



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Appunti di "*Informatica Teorica*"

A.A. 2024/25

Ivan De Simone

# Introduzione

## Presentazione del corso

Il corso si propone di fornire i concetti principali della teoria della calcolabilità e della complessità.

Per calcolabilità si intende se un dato problema sia risolvibile tramite una procedura algoritmica. Nell'ambito della teoria della calcolabilità introdurremo un modello formale di calcolo che è la macchina di Turing e le sue varianti. Introdurremo il concetto di macchina di Turing universale, che è alla base dei calcolatori programmabili. Vedremo anche dei problemi che non ammettono soluzione algoritmica.

Per complessità si intende, per un problema calcolabile, quante risorse di calcolo siano necessarie a risolverlo. Per risorse di calcolo intendiamo o la memoria o il tempo necessario per risolvere un'istanza del problema. Vedremo quindi che i problemi possono essere classificati in base alla loro complessità e questo ci permetterà di individuare problemi più semplici e problemi più complicati. Si studieranno le metodologie formali che permettono di individuare la complessità intrinseca dei problemi.

Il nostro focus sarà sui problemi piuttosto che sugli algoritmi. Inizieremo distinguendo tra problemi decidibili (risolvibili) e non decidibili. Successivamente, ci concentreremo sui problemi decidibili, classificandoli in *facili*, per i quali esiste un algoritmo polinomiale, e *difficili*, per i quali non esiste un algoritmo polinomiale.

# I problemi

Un **problema** è una relazione che associa stringhe di input a stringhe di output, ovvero un sottoinsieme di tutte le possibili coppie di stringhe `< input, output >`.

Tali possibili coppie sono infinite, quindi elencarle non è una rappresentazione funzionale. Per formalizzare un problema, descriviamo a parole la relazione che lega input e output.

Dobbiamo essere estremamente precisi nel farlo, in quanto potremmo introdurre vaghezza e ambiguità. Per descrivere un problema dobbiamo avere chiari tre elementi:

- **Input**: cosa è l'input, come è rappresentato, che significato ha
- **Output**: cosa è l'output, come è rappresentato, che significato ha
- **Relazione**: qual è la relazione che lega l'input all'output

*Per dare risposta a tali quesiti ci poniamo delle domande internamente.*

Esempio della somma:

- input = due numeri in formato decimale
- output = un numero in formato decimale
- relazione = l'output è la somma aritmetica dei due numeri in input

Questo problema è banale, con problemi più sofisticati non siamo in grado di definire come calcolare l'output a partire dall'input semplicemente descrivendo il problema.

*Nota: la definizione di problema non coincide con la definizione di funzione, in quanto un problema potrebbe avere più output per uno stesso input (pensiamo ai percorsi su un grafo).*

# Calcolabilità dei problemi

## Teoria della calcolabilità

Andremo a dimostrare che i risultati di calcolabilità sono "scolpiti nella pietra", ovvero non dipendono dalla tecnologia degli strumenti fisici che abbiamo a disposizione. Se un problema non è decidibile, non esisterà mai in questo universo uno strumento fisico in grado di implementare una procedura algoritmica che risponda sempre correttamente a tale problema. Per dimostrare ciò, ci baseremo sul **problema dell'arresto** (halt problem).

## Il problema dell'arresto (HALT)

Idea informale: decidere se un determinato programma termina eseguendo con un certo input. Andiamo a modellare il problema:

- Input = una coppia di stringhe  $\langle P, I \rangle$ , dove  $P$  rappresenta il codice di un programma,  $I$  è l'input per il programma  $P$
- Output = un booleano YES o NO
- Relazione = se il programma  $P$  termina eseguendo sulla stringa  $I$ , l'output è YES, altrimenti se non termina l'output è NO.

La definizione del problema è indipendente dal linguaggio di programmazione utilizzato per codificare  $P$ , per comodità adotteremo una sintassi Python.

Un algoritmo che data qualsiasi coppia  $\langle P, I \rangle$  è sempre in grado di rispondere correttamente non esiste. Dimostriamolo per assurdo.

Assumiamo l'esistenza di una procedura `haltChecker(p, i) → bool`, che implementi tale algoritmo. Non stiamo assumendo nulla su linguaggio utilizzato, hardware, ecc.

Definiamo una funzione `reverse(p)` nel seguente modo, che va in loop se  $p$  eseguito con input  $p$  termina, altrimenti termina se  $p$  va in loop.

```
def reverse(p):
    halts = haltChecker(p, p)
    if halts:
        while(true)
    else:
        pass
```

La stringa che rappresenta questo codice la chiamiamo `codeReverse`.

Adesso effettuiamo la chiamata

```
reverse(codeReverse)
```

Tutto ciò che abbiamo scritto fin'ora è lecito e compilabile. Due sono i possibili casi:

- STOP: la funzione esce da `pass`, quindi il valore di `halts` era false. Ma questo significa che il programma `reverse` quando eseguito su se stesso dovrebbe andare in loop.
- LOOP: la funzione è bloccata nel `while(true)`, quindi `halts` aveva valore true. Ma questo vuol dire che `reverse` eseguendo su se stesso dovrebbe terminare.

Abbiamo qui una contraddizione logica. Tutti i passaggi della dimostrazione sono corretti, quindi è falsa l'ipotesi di partenza, ovvero che esista la procedura `haltChecker`.

Non esiste la procedura perfetta: possono esistere approssimazioni che in alcuni casi sbagliano, oppure procedure corrette che limitano il problema di partenza, ad esempio considerando solo linguaggi senza cicli.

## Problemi di ricerca e decisione

Consideriamo due problemi sui grafi:

- Path: trovare un percorso che colleghi il nodo sorgente al nodo destinazione
- Hamiltonian cycle: determinare l'esistenza di un ciclo che passi esattamente una volta su tutti i nodi

I due problemi si differenziano nell'output: Path è un percorso (una lista di nodi) mentre HC è un booleano. Distinguiamo quindi due classi di problemi:

- **Problemi di ricerca:** problemi il cui output può essere vario (percorso, moltiplicazione di matrici, derivate...)
- **Problemi di decisione:** problemi il cui output è un booleano.

Queste due classi non sono completamente slegate, infatti ad ogni problema di ricerca possiamo associare un problema di decisione, limitandoci a decretare l'esistenza di una soluzione, senza fornirla. Data la presenza di questo legame, ci focalizziamo sui problemi di decisione perché sono più semplici.

# Decisione di linguaggi

Per studiare la decidibilità di un problema ci serve qualcosa di più formale, in modo da non perderci nello zucchero sintattico (linguaggi di programmazione, strutture, ...). I problemi possono essere molto vari, ricorriamo quindi alla decisione dei linguaggi per semplificarcici il lavoro.

**Decidere un linguaggio:** data una parola, determinare se essa appartiene ad un certo linguaggio.

Tutti i problemi di decisione si possono ricondurre al problema di decidere un linguaggio. Tale problema viene affrontato tramite automi, così facendo non serve considerare il linguaggio di programmazione o la macchina ospite. Una "stringa sì" è un'istanza del problema che ha come output yes.

## Linguaggi

Un **alfabeto**  $\Sigma$  è un insieme finito di simboli, non per forza lettere.

Una **parola**  $w$  su un alfabeto  $\Sigma$ , è una concatenazione di zero o più simboli provenienti da  $\Sigma$ .

Denotiamo con  $\Sigma^*$  l'insieme di tutte le parole di qualsiasi lunghezza (finita) che si possono costruire su  $\Sigma$ .

Un **linguaggio**  $L$  su un alfabeto  $\Sigma$  è un sottoinsieme di tutte le parole che si possono costruire sull'alfabeto  $\rightarrow L \subseteq \Sigma^*$ .

**Possiamo sempre ricondurre un problema di decisione al problema di decidere un linguaggio**, codificando in maniera opportuna le istanze del problema. Decidere  $L$  significa, data una stringa  $w$ , stabilire se  $w \in L$ .

Il linguaggio  $L$  è parte della definizione del problema ma non fa parte dell'input:

- input = una stringa  $w$  definita sull'alfabeto di  $L$
- output = un booleano
- relazione = data dal linguaggio  $L$

Esempio, problema del percorso su un grafo:

$< g, s, t > \rightarrow \text{yes/no}$

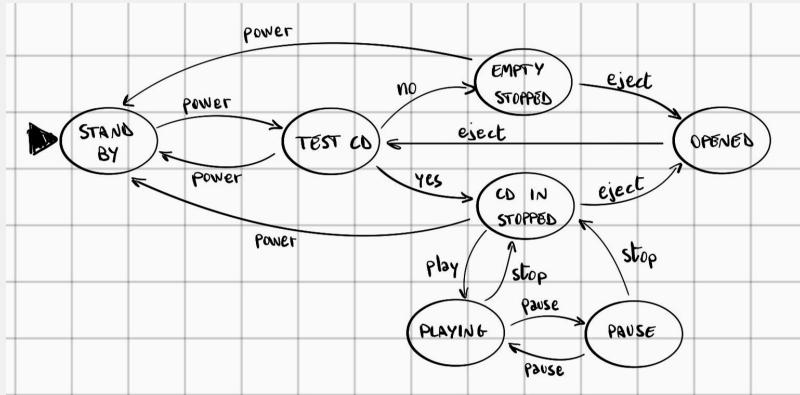
$L_p = \{ < g, s, t > \mid g \text{ è un grafo, } s \text{ e } t \text{ sono nodi di } g, \text{ esiste un percorso da } s \text{ a } t \text{ in } g \}$

La decidibilità del problema viene ricondotta alla capacità di un automa di riconoscere il linguaggio  $L_p$ .

# Automi

Un **automa** è uno strumento con varie modalità di funzionamento, che può reagire a segnali provenienti dall'esterno, in base alla modalità di funzionamento assunta in un determinato istante.

Vediamo come esempio un automa che modella un lettore CD:



Astraiamo questa descrizione nel concetto di automi a stati finiti, ovvero con un numero finito di stati.

Rappresentiamo uno stato con un cerchio ed un evento con una freccia. Il triangolo nero indica lo stato iniziale, il doppio cerchio uno stato finale. Se l'automa si arresta su uno stato finale, la stringa è accettata, altrimenti non è accettata.

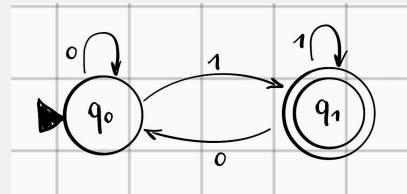
Torniamo ora al nostro problema di decidere un linguaggio, con un esempio:

$L$  = insieme delle stringhe binarie che rappresentano un numero dispari

$$\Sigma = \{0, 1\}$$

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$$

Le stringhe che cerchiamo sono caratterizzate dal finire con 1. Costruiamo quindi un automa che dice yes se la sequenza termina con 1, altrimenti no (anche  $\epsilon$  è no).



## Definizione

Informalmente, possiamo pensare ad un automa come un computer con un nastro di input su cui sono presenti i simboli, che ha la capacità di leggere un simbolo per volta (sequenzialmente, quello che è letto è perso). La sua computazione dipende dal simbolo che legge quando si trova in un determinato stato.

Un **automa a stati finiti**  $D$  è caratterizzato da una quintupla  $\langle \Sigma, Q, q_0, F, \delta \rangle$

- $\Sigma$  è un alfabeto
- $Q$  è un insieme di stati
- $q_0 \in Q$  è lo stato iniziale
- $F$  è l'insieme degli stati finali
- $\delta : Q \times \Sigma \rightarrow Q$  è la funzione di transizione, ci dice come la macchina si sposta da uno stato all'altro. È deterministica: per ogni coppia  $\langle q, \sigma \rangle$ , lo stato successivo è uno solo

La definizione dell'automa è fissata inizialmente e non può cambiare durante l'esecuzione.

Per descrivere il funzionamento di un automa, dobbiamo catturare lo stato di avanzamento della computazione in un determinato istante. Per fare ciò, abbiamo bisogno dello stato attuale e della porzione di input rimanente.

Una **configurazione** per un automa  $D$  è una sequenza di simboli, in cui il primo simbolo è uno stato  $q$ , seguito da altri simboli dell'alfabeto (l'input ancora da leggere).

Esempio con l'automa delle stringhe binarie dispari:

input = "001"

sequenza di configurazioni =  $q_0001 \vdash q_001 \vdash q_01 \vdash q_1\epsilon$

Intuitivamente, possiamo dire che la macchina accetta se e solo se a fine computazione si trova su uno stato finale e sul nastro non è rimasto nessun simbolo da leggere. Di conseguenza, la macchina rifiuta se sul nastro non c'è niente da leggere ma non si trova su uno stato finale, oppure se l'input non è finito ma non ci sono mosse disponibili da fare.

Una **computazione parziale** per un automa  $D$  su una stringa  $w := w_1 \dots w_n$  è una sequenza di  $m + 1$  configurazioni, con  $m \leq n$ ,

$r_0 w_1 \dots w_n \vdash r_1 w_2 \dots w_n \vdash \dots \vdash r_m w_{m+1} \dots w_n$  tale per cui:

- $r_0 = q_0$ , ovvero la partenza è corretta
- $r_{i+1} = \delta(r_i, w_{i+1}) \quad \forall 0 \leq i \leq m - 1$ , cioè solo mosse lecite

Informalmente, è una sequenza di configurazioni in cui la prima ha come stato quello iniziale della macchina, e per ogni configurazione, la successiva è ottenuta applicando la funzione di transizione  $\delta$ .

Una **computazione** è una computazione parziale in cui l'ultima configurazione non ammette passi successivi secondo  $\delta$  e ha come stringa residua  $\epsilon$ .

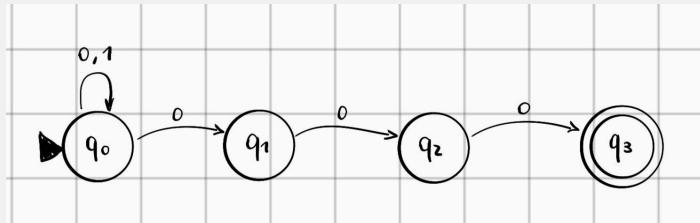
Una computazione su una stringa  $w$  la definiamo **accettante** se nella sua ultima configurazione lo stato è finale e l'input residuo è  $\epsilon$ .

Un automa a stati finiti **accetta una stringa**  $w$  se la computazione che parte dalla configurazione  $q_0 w$  termina in uno stato accettante. L'automa rifiuta la stringa se ciò non accade.

Un automa  $D$  **decide un linguaggio**  $L$  se ogni stringa  $w \in L$  è accettata da  $D$  e ogni stringa  $w \notin L$  è rifiutata.  $D$  decide  $L$  se accetta tutte e sole le stringhe  $w \in L$ .

## Automi non deterministici (NFA)

Supponiamo di voler riconoscere il linguaggio  $L = (0|1)^*000$



Una macchina di questo genere la chiamiamo **non deterministica**, perché ci sono delle configurazioni con più di una configurazione legale successiva. Durante la computazione ci sono dei "punti di scelta" non determinati.

