni:

DEF: (Grafo pesato)

Un grafo pesato G , è un grafo orientato o meno definito da

$$G = \langle V, E, \lambda \rangle \quad \lambda: E \rightarrow \mathbb{N}$$

Dove λ è una funzione che mappa archi su interi, cioè, associa un peso agli archi del grafo.

DEF: (Hamiltonian cycle)

Un Hamiltonian cycle di un grafo G (possibilmente pesato) diretto o non diretto, è un ciclo di G che visita ogni nodo esattamente una volta.

DEF: (FTSP)

Il (Functional) Travelling Salesman Problem è un problema di calcolo,

Dato un grafo non orientato e pesato $G = \langle V, E, \lambda \rangle$ restituire il costo minimo di un Hamiltonian cycle di G (se ne ha) altrimenti rispondere con un simbolo di default.

$$FTSP(G) = \min\{\lambda(\pi) \mid \pi \text{ è un Hamiltonian cycle di } G = \langle V, E, \lambda \rangle\}$$

Ci occuperemo prima della questione della funzione di calcolo e poi andremo a guardare la *Hardness*.

Risolveremo questo problema tramite una macchina a oracolo; supponiamo di avere un oracolo per *TSP*, ovvero la versione di decisione di questo problema:

$$TSP = \{(G, k) \mid G = \langle V, E, \lambda \rangle \text{ è un grafo non orientato e } G \text{ ammette un Hamiltonian cycle di peso } \leq k\}$$

Quindi un'istanza ‘sì’ di *TSP* sono le coppie (G, k) dove G è un grafo pesato non orientato e $k \in \mathbb{N}$

“è vero o no che esiste un Hamiltonian cycle in G di peso al più k ? ”

?

Come possiamo risolvere *FTSP* tramite chiamate a un oracolo per *TSP*?

- Testiamo se $\langle G, sum \rangle \in TSP$. Se l'oracolo risponde "no" $\Rightarrow G$ non ha un Hamiltonian cycle
- Altrimenti procediamo con una ricerca binaria sul dominio $[0, sum]$

Noi faremo un numero di domande logaritmiche nel dominio di ricerca, la dimensione è esponenziale nella taglia dell'input, di conseguenza, il numero di domande che facciamo all'oracolo è polinomiale nella taglia dell'input.

$$\Rightarrow FTSP \in FP^{NP}$$

Cioè, un trasduttore polinomiale deterministico, se mai avesse accesso a un oracolo per il problema del *TSP*, lui saprebbe risolvere il problema *TSP* in tempo polinomiale.

?

Ci chiediamo: "ci serve o non ci serve l'accesso all'oracolo?" cioè, siamo in grado di risolvere questo problema in tempo polinomiale senza avere aiuti da casa?

Dipende dalla complessità di *TSP*, dalla versione decisionale del problema.

Perché se *TSP* dovesse essere *NP* completo, cioè se è un problema tosto, allora noi potremmo risolvere *FTSP* (cioè, la versione funzionale del problema) in tempo polinomiale, solamente se $P = NP$.

Perché, se noi fossimo in grado di risolvere *FTSP* in tempo polinomiale \Rightarrow potremmo risolvere *TSP* in tempo polinomiale. Come?

Ci calcoliamo il peso dell'Hamiltonian cycle più leggero (stiamo assumendo sia fattibile in tempo polinomiale) una volta che abbiamo questo numero lo confrontiamo con il k che ci passa in input e stabiliamo se quella è un'istanza sì o no di *TSP*.

Quindi se *FTSP* fosse risolvibile in tempo polinomiale deterministico $\Rightarrow TSP \in P$.

Se invece dimostriamo che $TSP \in NP\text{-completo}$ e quindi che è poco probabile che sia un problema P , allora a quel punto noi sapremo che anche *FTSP*, con bassissima probabilità, sarà risolvibile in tempo polinomiale; a meno che *NP* non collassi su *P* (ma non ci aspettiamo che questa cosa accada)

Mostriamo che *TSP* è *NP-completo*:

1) *TSP* ∈ *NP*?

Sì. Guesso l'HC con peso $\leq k$ (polinomiale). Checkiamo: che i nodi siano effettivamente un ciclo, che il peso complessivo non ecceda k , tutti i nodi siano contenuti esattamente una volta, che non stiamo lasciando nodi per strada. Tutto ciò è fattibile in tempo polinomiale.