Un **automa a stati finiti non deterministico**  $N$  è definito dalla quintupla  $\langle \Sigma, Q, q_0, F, \delta \rangle$

- $\Sigma$  è un alfabeto
- $Q$  è un insieme di stati
- $q_0 \in Q$  è lo stato iniziale
- $F$  è l'insieme degli stati finali
- $\delta : Q \times \Sigma \rightarrow 2^Q$  è la relazione di transizione, ci dice come la macchina si sposta da uno stato all'altro. Non è deterministica: per ogni coppia  $\langle q, \sigma \rangle$ , lo stato successivo può essere più di uno

Abbiamo due differenze fondamentali rispetto agli automi deterministici. La prima è che in una computazione parziale, una configurazione può seguirne un'altra se il suo stato appartiene all'insieme degli stati raggiungibili da quest'ultima secondo la relazione di transizione, ovvero  $r_{i+1} \in \delta(r_i, w_{i+1})$ . La seconda è che un automa non deterministico  $N$  accetta una stringa  $w$  se esiste una computazione accettante di  $N$  su  $w$ . L'intuizione è che  $N$  accetta una stringa  $w$  se esiste un modo per accettarla. Ne consegue che un automa non deterministico può avere computazioni diverse per una stessa stringa.

Un automa non deterministico non esiste fisicamente: è un modello astratto di calcolo creato per studiare i linguaggi. Lo pensiamo come una macchina che quando arriva ad un punto di scelta, se c'è una strada corretta, la prende.

Un risultato (che non dimostreremo) prova che gli automi deterministici e non deterministici sono equivalenti, ovvero possiamo trasformare uno nell'altro. Sfruttiamo questo risultato per costruire automi non deterministici, che poi sappiamo potranno essere "tradotti" in deterministici.

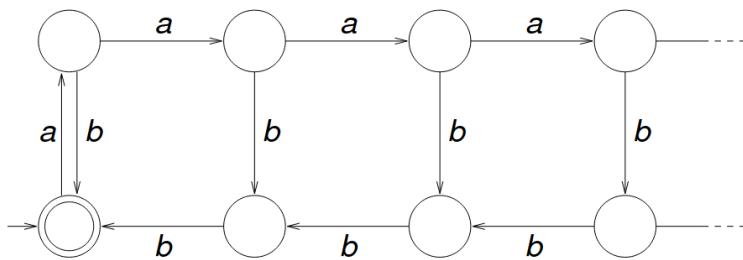
Bisogna definire attentamente la relazione di transizione  $\delta$  per far sì che una stringa da rifiutare non abbia nessun modo di essere accettata.

## Limiti degli automi

Supponiamo di voler riconoscere il seguente linguaggio:

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

Tale linguaggio non può essere accettato da un automa a stati finiti, in quanto non è in grado di contare  $\rightarrow$  ci servirebbero infiniti stati.



Dimostriamolo per assurdo:

Assumiamo di poter costruire un DFA in grado di riconoscere  $L$ .

Diciamo che l'automa ha un numero  $p$  di stati (essendo finiti).

L'automa deve essere in grado di riconoscere tutte le stringhe  $\in L$ .

Gli passiamo in input una stringa di lunghezza  $x$ , molto superiore a  $p$ .

Non avendo  $x$  stati differenti, almeno uno degli stati si deve ripetere  $\rightarrow$  esiste un ciclo.

Siccome l'automa non può contare, può riconoscere stringhe con un numero di  $a$  e  $b$  diverso.

Gli automi così come li abbiamo definiti, non sono in grado di calcolare a sufficienza. Abbiamo bisogno di un formalismo più potente, che è la macchina di Turing.

## Linguaggi regolari

Introduciamo ora la notazione dei linguaggi regolari:

Sia  $\Sigma$  un alfabeto. Un **linguaggio regolare** su  $\Sigma$  è un linguaggio ottenuto tramite concatenazione di simboli in  $\Sigma$ . Le stringhe appartenenti ad un linguaggio regolare possono essere espresse in modo compatto utilizzando le **espressioni regolari**.

Sia  $\alpha \in \Sigma$ , allora  $\alpha$  è un'espressione regolare.

- se  $\alpha$  e  $\beta$  sono espressioni regolari  $\implies \alpha\beta$  è un'espressione regolare.
- se  $\alpha$  e  $\beta$  sono espressioni regolari  $\implies \alpha | \beta$  è un'espressione regolare.
- se  $\alpha$  è un'espressione regolare  $\implies \alpha^*$  è un'espressione regolare. Inoltre,  $\alpha^+$  è un'espressione regolare (esclude  $\epsilon$ ).

Esiste un risultato il quale dimostra che DFA e NFA possono riconoscere tutte e sole le espressioni regolari.

Il linguaggio  $\{a^m b^m \mid m \geq 0\}$  non è esprimibile tramite espressioni regolari.

## Macchine di Turing

Una **macchina di Turing** (MT) è una macchina astratta per modellare il calcolo. Nasce da un'idea di Alan Turing per mostrare cosa fosse calcolabile automaticamente.

Una MT è un automa a stati finiti, dotato di un nastro di input infinito in entrambe le direzioni. Tale nastro è diviso in celle, le quali contengono i simboli di input. Poiché la stringa si trova su un nastro infinito, usiamo il simbolo *blank* ( $\emptyset$ ) per delimitarla. La MT ha uno stato interno, con cui mantiene una codifica del programma tramite stati.

Una MT si distingue da un automa a stati finiti per quello che può fare con il nastro. La testina della MT è in grado di leggere e scrivere simboli (anche sovrascrivendo) e si può muovere sia a destra che a sinistra.

Il suo flusso di funzionamento è il seguente:

1. legge un simbolo
2. fa una transizione verso un altro stato
3. scrive sul nastro
4. sposta la testina a dx o sx

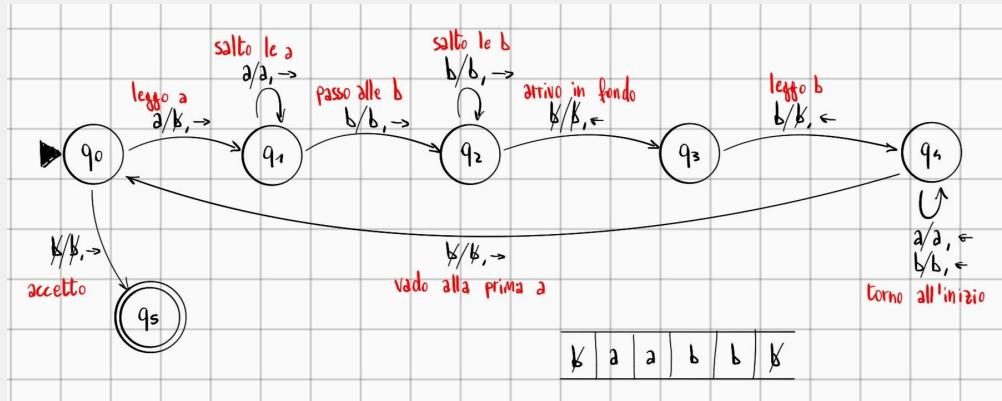
Per costruire una MT conviene pensare ad un filmato di quello che deve succedere alla testina sul nastro, e da questo tirare fuori un grafo di transizione.

Facciamo un esempio con il linguaggio precedente.

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

Filmato: leggo la prima *a*, la cancello, cerco l'ultima *b*, la cancello e torno alla prima *a*

*(che ora è quella successiva rispetto a prima).*



Rappresentiamo graficamente una MT come gli automi a stati finiti, caratterizzando le transizioni con  $\alpha/\beta$ ,  $\rightarrow$  o  $\alpha/\beta, \leftarrow$  per indicare "leggo  $\alpha$ , scrivo  $\beta$  e sposto la testina a ...".

## Definizione

Una **macchina di Turing**  $M$  è definita da una settupla  $\langle \Sigma, \Gamma, \mathcal{B}, Q, q_0, F, \delta \rangle$

- $\Sigma$  è l'alfabeto di input
  - $\Gamma \supseteq \Sigma$  è l'alfabeto di nastro, ovvero tutti i simboli che possono comparire sul nastro
  - $\emptyset \in \Gamma$  è il simbolo di vuoto, *blank*
  - $Q$  è un insieme finito di stati
  - $q_0 \in Q$  è lo stato iniziale
  - $F \subseteq Q$  è l'insieme degli stati finali
  - $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$  è la funzione di transizione, ci dice come la macchina si sposta da uno stato all'altro, come modifica il nastro e come muove la testina. È deterministica: per ogni coppia  $< q, \gamma >$ , il passo successivo è uno solo

Per catturare lo stato di avanzamento dell'esecuzione di una MT dobbiamo sapere lo stato attuale, cosa c'è sul nastro e dove si trova la testina.

Una **configurazione** per una MT si differenzia da quella per gli automi nella posizione dello stato, che viene inserito nel punto in cui si trova la testina, quindi non necessariamente all'inizio. I *blank* nella configurazione li mettiamo solo se sono indispensabili, ad esempio all'inizio e alla fine.

## Esempi:

$aq_1bb \rightarrow$  siamo nello stato  $q_1$  e stiamo leggendo il simbolo  $b$

*abbq, b oppure q<sub>1</sub>bbb*

Date due configurazioni  $c_1$  e  $c_2$  per una MT  $M$ , diciamo che  $c_2$  è il **legal successor** di  $c_1$  rispetto a  $M$ , e lo scriviamo  $c_1 \vdash_M c_2$ , se  $c_2$  è la configurazione che  $M$  raggiunge partendo da  $c_1$ , facendo un solo passo secondo  $\delta$ .

Esempio:

$$\underbrace{q_0aabb}_{c_1} \vdash_M \underbrace{q_1abb}_{c_2}$$

La **configurazione iniziale** di  $M$  su una stringa  $w := w_1 \dots w_n$  è  $q_0 w_1 \dots w_n$ .

Una **configurazione finale** per  $M$  è una configurazione che non ammette nessun legal successor secondo  $\delta$ .

Una **configurazione accettante** per  $M$  è una configurazione finale il cui stato è accettante.

Una **computazione parziale** di  $M$  è una sequenza di configurazioni  $c_1 \dots c_m$  tali che  $c_1 \vdash_M \dots \vdash_M c_m$ .

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

Una computazione di  $M$  su  $w$  è **accettante** se e solo se  $c_m$  è una configurazione accettante.

Il **linguaggio di una MT  $M$** , denotato come  $\mathcal{L}(M)$ , è l'insieme di tutte le stringhe  $w$  accettate da  $M$ .  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M(w) = 1\}$ .

Una MT  $M$  **decide** il linguaggio  $L$  se e solo se, per ogni stringa  $w$ :

- $w \in L \implies M(w) = 1$
- $w \notin L \implies M(w) = 0$

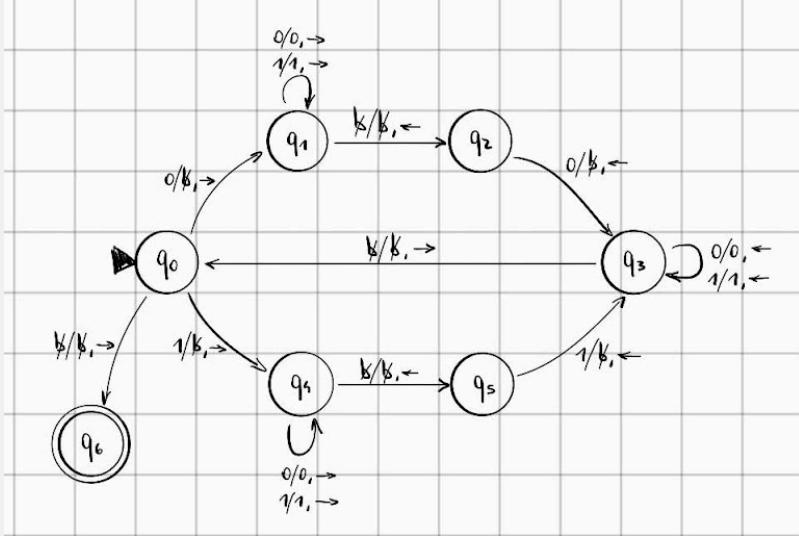
Una MT  $M$  **accetta** il linguaggio  $L$  se e solo se, per ogni stringa  $w$ :

- $w \in L \implies M(w) = 1$
- $w \notin L \implies M(w) = ?$

Il fatto che una MT *decida* piuttosto che *accetti* è una garanzia più forte, in quanto abbiamo la certezza di ricevere una risposta. Nel caso dell'accettazione, in presenza di un rifiuto potremmo non avere risposta.

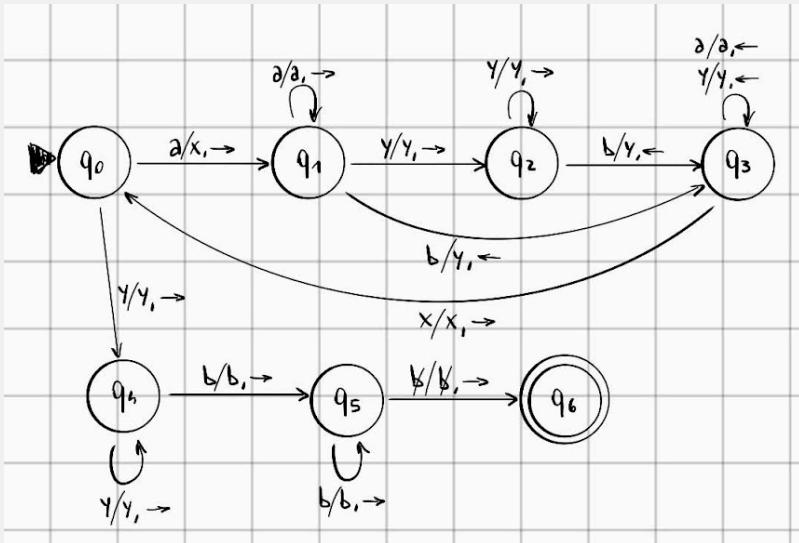
## Esercizi MT

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



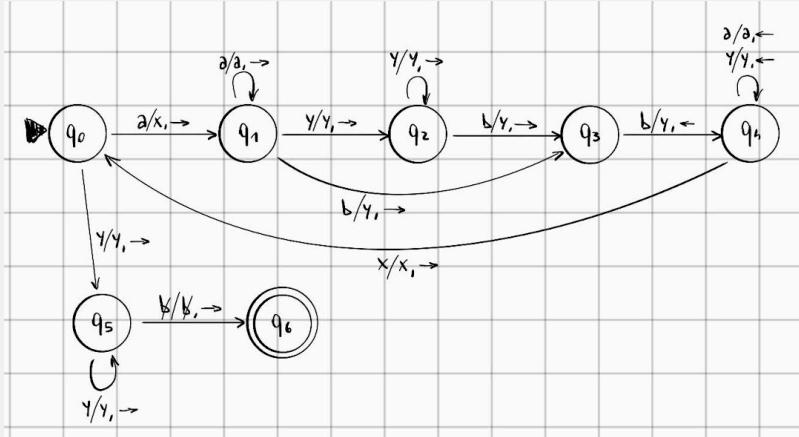
2.  $L = \{a^n b^m \mid m > n > 0\}$

Conviene non usare i blank per non perdere l'input.

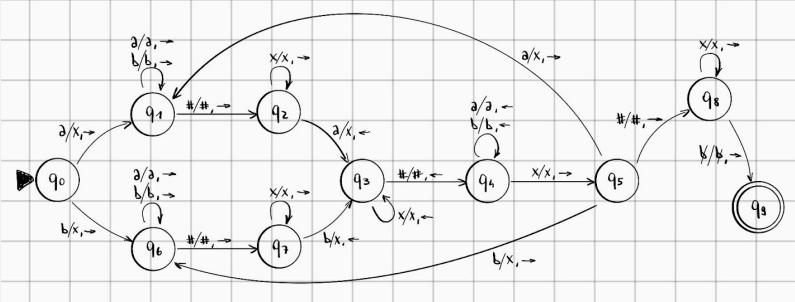


3.  $L = \{a^n b^m \mid n > 0, m = 2n\}$

Stesso ragionamento di prima, marcando 2 b anziché una.

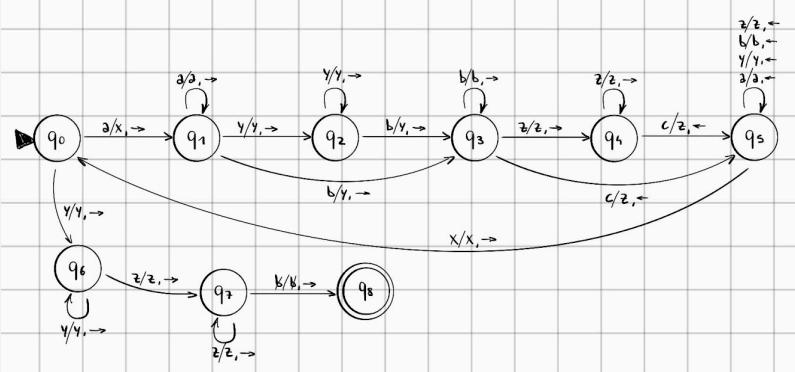


4.  $L = \{w\#w \mid w \in (a \mid b)^+\}$



5.  $L = \{a^n b^n c^n \mid n > 0\}$

Leggiamo una a, cerchiamo una b e una c, poi riavvolgiamo.



## Varianti più sofisticate

Le macchine di Turing classiche sono semplici da definire, ma difficili da programmare. Di seguito ne vediamo delle varianti via via più sofisticate.

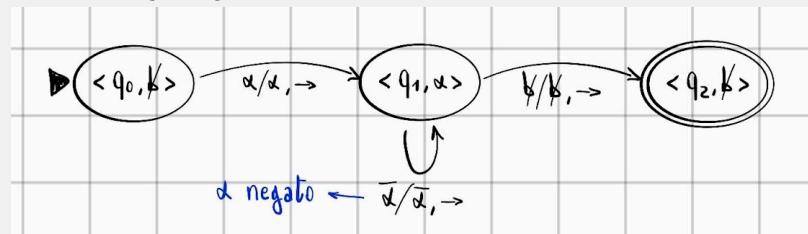
### Macchine con memoria

Introduciamo un modello di macchina dotato di una memoria nello stato, simile a dei registri, dove memorizzare dei simboli. Un registro interno è una coppia  $\langle q_i, x \rangle$ , dove il primo membro ci dice lo stato della macchina e il secondo memorizza un simbolo (solo uno). Non c'è un limite al numero dei registri, che viene però fissato alla definizione della macchina, quindi la memoria non può crescere dinamicamente.

Supponiamo di voler riconoscere il linguaggio  $L = 10^* \mid 01^*$

Leggiamo il primo simbolo, lo mettiamo in memoria e dopo ci aspettiamo di leggere solo simboli diversi da esso.

Sia  $\alpha \in \{0, 1\}$



Queste macchine hanno lo stesso potere espressivo di quelle classiche, in quanto sono solamente un modo compatto di scrivere una MT più grossa.

## Macchine multi-traccia

Definiamo ora le macchine multi-traccia: un'evoluzione delle MT con memoria, in cui il nastro è diviso orizzontalmente in più tracce. Ogni traccia contiene al massimo un simbolo per cella. Abbiamo una testina capace di leggere tutte le tracce contemporaneamente, muovendosi in blocco sopra il nastro (quindi le legge tutte alla stessa posizione). Come per i registri, il numero di tracce è illimitato, ma fissato al momento della definizione.

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

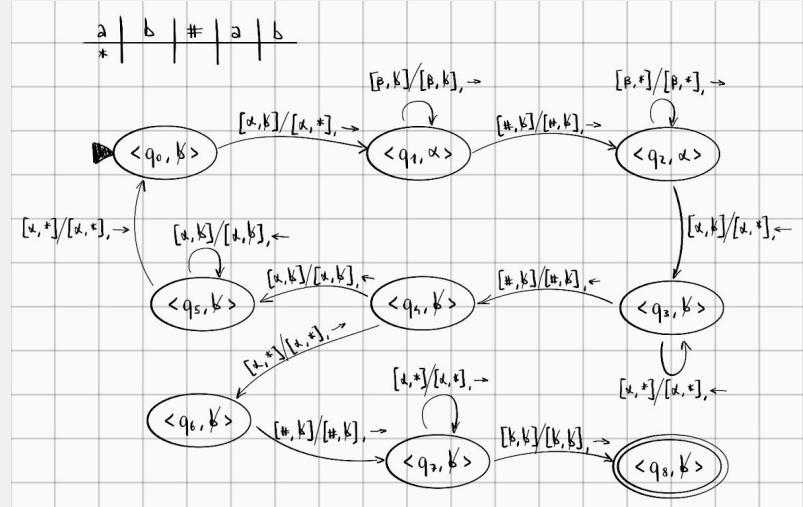
Agisce come una MT classica, con la possibilità di leggere e scrivere tutte le tracce allo stesso momento.

Graficamente usiamo la stessa rappresentazione delle MT classiche, indicando i simboli letti/scritti con una lista relativa alle diverse tracce.

Vogliamo riconoscere il linguaggio  $L = \{w\#w \mid w \in (a \mid b)^+\}$

*Avanzando con la computazione, marchiamo con un asterisco nella seconda traccia i simboli letti.*

Siano  $\alpha, \beta \in \{a, b\}$



Il principale vantaggio è che stiamo preservando l'input man mano che avanziamo, mentre con le macchine precedenti andava perso sovrascrivendolo.

Le macchine mono-traccia e multi-traccia hanno lo stesso potere espressivo. Si può dimostrare con due approcci differenti:

1. Possiamo definire una MT mono-traccia in cui usiamo un alfabeto più ricco per rappresentare tutte le possibili combinazioni dei simboli sulle tracce multiple.

2. Possiamo simulare una macchina multi-traccia rielaborando la traccia di input. All'inizio espandiamo l'input inserendo tanti spazi quante sono le tracce aggiuntive tra un simbolo e l'altro. Questi spazi saranno poi usati per scrivere ciò che la multi-traccia scriverebbe nelle altre tracce.

È quindi possibile dire che le due macchine sono equivalenti.

## Macchine multi-nastro

Introduciamo adesso le macchine multi-nastro, le quali sono dotate di più nastri completamente indipendenti, ognuno con la propria testina. La cosa interessante è che possiamo spostare le testine a piacimento, anche in posizioni diverse l'una dalle altre.

Introduciamo inoltre la possibilità di tenere ferma una testina, con il simbolo - (potevamo simularlo anche prima con un movimento sx-dx).

La differenza rispetto alle macchine multi-traccia è sostanziale, poiché con queste ultime siamo vincolati a leggere la stessa posizione su tutte le sequenze, mentre con le macchine multi-nastro ci possiamo muovere liberamente nelle posizioni che ci interessano.

Assumiamo che all'avvio l'input della macchina si trovi sul primo nastro, che la testina del nastro di input si trovi sul primo simbolo e che gli altri nastri, chiamati *work tapes*, siano completamente vuoti.

Come nei casi precedenti, il numero di nastri è teoricamente infinito, ma fissato in fase di progettazione.

**Notazione per le etichette:** n° nastro: operazione .

Su tutti i nastri non menzionati non eseguiamo operazioni.

Separiamo le diverse etichette con delle graffe (come i sistemi).

Esempio:

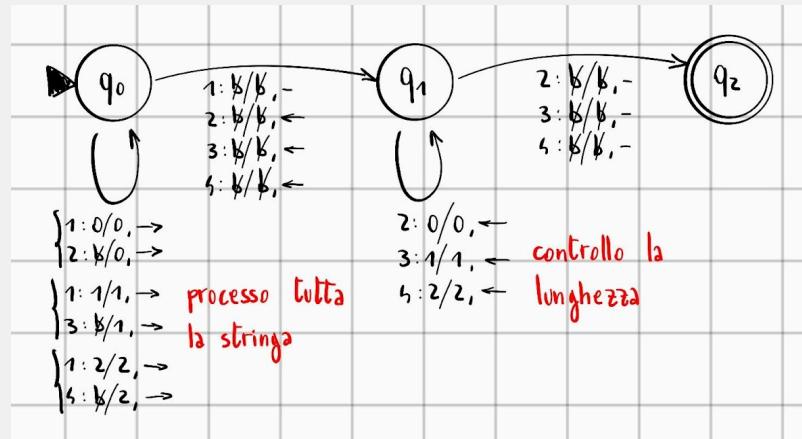
$$\begin{cases} 1 : 0/0, \rightarrow \\ 2 : \emptyset/0, \rightarrow \end{cases}$$

*Sul primo nastro c'è 0, scrivo 0 e vado avanti, sul secondo c'è blank, scrivo 0 e vado avanti.*

Vogliamo riconoscere il seguente linguaggio:

$L = \{w \mid w \in (0 \mid 1 \mid 2)^*, \text{ il numero di } 0, 1 \text{ e } 2 \text{ è lo stesso in } w\}$

Ricopiamo un tipo di simbolo per nastro e dopo processiamo al rovescio per verificare la lunghezza.



Il **tempo di computazione** di una MT su una certa stringa è pari al numero di passi che la macchina effettua prima di arrestarsi, ovvero quante transizioni di stato ha eseguito (la lunghezza della sua computazione per essere assolutamente precisi). Se la macchina non si arresta il suo tempo di esecuzione è infinito (unbounded).

Notiamo che è una definizione molto più precisa di quella data per la complessità computazionale degli algoritmi. Tramite questa analisi si può ad esempio dimostrare che moltiplicare due numeri è più complesso che sommarli.

Possiamo dimostrare che il comportamento di una MT multi-nastro è simulabile da una MT multi-traccia, al costo di una certa perdita di tempo.

### Teorema

Sia  $M$  una MT multi-nastro  $\implies \exists$  una MT multi-traccia  $S$  tale che  $\mathcal{L}(M) = \mathcal{L}(S)$ .

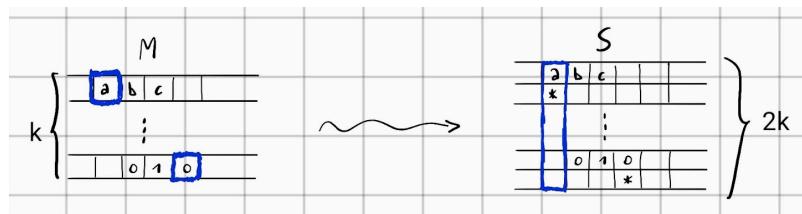
Ovvero, data una MT  $M$ , siamo sempre in grado di costruire una MT  $S$  equivalente.

### Dimostrazione

Supponiamo di avere una MT  $M$  con  $k$  nastri.

La simuliamo tramite una MT  $S$  che ha  $2 \cdot k$  tracce.

Metà delle tracce verranno usate per memorizzare il contenuto dei nastri della macchina  $M$ , le tracce restanti conterranno dei marcatori per indicare la posizione della testina del nastro corrispondente.



Partendo da sx, spostiamo la testina di  $S$  verso dx segnando nello stato i simboli letti in corrispondenza dei  $k$  marcatori. A questo punto sappiamo che operazione effettuare: cosa scrivere sulle tracce e dove spostare i marcatori. Fatto ciò, torniamo a sx e ricominciamo. Il tutto per simulare un singolo passo della macchina  $M$ .

Abbiamo dimostrato che le macchine multi-nastro e multi-traccia hanno lo stesso potere espressivo. Cerchiamo ora di stimare quanto è più lenta la macchina  $S$  rispetto a  $M$ .

Nel caso peggiore ci troviamo due nastri di  $M$  con le testine a  $m$  passi di distanza, rispettivamente a sx e dx. Quindi nella macchina  $S$ , per spostarci, dobbiamo effettuare  $2 \cdot m$  passi ad andare avanti e  $2 \cdot m$  passi a tornare indietro, più altri 2 passi per ogni marcatore, nel caso debba essere spostato nella direzione opposta al movimento delle testine. In totale  $S$  effettua  $4 \cdot m + 2 \cdot k$  passi per ogni passo di  $M$ .

Supponiamo che la macchina  $M$  faccia  $n$  passi.

$$\sum_{i=1}^n 4i + 2k \leq \sum_{i=1}^n 4n + 2k \leq n \cdot (4n + 2k) \rightarrow O(n^2)$$

Arriviamo ad ottenere un ordine quadratico: per una computazione,  $S$  ci mette almeno il quadrato del tempo che ci mette  $M$ .

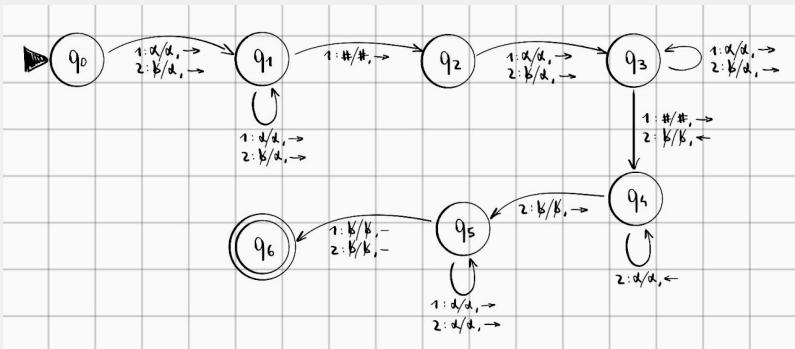
In conclusione, le MT multi-traccia e multi-nastro sono equivalenti, ma quest'ultime sono più veloci. Noi ci concentreremo sulle MT multi-nastro in quanto sono più semplici da programmare.