2) *TSP* ∈ *NP-Hard*

Per dimostrarlo non faremo una riduzione direttamente, ma partiremo da *3Sat* seguendo una catena di riduzioni:

$$3Sat \leq_P \underbrace{DHC}_{\substack{\text{Directed} \\ \text{Hamiltonian} \\ \text{cycle}}} \leq_P HC \leq_P TSP$$

$DHC = \{G \mid G \text{ è un grafo orientato che ammette un Hamiltonian cycle}\}$

$$3Sat \leq_P DHC$$

$$\phi \xrightarrow{f} G$$

$$3CNF \quad \quad \quad Grafo$$

Un'istanza sì per *3Sat* è una formula che ammette un assegnamento per le sue variabili di verità che la soddisfa; un'istanza sì per *DHC* è un grafo orientato che ammette un Hamiltonian cycle.

Noi dovremo mappare, fare una metafora, per cui "scegliere vero o falso sulle variabili booleane sarà scegliere strade particolari sul grafo"

Il grafo, oltre a dare la possibilità di codificare gli assegnamenti di verità (rispetto alla formula booleana da cui partiamo) dovrà avere una struttura che ha a che fare con le clausole della formula, perché il grafo deve ammettere un Hamiltonian cycle se la formula di partenza è soddisfacibile.

Data una variabile booleana essa ha 2 possibili assegnamenti

Date due variabili booleane hanno 4 possibili assegnamenti

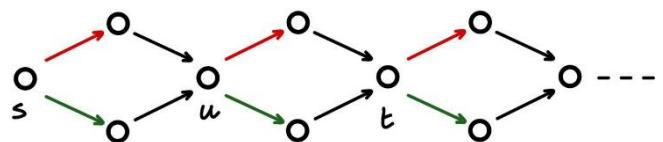
Date n variabili booleane hanno 2^n possibili assegnamenti

Quindi il numero degli assegnamenti

possibili cresce esponenzialmente col numero delle variabili.

Inventiamoci la struttura di un grafo che possa mimare il fatto che abbiamo un numero esponenziale di assegnamenti, quindi deve avere un numero esponenziale di percorsi:

Da s a t ci stanno 4 path perché, quando siamo su s abbiamo due scelte (**R** e **V**, neri sono scelte obbligate) poi anche nel nodo intermedio u andremo o rosso o verde, e se andassimo avanti avremo ancora la scelta... quindi a ogni livello stiamo moltiplicando per due la quantità di path che ci stanno



Quindi noi sfrutteremo un grafo con questa forma, per codificare assegnamenti di verità, cioè la strada che prendiamo ci dice se stiamo dando vero (**Verde**) o falso (**Rosso**).

Questo andrà fatto all'interno della formula:

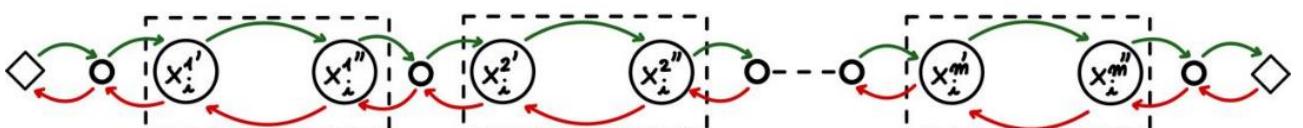
$$\phi = c_1 \wedge \dots \wedge c_m, \text{ quindi } |\phi| = m$$

Quindi abbiamo m clausole su n variabili

$$\bar{x} = \{x_1, \dots, x_n\}$$

Facciamo questa assunzione semplificatrice: "nelle clausole non appare mai un letterale e il suo negato" questo perché quella clausola è trivialmente vera e possiamo non considerarla.

Per ogni $x_i \in \bar{x}$ costruiamo in G una catena di nodi: Abbiamo tante coppie di nodi, quante sono le clausole (m), aggiungiamo dei \diamond intermedi prima e dopo le coppie, Abbiamo poi un nodo \diamond all'inizio e uno alla fine. Colleghiamo il tutto da una catena di archi verdi sopra e una catena di archi rossi sotto.



Adesso dobbiamo replicare la struttura di sopra per riuscire ad avere un numero esponenziale di path.

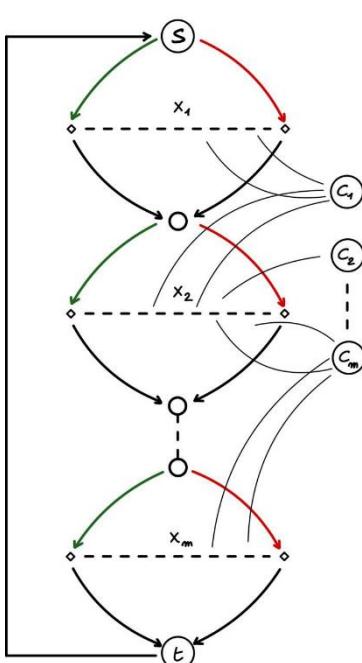
La struttura generale del grafo è questa a sinistra: abbiamo (s) da cui partono un arco **vero** e un arco **falso**

che vanno ad unirsi con la catena x_1 . Gli archi sotto partono dai nodi \diamond e poi continuano a fare la stessa cosa $\forall x_i$ fino alla catena per x_m i cui \diamond si collegano a un nodo (t) che collegiamo a (s).