## Esercizi MT multi-nastro

$$1. L = \{A\#B\#AB \mid A, B \in (0 \mid 1)^+\}$$

Sia  $\alpha \in \{0, 1\}$

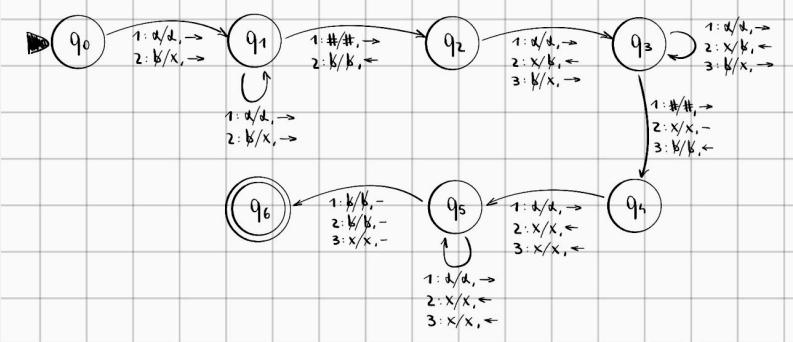
*Copiamo tutta A sul nastro 2, ci concateniamo tutta B, poi riavvolgiamo e confrontiamo con AB.*



$$2. L = \{A \# B \# C \mid A, B, C \in (0 \mid 1)^+, |A| > |B| > |C|, |C| = |A| - |B|\}$$

Sia  $\alpha \in \{0, 1\}$

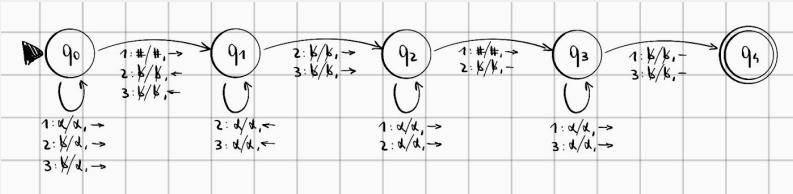
Segniamo la lunghezza di  $A$  sul nastro 2 e di  $B$  sul nastro 3, verifichiamo che  $|A| > |B|$ , controlliamo che  $|C|$  sia uguale alla differenza di  $|A|$  e  $|B|$  e poi che  $|C| < |B|$ .



$$3. L = \{w \# w \# w \mid w \in (0 \mid 1)^*\}$$

Sia  $\alpha \in \{0, 1\}$

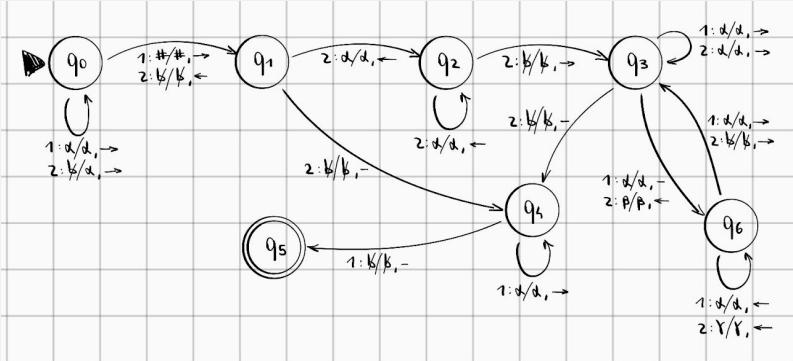
Copiamo la prima  $w$  sui nastri 2 e 3, li riavvolgiamo, poi confrontiamo la prima  $w$  con la seconda e la terza. Usiamo 3 nastri per non dover riavvolgere dopo la seconda  $w$ .



$$4. L = \{A \# B \mid A, B \in (a \mid b)^*, A \subseteq B\}$$

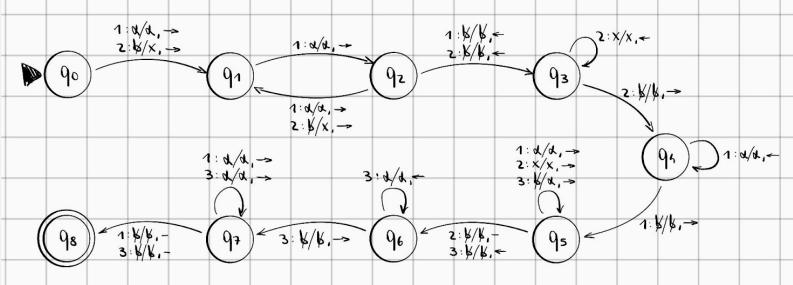
Siano  $\alpha, \beta, \gamma \in \{a, b\}$ ,  $\alpha \neq \beta$

Copiamo  $A$  sul nastro 2, verifichiamo se  $A$  è parte di  $B$ , se non lo è riavvolgiamo entrambi sfasando di una posizione le testine prima di ricontrillare.



$$5. L = \{ww \mid w \in (0 \mid 1)^+\}. \text{ Sia } \alpha \in \{0, 1\}$$

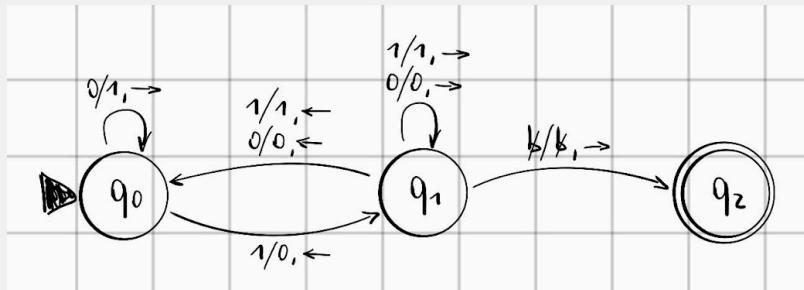
Nel nastro 2 scriviamo una  $x$  ogni due simboli di input, poi usiamo questa lunghezza per verificare che le due mezze stringhe siano identiche.



# Macchine di Turing non deterministiche

Introduciamo ora le macchine di Turing non deterministiche, un ulteriore modello astratto di calcolo, ancora più pratico da programmare. Il non determinismo in tali macchine è dato dalla possibilità di avere più stati verso cui transire, per ogni coppia *<stato attuale, simbolo letto>*.

Esempio: assumiamo di avere la seguente macchina



La definiamo non deterministica in quanto da  $q_1$  abbiamo più scelte per uno stesso simbolo letto.

## Definizione

Una **macchina di Turing non deterministica**  $N$  è definita da una settupla  $\langle \Sigma, \Gamma, \emptyset, Q, q_0, F, \delta \rangle$  in cui:

- $\Sigma$  è l'alfabeto di input
- $\Gamma \supseteq \Sigma$  è l'alfabeto di nastro, ovvero tutti i simboli che possono comparire sul nastro
- $\emptyset \in \Gamma$  è il simbolo di vuoto, *blank*
- $Q$  è un insieme finito di stati
- $q_0 \in Q$  è lo stato iniziale
- $F \subseteq Q$  è l'insieme degli stati finali
- $\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{\leftarrow, \rightarrow\}}$  è la relazione di transizione, ci dice come la macchina si sposta da uno stato all'altro, come modifica il nastro e come muove la testina. Non è deterministica: per ogni coppia  $\langle q, \gamma \rangle$ , possiamo avere più di un passo successivo

Un esempio di transizione usando la macchina precedente:

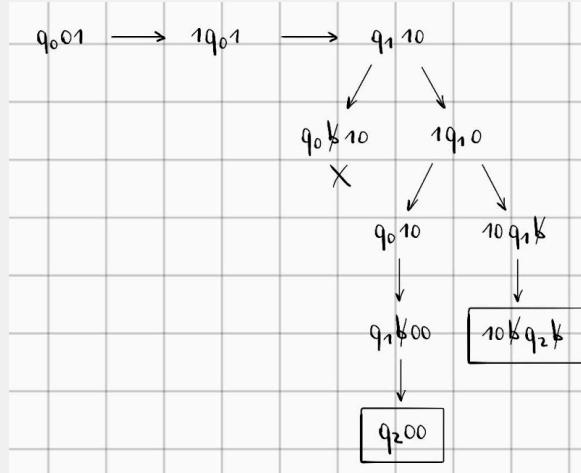
$$\delta(q_1, 0) = \{(q_0, 0, \leftarrow), (q_1, 0, \rightarrow)\}$$

Le definizioni di configurazione sono le stesse delle MT deterministiche. Le macchine non deterministiche si differenziano per la possibilità che una configurazione abbia più di un legal successor. Tuttavia, non è necessario che tutte le configurazioni abbiano del non determinismo.

Un **computation tree** (albero di computazione) per una MT non deterministica è un albero che cattura tutte le possibili computazioni effettuabili dalla macchina su una stringa in input. I nodi rappresentano tutte le configurazioni in cui la macchina può trovarsi processando la stringa  $w$ . La radice è la configurazione iniziale, le foglie sono le configurazioni finali. È presente un arco da  $\alpha$  a  $\beta$  se  $\beta$  è uno dei legal successor di  $\alpha$ .

Un esempio di computation tree usando la macchina precedente:

Sia  $w = 01$



Una MT  $N$  non deterministica accetta l'input  $w$  se e solo se all'interno del computation tree di  $N$  su  $w$  è presente una configurazione accettante (non ci interessa dove). La macchina rifiuta se non c'è nessun modo per accettare  $w$ .

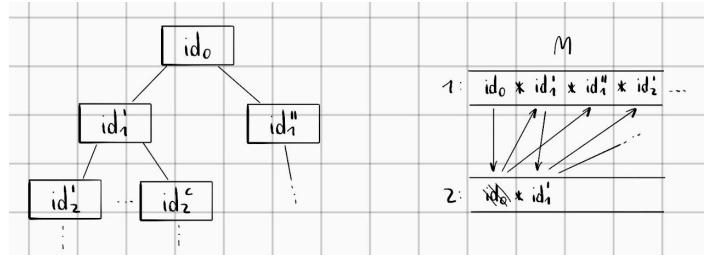
## Equivalenza con le MT deterministiche

Vogliamo dimostrare che le MT non deterministiche sono equivalenti alle MT deterministiche, ovvero stabilire se, avendo una MT non deterministica che accetta un linguaggio  $L$ , esista una MT deterministica che accetti lo stesso linguaggio  $L$ .

Nota: i risultati di equivalenza raggiunti per le varianti più sofisticate valgono anche per le MT non deterministiche.

Supponiamo di avere una MT non deterministica  $N$ , per semplicità mono-nastro. Vogliamo simulare  $N$  tramite una MT deterministica  $M$  avente due nastri, con cui analizzeremo parallelamente le computazioni di  $N$ .

Supponiamo che il computation tree di  $N$  su una stringa  $w$  sia il seguente:



Andiamo a progettare  $M$ , conoscendo a priori il funzionamento di  $N$  ( $M$  non potrà calcolare più di  $N$ , in quanto la sua funzione di transizione è generata appositamente):

1. Scriviamo sul primo nastro la configurazione iniziale, poi copiamola sul secondo nastro.
2. Sapendo che  $id_0$  ha due legal successor, copiamo due volte  $id_0$  sul primo nastro.
3. Sostituiamo queste ultime due copie con i loro legal successor, secondo la relazione di transizione di  $N$ .
4. Eliminiamo  $id_0$  dal secondo nastro e ripartiamo avanzando le configurazioni.

Ripetiamo questi passi per ogni configurazione presente sul primo nastro, ci fermiamo se troviamo una configurazione accettante.

Praticamente stiamo esplorando l'albero in ampiezza. Non lo facciamo in profondità poiché potrebbe andare in loop (se  $N$  rifiuta a causa di un ciclo).

Siccome un qualsiasi linguaggio accettato da una MT non deterministica è accettato anche da una MT deterministica, possiamo affermare che le due macchine hanno lo stesso potere espressivo, ovvero sono equivalenti. Quanto costa simulare una macchina non deterministica tramite una deterministica?

Supponiamo che la relazione di transizione abbia al massimo  $c$  figli. Se volessimo simulare  $n$  passi della macchina non deterministica, dovremmo esplorare  $n$  livelli, vale a dire  $c^n$  nodi. Abbiamo quindi un ordine esponenziale:  $O(c^n)$ .

Il fatto che questa simulazione abbia un costo esponenziale non significa che non esistono simulazioni più veloci. È uno dei problemi ancora irrisolti dell'informatica teorica: non è detto che esista, ma non è provato che non esista.

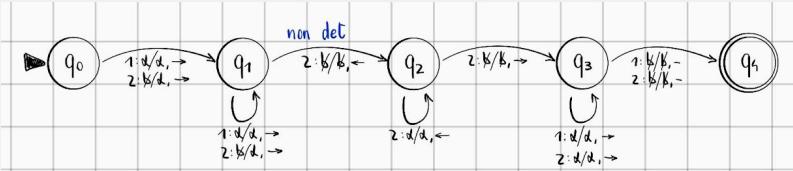
## Esercizi MT non deterministiche

Per progettare MT non deterministiche seguiamo un design pattern detto **guess and check**, secondo cui la macchina ad un certo punto "indovina" la scelta da prendere, dopodiché controlla che sia effettivamente corretta. Questa è solo una metafora che pensiamo noi, in realtà la macchina non è in grado di fare niente del genere. Quello che succede in realtà, nella computazione astratta, è che tramite il check andiamo a filtrare i rami del computation tree che non portano ad una configurazione accettante. È cruciale assicurarsi che non vengano accettati input scorretti, per questa ragione non si usano etichette multiple durante il check (c'è il rischio di filtrare male).

$$1. L = \{ ww \mid w \in (0 \mid 1)^+ \}$$

Sia  $\alpha \in \{ 0, 1 \}$

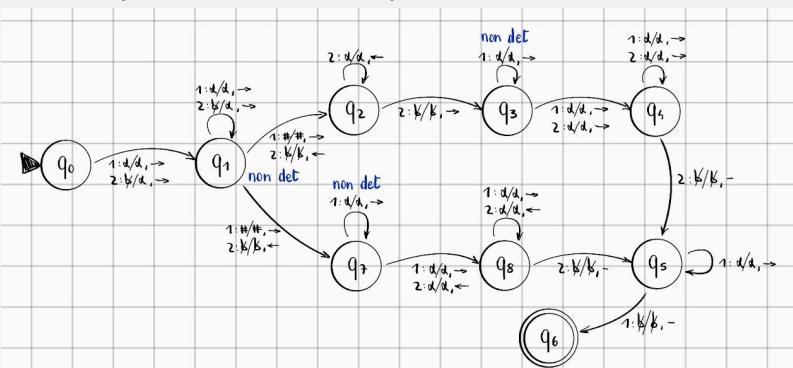
Copiamo sul secondo nastro fino a che non indoviniamo di essere a metà, poi controlliamo l'uguaglianza.



$$2. L = \{ A \# B \mid A, B \in (0 \mid 1)^+, A \subseteq B \vee A^R \subseteq B \}$$

Sia  $\alpha \in \{ 0, 1 \}$

Copiamo  $A$ , indoviniamo se cercare  $A$  oppure il reverse all'interno di  $B$ , indoviniamo da dove partire a cercare e poi controlliamo.



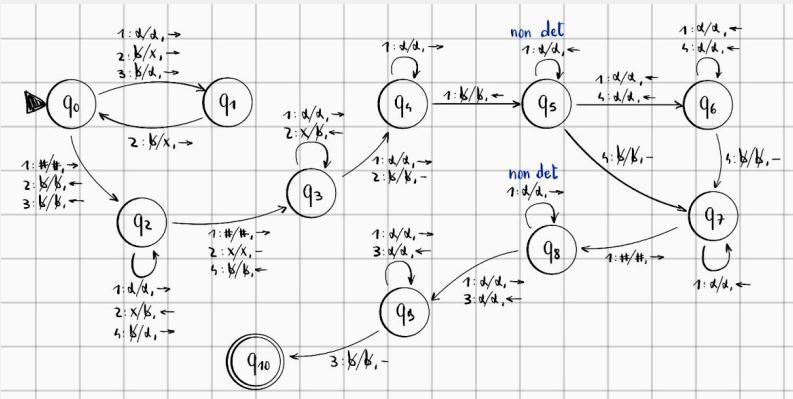
$$3. L = \{ A \# B \# W \mid A, B, W \in (0 \mid 1)^*, |W| > 2|A| - |B|,$$

$$|B| < 2|A|, A^R \subseteq W \wedge B \subseteq W \}$$

Sia  $\alpha \in \{ 0, 1 \}$

Segniamo  $2|A|$  sul secondo nastro, confrontiamo con  $|B|$ , verifichiamo che  $|W|$  sia maggiore del rimanente, indoviniamo dove iniziare a cercare  $A$  reverse e controlliamo, indoviniamo dove iniziare a cercare  $B$  e controlliamo.

Nastri: 1 = input, 2 =  $2|A| - |B|$ , 3 =  $A$ , 4 =  $B$



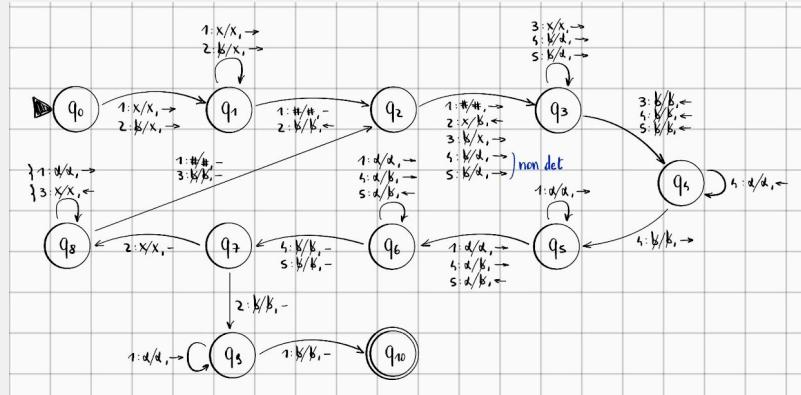
$$4. L = \{ x^n \# W_1 \# \dots \# W_n \mid W_i \in (a \mid b \mid c \mid d)^+, n > 0,$$

$$\forall i \ 1 \leq i \leq n \ \exists s_i : s_i \subseteq W_i, |s_i| = i, s_i = s_i^R \}$$

Sia  $\alpha \in \{a, b, c, d\}$

Calcoliamo  $n$ , per ogni  $i$  indoviniamo la stringa  $s$ , indoviniamo dove iniziare a cercare  $s$  dentro  $W$ , poi controlliamo verificando anche che  $s$  sia palindroma.

Nastri: 1 = input, 2 =  $n$ , 3 =  $i$ , 4 =  $s_i$ , 5 =  $s_i$  (per reverse)



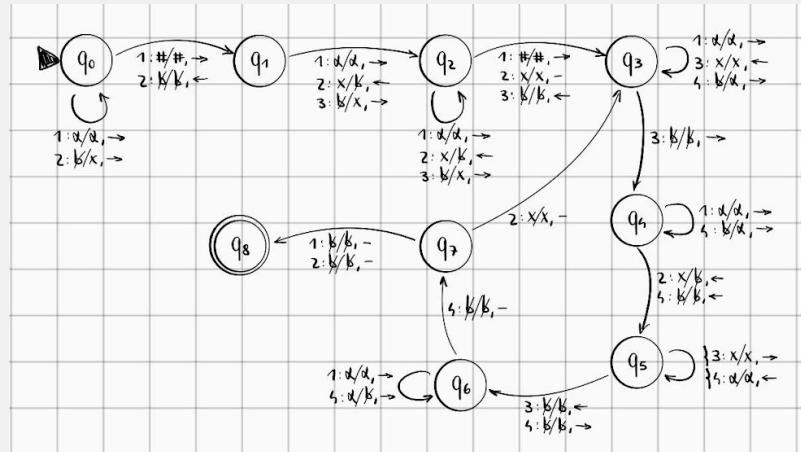
$$5. L = \{ A \# B \# w_1 w_1 w_2 w_2 \dots w_n w_n \mid A, B, w_i \in (0 \mid 1)^+, |A| > |B|,$$

$$n = |A| - |B|, |w_i| \geq |B| \}$$

Sia  $\alpha \in \{0, 1\}$

Verifichiamo che  $|A| > |B|$  mentre calcoliamo  $n$ , per ogni  $i$  copiamo  $w$  sul nastro 4 assicurandoci che sia più lunga di  $B$ , poi indoviniamo quando siamo a metà e controlliamo.

Nastri: 1 = input, 2 =  $|A| - |B|$ , 3 =  $|B|$ , 4 =  $w_i$



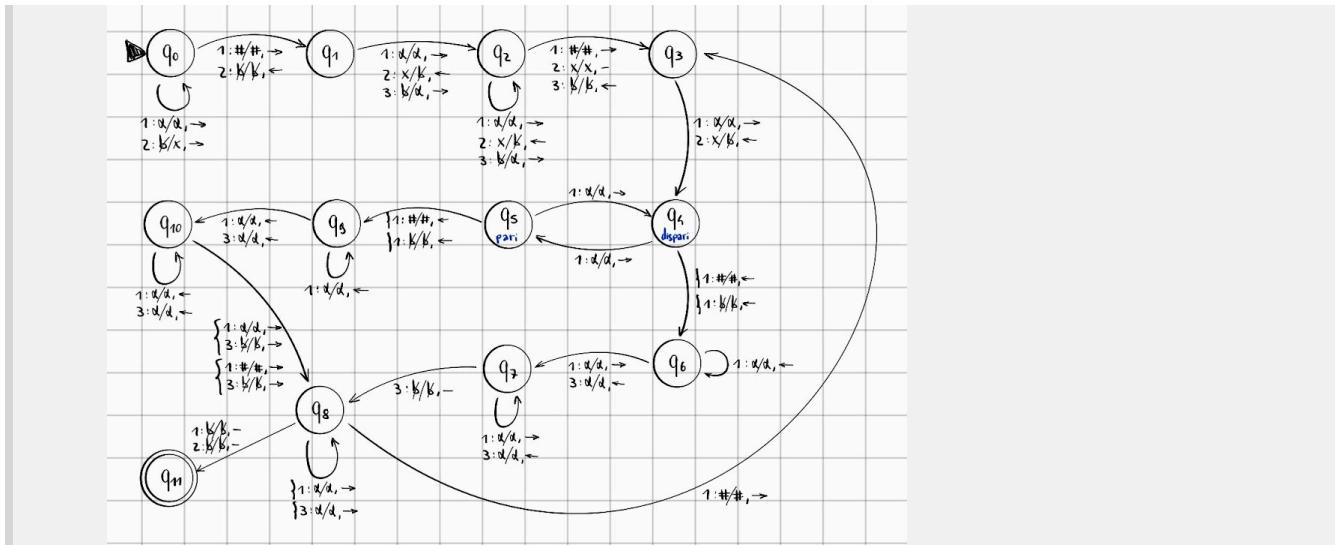
$$6. L = \{ A \# B \# W_1 \# \dots \# W_n \mid A, B, W_i \in (a \mid b \mid c \mid d)^+, |A| > |B|,$$

$$n = |A| - |B|, B \subseteq W_i \text{ se } |W_i| \text{ è pari}, B^R \subseteq W_i \text{ se } |W_i| \text{ è dispari} \}$$

Sia  $\alpha \in \{a, b, c, d\}$

Verifichiamo che  $|A| > |B|$  mentre calcoliamo  $n$  e copiamo  $B$ , per ogni  $i$  controlliamo se  $|W|$  è pari o dispari, poi cerchiamo  $B$  oppure  $B$  indovinando da che punto partire.

Nastri: 1 = input, 2 =  $|A| - |B|$ , 3 =  $B$



## Risultati di calcolabilità

Nonostante tutti i formalismi che abbiamo introdotto, non siamo riusciti ad allargare il "regime di calcolabilità" delle macchine di Turing standard.

## Tesi di Church-Turing

"Tutto ciò che è calcolabile, è calcolabile da una macchina di Turing."

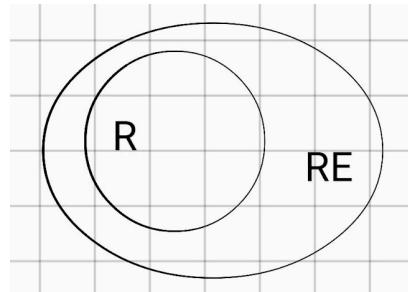
Viene chiamata "tesi" perché non definisce la calcolabilità di un problema a prescindere.

Nessuno, al momento, è riuscito a contraddirla, formulando un modello di calcolo più potente delle macchine di Turing, quindi si assume che questa tesi sia vera.

A questo punto possiamo introdurre due **classi di calcolabilità**:

- **R** → classe dei problemi (o linguaggi) Ricorsivi. Ne fanno parte tutti i linguaggi per cui esiste una MT che li decida. Sono anche detti problemi decidibili.
- **RE** → classe dei problemi (o linguaggi) Ricorsivamente Enumerabili. Contiene tutti i linguaggi per cui esiste una MT che li accetti. Sono anche detti problemi indecidibili (quelli  $p \in RE$  ma  $p \notin R$ ). A volte li definiamo semi-decidibili, poiché abbiamo una risposta in tempo finito solo in caso affermativo.

Esistono anche problemi fuori da RE, che non danno garanzia di risposta in nessun caso.



# Macchina di Turing universale

Diversamente dalle MT presentate fino a questo momento, la nostra nozione di computer permette la programmabilità: dato in input un programma, il computer lo esegue "adattando" la propria funzione di transizione.

Un computer è sostanzialmente una macchina di Turing universale, ovvero un tipo di macchina in grado di comportarsi come altre MT, nel momento in cui gli viene detto come farlo.

Andiamo a specializzare la definizione di MT per definire una **macchina di Turing universale**:

- l'alfabeto di input ha solo i simboli 0 e 1  $\implies \Sigma = \{0, 1\}$
- l'alfabeto di nastro è composto da 0, 1, blank e altre cose  $\implies \Gamma = \{0, 1, \emptyset, \dots\}$

## Codifica MT

Dobbiamo codificare la funzione di transizione delle MT (che sarebbe il programma), in modo che la macchina universale la possa prendere in input per replicarne il comportamento.

Sia  $\delta(q_i, x_j) = (q_k, x_l, D_m)$ .

Codifichiamo i simboli tramite un numero di zeri pari al pedice: la codifica di  $q_i$  sarà una sequenza di  $i$  zeri, per i simboli  $x_j$  avremo  $j$  zeri, mentre per la direzione  $D_m$  assumeremo che  $\leftarrow$  sia  $D_1$  e  $\rightarrow$  sia  $D_2$ , rispettivamente uno e due zeri.

Per codificare i simboli di  $\Gamma$  assumiamo che 0 sia  $x_1$ , 1 sia  $x_2$ ,  $\emptyset$  sia  $x_3$  e i restanti simboli siano da  $x_4$  a seguire. Assumiamo inoltre che  $q_1$  sia sempre lo stato iniziale e  $q_2$  sia sempre il singolo stato accettante.

Siccome usiamo solo gli zeri per codificare i simboli, inseriamo un 1 per separarli tra loro.

Esempio di codifica:

Sia  $\delta(q_3, \emptyset) = (q_5, 0, \rightarrow)$

Codifica:  $00010001000001 \underbrace{0}_{0} \underbrace{1}_{1} \underbrace{\rightarrow}_{\rightarrow}$

Quella nell'esempio è una sola entry della funzione di transizione, per concatenarne più di una usiamo un doppio uno (11).

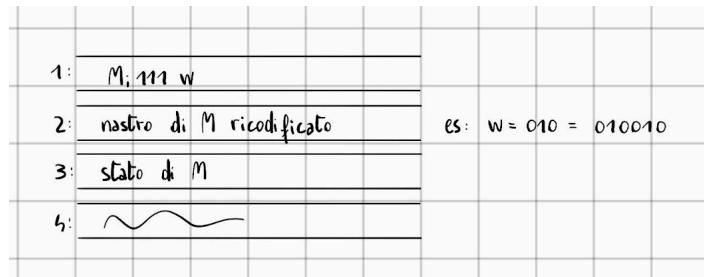
In questo modo possiamo codificare il programma di una MT in una stringa binaria. Se una stringa non rispetta la codifica, assumiamo che sia una MT con un solo stato, non accettante.

## Progettazione della macchina

Procediamo adesso con la progettazione della MT universale. La macchina  $M_u$  prende in input la codifica di una MT  $M_i$  e una stringa  $w$ , come output ci dice se la macchina  $M_i$  accetta la stringa  $w$ . Quindi la macchina universale simula  $M_i$  su  $w$  e restituisce il suo risultato.

Progettiamo una macchina avente 4 nastri:

- Il primo è il nastro di input, su cui abbiamo  $M_i$  e  $w$ , separati tra loro da un triplo uno (111). Questo nastro non viene mai modificato, in quanto il programma della macchina  $M_i$  deve essere preservato.
- Sul secondo nastro viene scritta la stringa  $w$ , ricodificata secondo i simboli introdotti precedentemente ( $x_1, \dots, x_n$ ).
- Il terzo nastro mantiene lo stato attuale della macchina  $M_i$ , in primis ci viene scritto lo stato iniziale.
- Il quarto nastro viene utilizzato come appoggio per le operazioni.



Tramite il secondo e terzo nastro,  $M_u$  conosce la configurazione in cui si trova la macchina  $M_i$ , quindi analizza il primo nastro per cercare l'entry della funzione di transizione corrispondente, la quale codifica l'operazione da effettuare. Fatto ciò, la macchina  $M_u$  modifica lo stato sul terzo nastro ed altera la stringa sul secondo.

La macchina  $M_u$  fa passo passo quello che farebbe la macchina  $M_i$ , compreso il risultato di accettazione o rifiuto (può anche andare in loop).

La funzione di transizione universale, ovvero il programma di  $M_u$ , non cambia, e ci permette di simulare il comportamento di altre MT. Possiamo quindi dire che la macchina  $M_u$  è programmabile.

I nostri computer sono MT universali, pertanto hanno una funzione di transizione universale, che sarebbe il ciclo di microistruzioni della CPU.

# Calcolabilità dei linguaggi

D'ora in poi ci focalizzeremo su MT aventi un alfabeto  $\Sigma = \{0, 1\}$  (è dimostrabile che sono equivalenti a tutte le altre).

Secondo la nostra codifica le MT iniziano per zero, aggiungendo un 1 in testa (per non perdere gli zeri a sinistra) otteniamo un numero binario, che identifica la specifica MT. Quindi tramite la codifica, le MT sono infinite e numerabili.

Il **vettore caratteristico** di un linguaggio  $L$ , denotato con  $x_L$ , è una sequenza binaria infinita che ci dice se la stringa  $i$ -esima fa parte del linguaggio o meno. Questo è possibile perché siamo in grado di enumerare le stringhe.

Iniziamo a mostrare l'esistenza di linguaggi non accettati da nessuna macchina di Turing, a cominciare dal linguaggio diagonale  $L_D$ , definito a seguire.

## Teorema

$$L_D \notin RE$$

## Dimostrazione

Costruiamo una matrice: sulle colonne mettiamo le stringhe (enumerabili), sulle righe le MT (anch'esse enumerabili).