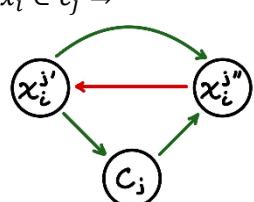
Quindi il grafo, se noi partiamo da s , ha 2^n percorsi. Perché ogni volta che abbiamo la scelta **verde** o **rosso**, il resto è forzato perché, se abbiamo scelto **verde** la catena la dovremo attraversare sugli archi **verdi**, se no ci blocchiamo, se scegliamo **rosso** la catena la dobbiamo attraversare sugli archi **rossi**. Quindi noi facciamo una scelta ad ogni \circ tra catene.

Avremo poi $(c_1), (c_2), \dots, (c_m)$ questi nodi aggiuntivi per le clausole. Questi nodi verranno agganciati alle catene in dipendenza della struttura delle clausole.

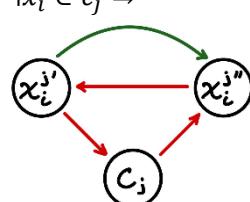
Sia x_i una variabile booleana:



se $x_i \in c_j \Rightarrow$



se $\neg x_i \in c_j \Rightarrow$



Dimostriamo che la formula ϕ è soddisfacibile se e soltanto se questo grafo ha un Hamiltonian cycle:

$\Rightarrow)$ se ϕ è soddisfacibile \Rightarrow un grafo con questa forma ammette un HC

Sia σ un assegnamento che soddisfa ϕ , mostriamo che G_ϕ ammette un HC , ne esibiamo uno:

L' HC è fatto così:

Parte da (s) , se x_1 in σ è vero \Rightarrow percorriamo il ramo vero; quello rosso se falso. Quindi partendo da σ che è un assegnamento di verità che soddisfa la formula, noi seguiamo le istruzioni di σ . Una volta che scendiamo percorriamo la catena (archi verdi se scendiamo dal verde, rossi se scendiamo dal rosso). Se abbiamo la possibilità di fare un *detour* su nodi c_i lo facciamo.

Fino a quando noi non raggiungiamo un \diamond da cui poi scendiamo, andiamo all'altro punto di decisione e guardiamo nuovamente σ , e di nuovo, se σ ci dice falso \rightarrow rosso, se dice vero \rightarrow verde.

A un certo punto arriveremo a (t) , torniamo indietro e andiamo a (s) .

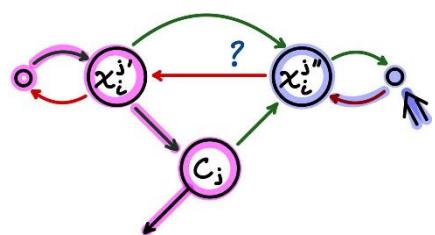
I nodi clausola sono la chiave, perché noi li visitiamo in base all'assegnamento di verità, ed è questo che ci fa "completare" il ciclo per visitare tutti i nodi. \Rightarrow se ϕ è sodd. allora il grafo ammette un HC .

$\Leftarrow)$ Se il grafo ammette un $HC \Rightarrow \phi$ è soddisfacibile.

Dobbiamo mostrare che dall' HC di questo grafo siamo in grado di generare un assegnamento σ per ϕ che la soddisfi; quindi, sia π un HC di G ; π ha due proprietà:

1) se in π arriviamo a (c_j) dal nodo $(x_i^{j'})$ \Rightarrow dopo (c_j) si va in $(x_i^{j''})$

Poniamo il caso di un HC che segue il percorso in magenta: dopo aver visitato c_j andremo a visitare qualcosa in un'altra catena, per completare il ciclo ci manca ancora $x_i^{j''}$ al quale potrò accedere soltanto dal \circ accanto, ma mi troverò in un a strada chiusa poi; quindi, per ottenere un HC dopo aver visitato c_j dovrò per forza visitare $x_i^{j''}$.



2) se in π arriviamo a (c_j) dal nodo $(x_i^{j''})$ \Rightarrow dopo (c_j) si va in $(x_i^{j'})$

Anche qui è simmetrico alla situazione di prima.

Quindi non possono succedere "cose strane" grazie a queste due proprietà quando visitiamo un nodo clausola non cambiamo "catena", e se c_j viene visto dopo $x_i^{j'}$ allora stiamo sugli archi verdi, se c_j viene visto dopo $x_i^{j''}$ allora stiamo sugli archi rossi.