Per ciascuna cella scriviamo 1 se la MT accetta la stringa, 0 altrimenti. Quindi le righe della matrice contengono il vettore caratteristico del linguaggio accettato dalla MT corrispondente.

Prendiamo allora la diagonale  $D = \{0 1 0 1 \dots\}$  e ne facciamo il complemento

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

	$w_1$	$w_2$	$w_3$	$w_4$	$\dots$
$M_1$	0	1	0	0	$\dots$
$M_2$	1	1	0	1	$\dots$
$M_3$	0	0	0	1	$\dots$
$M_4$	1	0	0	1	$\dots$
$\vdots$					

Possiamo affermare che  $\bar{D}$  è il vettore caratteristico di un certo linguaggio, chiamato  $L_D$ .

Nella matrice sono elencate tutte le possibili MT, nessuna esclusa. Per ogni MT  $M_i$ , il vettore caratteristico differisce da  $\bar{D}$  per la stringa  $i$ -esima, quindi nessuna MT può accettare il linguaggio  $L_D$ .

Non c'è una MT in grado di accettare  $L_D \implies L_D \notin RE$ .

$L_D = \{w_i \mid M_i \not\models w_i\}$ , ovvero l'insieme delle macchine di Turing (codificate) che ricevendo in input la propria codifica, non accettano.

I possibili linguaggi sono tanti quanti i numeri reali (insieme delle parti di  $\mathbb{N}$ ), mentre le macchine di Turing sono tante quante i numeri naturali. Questo spiega l'esistenza dei linguaggi non accettati da nessuna MT (sono molti più di quelli accettati).

## Proprietà dei linguaggi

Sia  $L \subseteq \Sigma^*$ .

Denotiamo con  $\overline{L}$  il complemento di  $L$ , ovvero l'insieme di tutte le stringhe che non appartengono a  $L$ .  $\overline{L} = \Sigma^* \setminus L$ .

Dimostriamo due proprietà che chiamano in causa  $L$  e  $\overline{L}$ :

### 1. Teorema

Se  $L \in R \implies \overline{L} \in R$

#### Dimostrazione

Se  $L \in R \implies \exists$  una MT  $M$  che decide  $L$ .

Quindi possiamo definire una macchina  $\overline{M}$ , che prende l'input, lo passa ad  $M$  e poi in output restituisce l'opposto di  $M$ .

Tale macchina rispetto ad  $M$  ha gli stati accettanti "invertiti" e solo le transizioni non presenti nella prima.

$\overline{M}$  decide  $\overline{L} \implies \overline{L} \in R$ .

### 2. Teorema

Se  $L \in RE \wedge \overline{L} \in RE \implies L \in R$

#### Dimostrazione

Se  $L \in RE \implies \exists$  una MT  $M$  che accetta  $L$ .

Se  $\overline{L} \in RE \implies \exists$  una MT  $\overline{M}$  che accetta  $\overline{L}$ .

Quindi per uno stesso input  $w$ :

-  $M$  risponde di sì in tempo finito se  $w \in L$ .

-  $\overline{M}$  risponde di sì in tempo finito se  $w \in \overline{L}$ , ovvero se  $w \notin L$ .

Abbiamo sempre una risposta in tempo finito per  $L \implies L$  è decidibile.

In generale, per dimostrare l'appartenenza di un linguaggio ad una classe dobbiamo costruire una MT che lo accetti (per  $RE$ ) o lo decida (per  $R$ ), mentre per dimostrare la non appartenenza dobbiamo scrivere una dimostrazione formale.

## Linguaggio universale

Il linguaggio universale  $L_u$  è l'insieme delle coppie  $(M, w)$  tali che la macchina  $M$  accetti la stringa  $w$ , ovvero  $L_u = \{(M, w) \mid M \models w\}$ .

Chiaramente  $L_u \in RE$ , poiché è il linguaggio accettato dalla macchina universale.

Ci chiediamo se  $L_u \in R$ . L'intuizione è che se diamo in input a  $M_u$  una MT  $M$  che non termina su  $w$ , anche  $M_u$  non si arresterà, quindi  $L_u \notin R$ . Questo però dimostra soltanto che  $M_u$  non decide  $L_u$ , non garantisce nulla sull'esistenza di una MT che lo faccia.

### Teorema

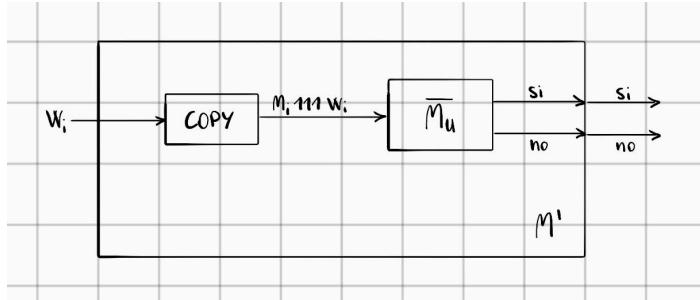
$$L_u \notin R$$

### Dimostrazione

Supponiamo per assurdo che  $L_u \in R$ .

Se  $L_u \in R \implies \overline{L_u} \in R \implies \exists$  una MT  $\overline{M_u}$  che decide  $\overline{L_u}$ .

Costruiamo una MT  $M'$  che, presa in input una stringa  $w_i$ , genera una stringa  $M_i 111 w_i$  replicando  $w_i$ , da dare in input alla macchina  $\overline{M_u}$ , il cui output sarà l'output di  $M'$ .



Esaminiamo il linguaggio riconosciuto da  $M'$ :

- se  $M'$  risponde sì, vuol dire che  $M_i \not\models w_i$
- se  $M'$  risponde no, vuol dire che  $M_i \models w_i$

Il linguaggio riconosciuto da  $M'$  è  $L_D \implies M'$  decide  $L_D$ .

Poiché  $L_D \notin RE$  non può esistere una MT in grado di deciderlo  $\implies L_u \notin R$ .

### Corollario

$$\overline{L_u} \notin RE$$

### Dimostrazione

Se  $\overline{L_u} \in RE \implies L_u \in R$ .

Ma questo è impossibile per il teorema soprastante.

## Complemento del linguaggio diagonale

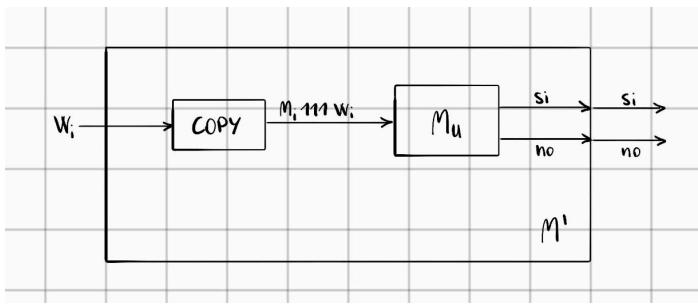
Sia  $\overline{L_D} = \{ w_i \mid M_i \models w_i \}$ .

### Teorema

$$\overline{L_D} \in RE$$

### Dimostrazione

Definiamo una MT  $M'$  che, presa in input una stringa  $w_i$ , genera una stringa  $M_i 111 w_i$  replicando  $w_i$ , da dare in input alla macchina universale, il cui output sarà l'output di  $M'$ .



Analizziamo il funzionamento di  $M'$ :

- se  $w_i \in \overline{L_D} \implies M' \text{ risponde sì}$
- se  $w_i \notin \overline{L_D} \implies M' \text{ risponde no oppure non risponde}$
- $\exists \text{ una MT che accetta } \overline{L_D} \implies \overline{L_D} \in RE.$

### Corollario

$$\overline{L_D} \notin R$$

### Dimostrazione

Se  $\overline{L_D} \in R \implies L_D \in R$ , il che è falso.

## Linguaggio dell'arresto

Definiamo il linguaggio dell'arresto come l'insieme delle coppie di stringhe  $(M, w)$  tali che  $M$  termini su  $w$ , ovvero  $\text{HALT} = \{ (M, w) \mid M \text{ si arresta su } w \}$ .

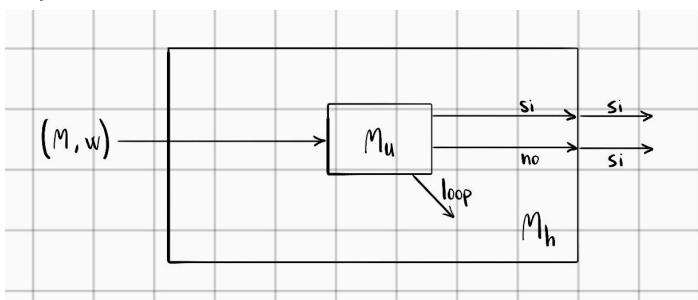
È una versione più lasca di  $L_u$ , in quanto non ci interessa accettare, ma solo verificare la terminazione.

### Teorema

$$\text{HALT} \in RE$$

### Dimostrazione

Costruiamo la macchina  $M_h$ , che presa in input una coppia  $(M, w)$ , la passa alla macchina universale. Se  $M_u$  termina in tempo finito  $M_h$  risponde sì, se  $M_u$  non si arresta  $M_h$  non dà risposta.



Analizziamo il comportamento di  $M_h$ :

- se  $(M, w) \in \text{HALT} \implies M_u$  si arresta  $\implies M_h$  risponde sì
- se  $(M, w) \notin \text{HALT} \implies M_u$  va in loop  $\implies M_h$  non risponde
- $\exists$  una MT che accetta HALT  $\implies \text{HALT} \in RE$ .

Come prima, questa dimostrazione non prova che non esista una MT in grado di decidere il linguaggio HALT.

### Teorema

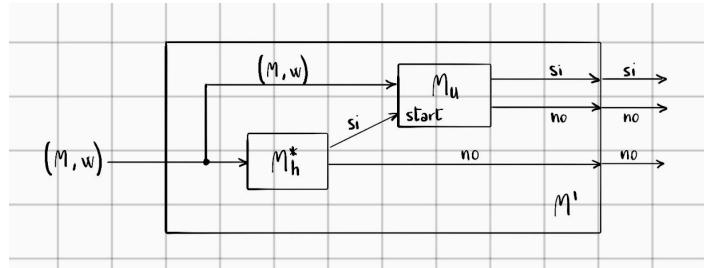
$$\text{HALT} \notin R$$

### Dimostrazione

Supponiamo per assurdo che  $\text{HALT} \in R$ .

Se  $\text{HALT} \in R \implies \exists$  una MT  $M_h^*$  che decide HALT.

Costruiamo una MT  $M'$  che riceve in input una coppia  $(M, w)$  e la trasmette in parallelo ad  $M_h^*$  e ad una macchina universale. Se  $M_h^*$  risponde no, anche  $M'$  risponde no. Se  $M_h^*$  risponde sì, facciamo partire  $M_u$ , il cui output sarà l'output di  $M'$ .



Esaminiamo il linguaggio riconosciuto da  $M'$ :

- se  $M'$  risponde sì, vuol dire che  $M \models w$
- se  $M'$  risponde no, vuol dire che  $M \not\models w \vee M$  non si arresta su  $w$

Il linguaggio riconosciuto da  $M'$  è  $L_u \implies M'$  decide  $L_u$ .

Poiché  $L_u \notin R$  non può esistere una MT in grado di deciderlo  $\implies \text{HALT} \notin R$ .

### Corollario

$$\overline{\text{HALT}} \notin RE$$

### Dimostrazione

Se  $\overline{\text{HALT}} \in RE \implies \text{HALT} \in R$ , il che è falso.

### Linguaggio dell'arresto su $\epsilon$

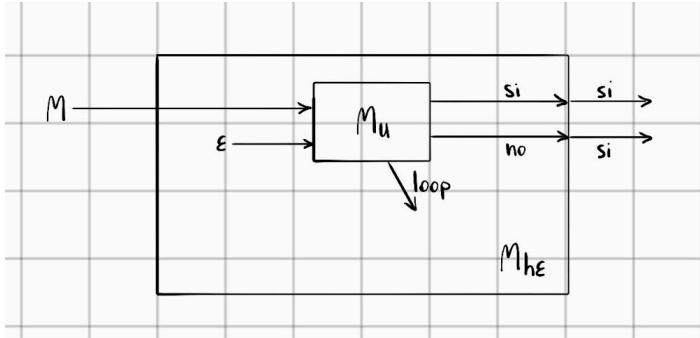
Vediamo una specializzazione del linguaggio dell'arresto, che considera solo le stringhe vuote:  $\text{HALT}_\epsilon = \{ M \mid M \text{ si arresta su } \epsilon \}$ .

## Teorema

$\text{HALT}_\epsilon \in RE$

### Dimostrazione

Costruiamo una MT  $M_{h\epsilon}$ , specializzando la macchina usata per provare che  $\text{HALT} \in RE$ . Anziché ricevere in input una coppia  $(M, w)$ , riceve solo  $M$  e come stringa passa direttamente  $\epsilon$ .



Analizziamo il funzionamento di  $M_{h\epsilon}$ :

- se  $M \in \text{HALT}_\epsilon \implies M_u$  si arresta  $\implies M_{h\epsilon}$  risponde sì
  - se  $M \notin \text{HALT}_\epsilon \implies M_u$  va in loop  $\implies M_{h\epsilon}$  non risponde
- $\exists$  una MT che accetta  $\text{HALT}_\epsilon \implies \text{HALT}_\epsilon \in RE$ .

## Teorema

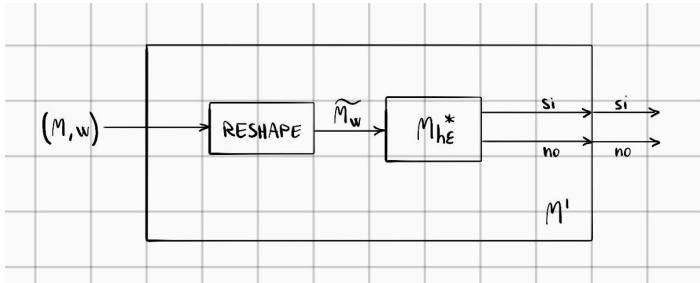
$\text{HALT}_\epsilon \notin R$

### Dimostrazione

Supponiamo per assurdo che  $\text{HALT}_\epsilon \in R$ .

Se  $\text{HALT}_\epsilon \in R \implies \exists$  una MT  $M_{h\epsilon}^*$  che decide  $\text{HALT}_\epsilon$ .

Costruiamo una MT  $M'$  che prende in input una coppia di stringhe  $(M, w)$  e la trasforma in una macchina  $\widetilde{M}_w$ . Tale macchina scrive  $w$  sul nastro (ignorando l'input), poi si comporta come  $M$ . Diamo  $\widetilde{M}_w$  in pasto ad  $M_{h\epsilon}^*$ , il cui output sarà l'output di  $M'$ .



Siccome  $\widetilde{M}_w$  ignora l'input  $\epsilon$  dato da  $M_{h\epsilon}^*$ , in realtà  $M'$  ci sta dicendo se  $M$  si arresta su  $w$ :

- se  $M'$  risponde sì, vuol dire che  $M$  si arresta su  $w$
- se  $M'$  risponde no, vuol dire che  $M$  non si arresta su  $w$

Il linguaggio riconosciuto da  $M'$  è  $\text{HALT} \implies M'$  decide  $\text{HALT}$ .

Poiché  $\text{HALT} \notin R$  non può esistere una MT in grado di deciderlo  $\implies \text{HALT}_\epsilon \notin R$ .

## Corollario

$\overline{\text{HALT}_\epsilon} \notin RE$

## Dimostrazione

Se  $\overline{\text{HALT}_\epsilon} \in RE \implies \text{HALT}_\epsilon \in R$ , il che è falso.

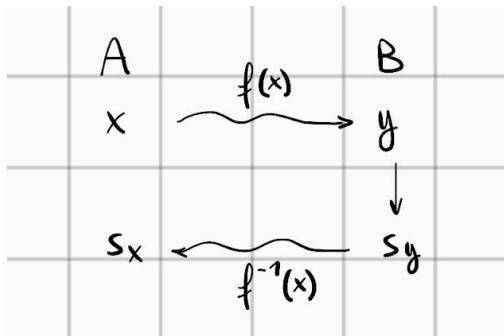
## Riduzioni

Introduciamo ora il concetto di riduzione tra linguaggi, per semplificare le dimostrazioni in cui dobbiamo provare che un linguaggio non appartiene ad una certa classe.

Intuitivamente, l'idea è risolvere istanze di un problema  $A$  sfruttando un algoritmo che risolve istanze di un problema  $B$ . Le riduzioni hanno una direzione: c'è un problema di partenza ed un problema di arrivo.

Vogliamo trasformare le istanze del problema di partenza in istanze del problema di arrivo, in modo da poter usare un solutore del problema di arrivo per ottenere una soluzione, che riutilizzeremo come soluzione del problema di partenza.

Questa trasformazione da problema di partenza a problema di arrivo è detta **riduzione**. Il fatto che ci sia una direzione è cruciale, in quanto genera due riduzioni differenti.



Dobbiamo quindi progettare una funzione  $f$  per trasformare le istanze di  $A$  in istanze di  $B$ , in modo tale che la soluzione di  $B$  possa essere riutilizzata per il problema  $A$ . La riduzione funziona solo se  $f$  è pensata in maniera sensata. La funzione  $f^{-1}$  con cui torniamo al problema di partenza non deve necessariamente essere l'inversa di  $f$ , tuttavia in qualche modo sono legate.

Vediamo un esempio di riduzione.

Dobbiamo risolvere il problema del calendario, sapendo risolvere il problema della  $k$ -colorabilità.

- Problema di partenza: ci sono 4 squadre (J, M, B, R), vogliamo generare un calendario all'italiana.
- Problema di arrivo: sappiamo colorare i nodi di un grafo, facendo sì che nodi adiacenti non abbiano lo stesso colore.

Generiamo tutte le possibili coppie di squadre, poi le partizioniamo in sottoinsiemi rappresentanti le giornate.

- Soluzione del problema di partenza: un gruppo di insiemi di "partite compatibili" (ogni squadra appare in una sola partita per insieme).

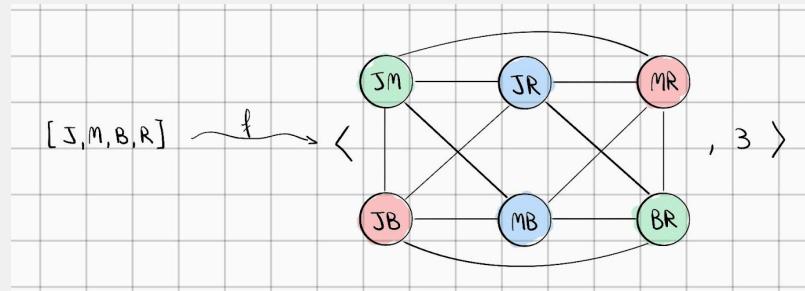
- Soluzione del problema di arrivo: un assegnamento di colori ai nodi tale che nodi adiacenti non abbiano lo stesso colore.

Da un lato le partite nella stessa giornata non devono collidere, dall'altro i nodi dello stesso colore non devono essere collegati.

L'input del problema di partenza è una lista di squadre  $L_s$ , l'input del problema di arrivo è una coppia  $\langle \text{grafo}, \text{numero-di-colori} \rangle$ . Dobbiamo inventarci una funzione che da una lista di squadre tiri fuori una coppia  $\langle g, k \rangle$ .

$f : L_s \rightarrow \langle g, k \rangle$  genera tanti nodi quante sono tutte le possibili partite, poi collega assieme i nodi che hanno una squadra in comune. Fatto ciò, risolviamo la  $k$ -colorabilità.

Dalla soluzione del problema di arrivo dobbiamo tornare alla soluzione di partenza: stabiliamo che ogni colore è una giornata.



Per progettare la funzione  $f$ , pensare a come sono fatte le istanze dei due problemi, nello specifico come sono strutturate le soluzioni (cercare delle similitudini). La funzione deve essere descritta in maniera esaustiva, non serve definire un algoritmo.

Per denotare che un problema si riduce ad un altro, usiamo il simbolo  $\leq$ .

calendar si riduce a  $k\text{-col}$ : calendar  $\leq k\text{-col}$ .

## Definizione

Siano  $A$  e  $B$  due linguaggi.

La funzione  $f$  è una **riduzione** da  $A$  a  $B$  se:

- per ogni stringa  $w$ ,  $w \in A \iff f(w) \in B$
- è calcolabile

Se  $w$  è istanza-sì per  $A$ , lo deve essere anche  $f(w)$  per  $B$ , e viceversa. Stessa cosa per le istanze-no. Siccome siamo interessati ai problemi di decisione, la soluzione di  $B$  sarà anche la soluzione di  $A$ , ovvero sì o no.

Definiamo formalmente quando una funzione è calcolabile, tramite il concetto di trasduttore.

Un **trasduttore** è una macchina di Turing caratterizzata da 3 nastri:

- il primo è il nastro di input, *read-only*
- il secondo è un work tape, dove fare le operazioni, *read-write*
- il terzo è il nastro di output, *write-only*

Diciamo quindi che 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. Lo chiamiamo anche calcolatore (la sua controparte è il decisore, ovvero la MT).

Una funzione  $f$  è calcolabile se esiste un trasduttore che la calcola in tempo finito.

La funzione  $f$  deve trasformare strutturalmente e basta, senza sapere se l'input è istanza-sì o istanza-no.

Quando effettuiamo la riduzione,  $f$  mappa tutte le istanze-sì del problema di partenza in un sottoinsieme (non necessariamente esaustivo) di istanze-sì del problema di arrivo. Stessa cosa per le istanze-no. Questo fatto non ci fa perdere generalità, in quanto è sufficiente anche una sola istanza non decidibile per rendere indecidibile l'intero problema.

## Proprietà delle riduzioni

Di seguito useremo la seguente trasformazione logica:  $x \implies y \equiv \neg x \vee y \equiv x \wedge \neg y$ .

Siano  $A$  e  $B$  due linguaggi tali che  $A \leq B$ .

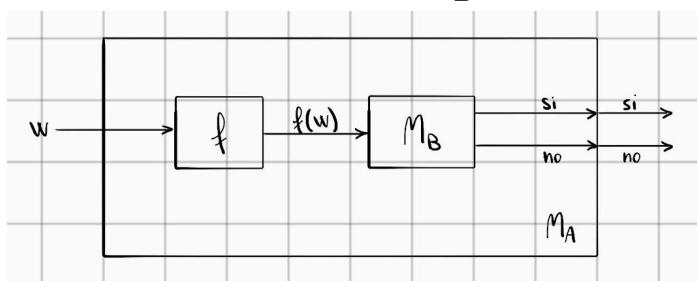
### 1. Teorema

Se  $A \notin R \implies B \notin R$

#### Dimostrazione

Supponiamo per assurdo che  $A \notin R \wedge B \in R$ .

Se  $B \in R \implies \exists$  una MT  $M_B$  che decide  $B$ .



Il linguaggio riconosciuto da  $M_A$  è  $A \implies M_A$  decide  $A$ .

Poiché  $A \notin R$  non può esistere una MT in grado di deciderlo  $\implies B \notin R$ .

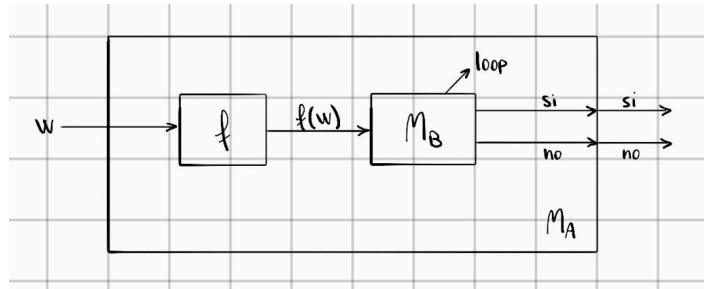
## 2. Teorema

Se  $A \notin RE \implies B \notin RE$

### Dimostrazione

Supponiamo per assurdo che  $A \notin RE \wedge B \in RE$ .

Se  $B \in RE \implies \exists$  una MT  $M_B$  che accetta  $B$ .



Il linguaggio riconosciuto da  $M_A$  è  $A \implies M_A$  accetta  $A$ .

Poiché  $A \notin RE$  non può esistere una MT in grado di accettarlo  $\implies B \notin RE$ .

### Corollario

Siano  $A$  e  $B$  due linguaggi tali che  $A \leq B$ .

1. Se  $B \in R \implies A \in R$

### Dimostrazione

Ottenuto per contronominale da (1)

2. Se  $B \in RE \implies A \in RE$

### Dimostrazione

Ottenuto per contronominale da (2)

## Linguaggio not-empty

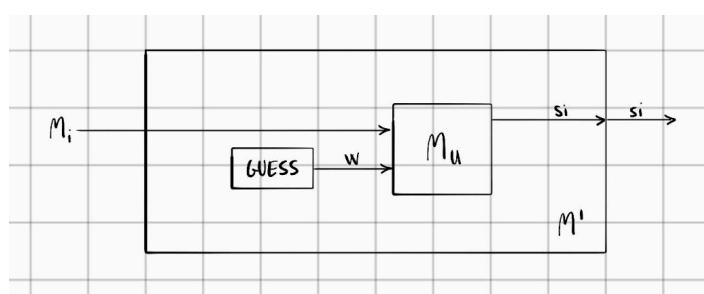
Identifichiamo con  $L_{ne}$  l'insieme dei codici di MT che accettano almeno una stringa, ovvero tali che il linguaggio della macchina  $M_i$  non è vuoto.

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

### Teorema

$$L_{ne} \in RE$$

### Dimostrazione



La macchina non deterministica  $M'$  indovina la stringa  $w$  accettata da  $M_i$ , scrivendola preventivamente sul nastro.

In caso di istanza-sì abbiamo risposta in tempo finito, in caso di istanza-no potremmo non ricevere risposta.

$\exists$  una MT in grado di accettare  $L_{ne} \implies L_{ne} \in RE$ .

Per stabilire se un problema è in  $R$  o meno, domandarsi se è difficoltoso identificare le istanze-no.

Intuitivamente, non possiamo affermare che una macchina non accetti nulla, in quanto dovremmo provare tutte le stringhe, che sono infinite.

Per dimostrare che  $L_{ne} \notin R$  tramite una riduzione,  $L_{ne}$  deve essere il problema di arrivo partendo da un problema indecidibile.

### Teorema

$$L_{ne} \notin R$$

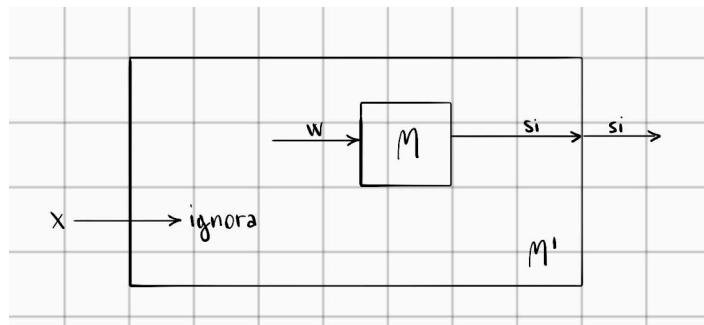
### Dimostrazione

Prendiamo come problema di partenza  $L_u$ .

	$L_u$	$\leq$	$L_{ne}$
<b>Input</b>	Una coppia macchina, stringa $(M, w)$	$\rightsquigarrow$	Una macchina $M'$
<b>Istanza-sì</b>	$M$ accetta $w$		$M'$ accetta almeno una stringa
<b>Istanza-no</b>	$M$ non accetta $w$		$M'$ non accetta nessuna stringa

La funzione  $f$  deve prendere in input una coppia  $(M, w)$  e restituire in output una macchina  $M'$ , tale da mantenere la relazione tra le istanze.

**Riduzione:** dalla coppia  $(M, w)$  generiamo una macchina  $M'$  che al suo interno esegue  $M$  su  $w$ . Se  $M$  risponde sì,  $M'$  risponde sì.



Verifichiamo che la riduzione sia corretta:

$$\Rightarrow) M \models w \implies \mathcal{L}(M') = \Sigma^* \neq \emptyset$$

$$\Leftarrow) M \not\models w \implies \mathcal{L}(M') = \emptyset$$

$$\text{Siccome } L_u \text{ è riducibile a } L_{ne} \implies L_{ne} \notin R$$

Procedimento: scrivere i due problemi a sinistra e destra, con i rispettivi input. Fatto ciò, progettare la riduzione pensando a come mantenere la relazione tra istanze-sì ed istanze-no.

## Linguaggio empty

Sia  $L_e = \overline{L_{ne}} = \{ M_i \mid \mathcal{L}(M_i) = \emptyset \}$ , ovvero il linguaggio delle MT che non accettano nessuna stringa.

### Teorema

$$L_e \notin RE$$

### Dimostrazione

Se  $L_e \in RE \implies L_{ne} \in R$ , il che è falso.

## Problema di corrispondenza di Post (PCP)

Si tratta di un problema su stringhe. In input abbiamo due liste della stessa lunghezza (finita), chiamate  $A$  e  $B$ , composte da stringhe definite sullo stesso alfabeto.

Ci chiediamo se esiste una sequenza di indici tale che concatenando gli elementi delle liste  $A$  e  $B$ , nella sequenza degli indici, arriviamo ad ottenere la stessa stringa. Gli indici non devono necessariamente essere contigui e/o ordinati.

Formalmente, ci chiediamo se  $\exists i_1, \dots, i_n$  con  $n > 0$  tale che  $r_{i1} \cdot \dots \cdot r_{in} = s_{i1} \cdot \dots \cdot s_{in}$ .

$PCP \in RE$ , in quanto una MT non deterministica lo accetta "indovinando" la sequenza di indici che fa al caso nostro.

Per dimostrare che  $PCP \notin R$  sfruttiamo la proprietà transitiva delle riduzioni, passando per un problema intermedio. Possiamo sempre ridurre un problema ad uno più complesso, ma non viceversa, quindi fare attenzione a non complicarsi la vita.

Inizialmente riduciamo  $L_u$  ad una versione modificata di PCP, che chiameremo MPCP.