Ciò significa che un HC sensato fa il giro che vogliamo noi, e possiamo ricavarci il σ in questo modo: Sui nodi di scelta guardiamo che arco stiamo prendendo, se andiamo sul verde diamo vero, se andiamo sull'arco rosso in sigma diamo a x_i falso.

Questo ci permette di assegnare tutti i valori di verità a tutte le variabili e siamo sicuri che soddisfa ϕ perché se quello è un HC e visitiamo tutti i nodi clausola con una visita coerente sul colore degli archi, allora se lo vediamo arrivando da rosso è perché abbiamo assegnato falso a un letterale negato, se ci arriviamo da vero è perché abbiamo assegnato vero a un letterale positivo.

Quindi avremo che tutte le clausole sono verificate da quell'assegnamento di verità.

$$\Rightarrow 3Sat \leq_P DHC$$

Riduciamo ora invece

$$\begin{array}{ccc} DHC & \xrightarrow[f]{\leq_P} & HC \\ G & \longrightarrow & H \\ \text{Grafo} & & \text{Grafo} \end{array}$$

La costruzione è la seguente:

Supponiamo di avere due nodi, (in G ci saranno dei nodi, ne dobbiamo ottenere in H) per ognuno dei nodi in G mettiamo tre nodi in H , li chiamiamo $a^{(in)}$, $a^{(mid)}$, $a^{(out)}$ e per b : $b^{(in)}$, $b^{(mid)}$, $b^{(out)}$. Attacchiamo in a mid e mid a out , dopodiché, come abbiamo un arco da $a \rightarrow b$ colleghiamo $a^{(out)}$ e $b^{(in)}$

Questo va fatto per tutti i nodi che stanno in G .

Dobbiamo dimostrare che abbiamo un HC orientato in $G \Leftrightarrow$ c'è un HC non orientato in H .

\Rightarrow Supponiamo che esista un HC orientato in G , lo chiamiamo π , che ha una sequenza di nodi tipo:

$$\pi = v_1, v_2, v_3, \dots$$

Un HC in H che è speculare di v_1, v_2, v_3, \dots può essere γ

$$\gamma = v_1^{(in)}, v_1^{(mid)}, v_1^{(out)}, v_2^{(in)}, v_2^{(mid)}, v_2^{(out)}, v_3^{(in)}, v_3^{(mid)}, v_3^{(out)}, \dots$$

\Leftarrow Mostriamo che, se c'è un HC in $H \Rightarrow$ c'è un HC anche in G

I nodi di H hanno la particolarità che tra $l^{(in)}$ e $l^{(out)}$ ci sono i nodi $l^{(mid)}$, di conseguenza, preso un qualsiasi HC di H , gli apici dei suoi nodi dovranno essere per forza in - mid - out - in - mid - out - in - mid - out oppure, se lo percorriamo al contrario out - mid - in - out - mid - in - out - mid - in perché H non è orientato.

Scegliamo la variante $[in-mid-out]$ avremo:

$$\begin{array}{ccc} v_1^{(in)}, v_1^{(mid)}, v_1^{(out)}, v_2^{(in)}, v_2^{(mid)}, v_2^{(out)}, v_3^{(in)}, v_3^{(mid)}, v_3^{(out)}, \dots \\ \downarrow \quad \downarrow \quad \downarrow \\ v_1 \quad v_2 \quad v_3 \end{array}$$

Io quindi avrò la certezza che c'è un arco tra v_1 e v_2 perché c'è un arco tra $v_1^{(out)}$ e $v_2^{(in)}$ o tra $v_2^{(in)}$ e $v_3^{(in)}$

$$\Rightarrow DHC \leq_P HC$$

Infine, riduciamo:

$$\begin{array}{ccc} HC & \xrightarrow[f]{\leq_P} & TSP \\ G = \langle V, E \rangle & \longrightarrow & \langle H = \langle W, A \rangle, \lambda \rangle, k \\ \text{Grafo non} & & \text{Grafo e funzione} \\ \text{orientato} & & \text{di labelling, numero} \end{array}$$

Dobbiamo trasformare il problema di decidere se un grafo G ha un HC nel problema di sé un grafo H pesato ha un HC di peso al più k .

$$H = G, \text{ mettiamo peso 1 a tutti gli archi e } k = |V|$$

Quindi se G ha un $HC \Rightarrow$ anche H avrà un HC , perché la struttura è la stessa. H ha un HC di peso al più k perché tutti i pesi sono 1 e $k =$ al numero di nodi.

Se H ha un HC di taglia k G avrà per forza un HC per costruzione.

$$\Rightarrow HC \leq_P TSP \Rightarrow TSP \in NP-Hard$$

■