Questa versione si differenzia dall'originale per il vincolo di avere come primo indice 1, ovvero prendere per prime le stringhe in testa alle due liste. Successivamente, ridurremo MPCP al problema PCP originale.

### Prima riduzione: $L_u \leq \text{MPCP}$

	$L_u$	$\leq$	MPCP
<b>Input</b>	Una coppia macchina, stringa $(M, w)$	$\rightsquigarrow$	Due liste di stringhe $A$ e $B$
<b>Istanza-sì</b>	$M$ risponde sì elaborando $w$		$\exists$ una sequenza di indici cominciante per 1 che porta a costruire la stessa stringa
<b>Istanza-no</b>	$M$ non risponde sì elaborando $w$		$\nexists$ una sequenza di indici cominciante per 1 che porta a costruire la stessa stringa

La funzione  $f$  deve essere calcolabile, quindi per trasformare  $(M, w)$  in due liste di stringhe  $A$  e  $B$  non può simulare  $M$  su  $w$ , in quanto la macchina potrebbe non arrestarsi.

Idea alla base della riduzione: vogliamo codificare i passi della macchina  $M$  su  $w$  tramite le stringhe delle liste  $A$  e  $B$ . Dobbiamo quindi inserire in  $A$  e  $B$  stringhe tali che se  $M \models w \implies$  otteniamo la corrispondenza, altrimenti no.

**Riduzione:** sappiamo che la computazione di  $M$  su  $w$  è rappresentabile come una stringa  $\# \alpha_1 \# \alpha_2 \# \dots$  tale che  $\alpha_1 \vdash_M \alpha_2 \vdash_M \dots$

Definiamo 5 classi di coppie di stringhe  $\langle r_i, s_i \rangle$  (pezzi di configurazioni  $\alpha$ ), che ci permetteranno di ricostruire il comportamento di  $M$  su  $w$ .

La codifica della computazione si potrà ottenere sia concatenando gli  $r_i$  che concatenando gli  $s_i$ . La stringa  $s$  sarà sempre un passo avanti rispetto ad  $r$ , la quale raggiungerà  $s$  soltanto se la macchina  $M$  arriva ad uno stato accettante. Dato che la stringa proveniente da  $B$  è sempre più avanti rispetto a quella proveniente da  $A$ , abbiamo costantemente un'indicazione sulla prossima coppia di stringhe da selezionare.

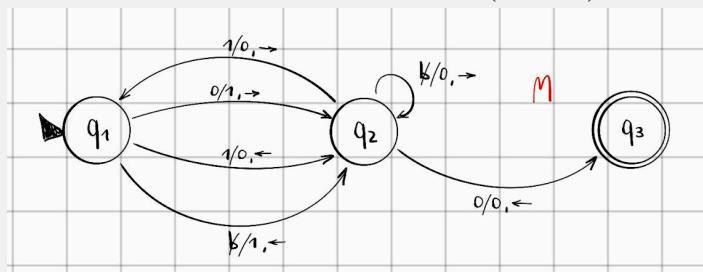
Assumiamo che  $M$  abbia un nastro semi-infinito, ovvero non sposti mai la testina prima della cella iniziale. Assumiamo inoltre che  $M$  non scriva mai  $\$$ .

Classe	$A$	$B$	Applicazione
1	#	$\# q_0 w_1 \dots w_n \#$	
2	$X$	$X$	$\forall X \in \Gamma$
	#	#	
3 ( $q \notin F$ )	$qX$	$Yp$	$\forall \text{ regola } \delta(q, X) = (p, Y, \rightarrow)$

Classe	A	B	Applicazione
	$ZqX$	$pZY$	$\forall \text{ regola } \delta(q, X) = (p, Y, \leftarrow)$
	$q\#$	$Yp\#$	$\forall \text{ regola } \delta(q, \not{p}) = (p, Y, \rightarrow)$
	$Zq\#$	$pZY\#$	$\forall \text{ regola } \delta(q, \not{p}) = (p, Y, \leftarrow)$
<b>4</b> ( $q \in F$ )	$XqY$	$q$	
	$Xq$	$q$	
	$qY$	$q$	
<b>5</b> ( $q \in F$ )	$q\#\#$	#	

### Esempio di trasformazione

Supponiamo di avere una coppia  $(M, w)$  da fornire in input alla funzione  $f$ .



Sia  $w = 01$ .

La computazione della macchina  $M$  su  $w$  è la seguente:

$$q_1 01 \vdash 1 q_2 1 \vdash 10 q_1 \vdash 1 q_2 01 \vdash q_3 101$$

Andiamo a produrre le liste  $A$  e  $B$ .

Classe	A	B
<b>1</b>	#	$\#q_1 01 \#$
<b>2</b>	0	0
	1	1
	$\not{p}$	$\not{p}$
	#	#

Classe	A	B
3	$q_1 0$	$1 q_2$
	$0 q_1 1$	$q_2 0 0$
	$1 q_1 1$	$q_2 1 0$
	$q_2 1$	$0 q_1$

Dovendo estendere la stringa  $A$  con i simboli già presenti sulla stringa  $B$ , progrediamo con entrambe le stringhe assieme, avendo sempre un'indicazione sulla coppia di stringhe da selezionare successivamente.

$$A \rightsquigarrow \# q_1 0 1 \# 1 q_2 1 \# \dots$$

$$B \rightsquigarrow \# q_1 0 1 \# 1 q_2 1 \# 1 0 q_1 \# \dots$$

Dobbiamo ora dimostrare la correttezza della riduzione:

$\Rightarrow)$   $M \models w \implies$  siamo in grado di concatenare pezzi di configurazione fino a costruire la stessa stringa di computazione sia per  $A$  che per  $B$ .

$\Leftarrow)$  se abbiamo raggiunto un'istanza-sì per MPCP  $\implies$  abbiamo costruito una stringa rappresentante una computazione accettante per la macchina  $M$  su  $w$ , quindi  $(M, w)$  era un'istanza-sì per  $L_u$ .

Siccome  $L_u$  è riducibile a MPCP  $\implies$  MPCP  $\notin R$ .

Dimostrando che da istanze-sì arriviamo ad istanze-sì, per equivalenza logica abbiamo anche dimostrato che partendo da istanze-no arriviamo ad istanze-no. Questo vale solo se dimostrato da ambo i lati.

### Seconda riduzione: MPCP $\leq$ PCP

	MPCP	$\leq$	PCP
<b>Input</b>	Due liste di stringhe $A$ e $B$	$\rightsquigarrow$	Due liste di stringhe $C$ e $D$
<b>Istanza-sì</b>	$\exists$ una sequenza di indici cominciante per $\boxed{1}$ che porta a costruire la stessa stringa		$\exists$ una sequenza di indici che porta a costruire la stessa stringa
<b>Istanza-no</b>	$\nexists$ una sequenza di indici cominciante per $\boxed{1}$ che porta a costruire la stessa stringa		$\nexists$ una sequenza di indici che porta a costruire la stessa stringa

**Riduzione:**

- trasformiamo le stringhe da  $A$  a  $C$  inserendo un \* dopo ogni simbolo.
  - trasformiamo le stringhe da  $B$  a  $D$  inserendo un \* prima di ogni simbolo.
- Abbiamo bisogno di due coppie aggiuntive per cominciare e terminare le concatenazioni:
- una con il primo elemento di  $C$  preceduto da \* ed il primo elemento di  $D$
  - una con \$ e \*\$

$A (r_i)$	$B (s_i)$	$\rightsquigarrow$	$C (x_i)$	$D (y_i)$
			*1*	*1 * 1 * 1
1	111		1*	*1 * 1 * 1
10111	10		1 * 0 * 1 * 1 * 1*	*1 * 0
10	0		1 * 0*	*0
			\$	*\$

Verifichiamo quindi la correttezza della riduzione:

$\Rightarrow$ ) supponiamo di partire da una coppia  $(A, B)$  tale che  $\exists$  una sequenza di indici cominciante per 1 tramite cui costruiamo la stessa stringa.

Ovvero,  $\exists 1, i_2 \dots i_m$  tale che  $r_1 \cdot r_{i_2} \cdot \dots \cdot r_{i_m} = s_1 \cdot s_{i_2} \cdot \dots \cdot s_{i_m}$ .

Consideriamo ora l'uguaglianza  $x_1 \cdot x_{i_2} \cdot \dots \cdot x_{i_m} = y_1 \cdot y_{i_2} \cdot \dots \cdot y_{i_m}$ .

Essa non è verificata in quanto la stringa  $x$  finisce per \*, mentre la stringa  $y$  inizia per \*. Ecco che entrano in gioco le due coppie aggiuntive: le concateniamo rispettivamente all'inizio e alla fine di entrambe le stringhe per verificare l'uguaglianza.

$\Leftarrow$ ) supponiamo di aver raggiunto un'istanza-sì di PCP, esiste quindi una sequenza  $i_1 \dots i_m$  tale che costruiamo la stessa stringa. Necessariamente il primo indice della sequenza è il numero 1, essendo l'unica coppia che comincia con lo stesso simbolo. Da ciò si può dedurre che siamo partiti da un'istanza-sì di MPCP.

Siccome MPCP è riducibile a PCP  $\implies$  PCP  $\notin R$ .

## Problema del tassellamento (Tiling)

Supponiamo di avere una superficie da ricoprire con delle piastrelle. Tale superficie è suddivisa in riquadri di dimensione 1 unità, dentro ai quali verranno posizionate le piastrelle. Analogamente al primo quadrante del piano cartesiano, la superficie ha un riquadro  $(0,0)$  di origine e si espande all'infinito sia per vie verticali che orizzontali.

Le piastrelle, che sono tutte di dimensione 1 unità, sono divise diagonalmente in 4 settori, ognuno di un colore (non ci sono vincoli sui colori di una singola piastrella). Ogni piastrella è caratterizzata dai colori e dalla loro disposizione.

Per tassellare, le piastrelle possono essere accostate orizzontalmente o verticalmente secondo alcune regole: possono essere affiancate se e solo se i due lati adiacenti hanno lo stesso colore, non possono essere ruotate. Esiste una piastrella specifica che sarà l'origine. Dato un sistema di questo tipo, ci chiediamo se è possibile ricoprire l'intero quadrante.

Un'istanza del problema è una quadrupla  $T = \langle D, d_0, H, V \rangle$ , dove:

- $D$  è l'insieme finito dei tipi di piastrella
- $d_0 \in D$  è il tipo della piastrella di origine
- $H \subseteq D \times D$  è l'insieme delle regole di affiancamento orizzontale
- $V \subseteq D \times D$  è l'insieme delle regole di affiancamento verticale

Un tiling attraverso il sistema  $T$  è un'allocazione che mappa la posizione di un riquadro al tipo della piastrella, rispettando i vincoli orizzontali e verticali. Definiamo quindi un tiling come una funzione  $f : \mathbb{N} \times \mathbb{N} \rightarrow D$ , tale che:

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

Per dimostrare che il problema del Tiling è indecidibile eseguiremo una riduzione a partire da  $\overline{\text{HALT}}_\epsilon$ .

	$\overline{\text{HALT}}_\epsilon$	$\leq$	TILING
<b>Input</b>	Una macchina $M$	$\rightsquigarrow$	Un sistema di tiling $T$
<b>Istanza-sì</b>	$M$ non si arresta processando $\epsilon$		È possibile tassellare la superficie attraverso $T$
<b>Istanza-no</b>	$M$ si arresta processando $\epsilon$		Non è possibile tassellare la superficie attraverso $T$

Assumiamo che  $M$  abbia un nastro semi-infinito (non limita l'espressività).

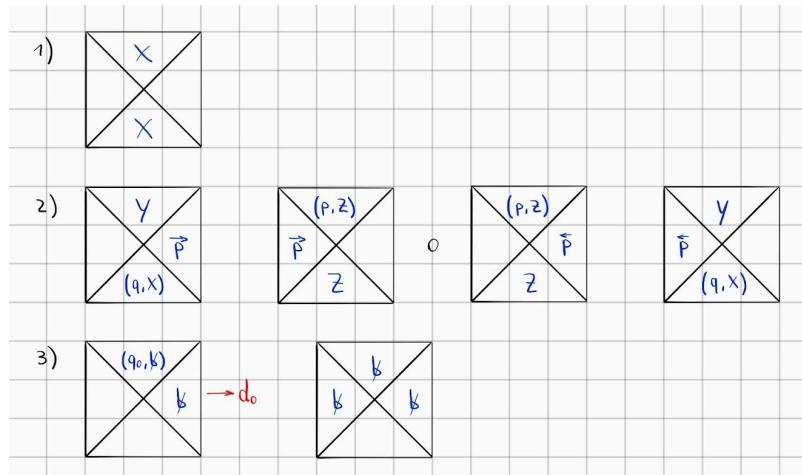
L'idea è codificare la computazione della macchina  $M$  su  $\epsilon$  tramite il tiling: descriveremo orizzontalmente le configurazioni, le quali evolveranno verso l'alto. In questo modo, se la macchina non si arresta potremo piastrellare all'infinito.

Fare riferimento ai colori è complicato, quindi usiamo delle etichette.

Vogliamo codificare le configurazioni sui bordi che uniscono un livello orizzontale ad un altro. I bordi verticali, all'interno di una stessa fila, serviranno per forzare una certa disposizione delle piastrelle.

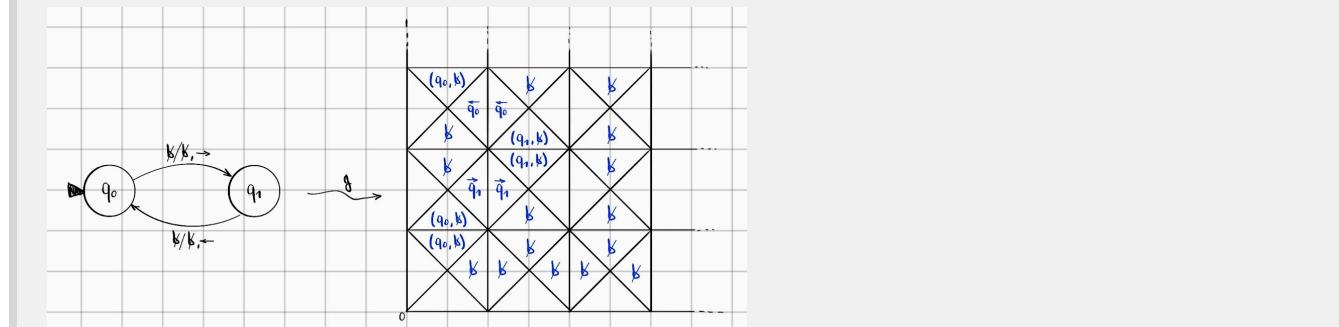
La funzione di trasformazione  $g$  prende in input una macchina  $M$  e restituisce un sistema di tiling  $T$  avente un insieme di tipi di piastrella così composto:

- 1)  $\forall x \in \Gamma \rightarrow$  lascia invariato il simbolo se la testina è distante
- 2)  $\forall q, p \in Q \setminus F$  e  $\forall X, Y, Z \in \Gamma \rightarrow$  codifica una generica transizione  $qXZ \rightsquigarrow YpZ$  data da  $\delta(q, X) = (p, Y, \rightarrow)$  oppure  $ZqX \rightsquigarrow pZY$  data da  $\delta(q, X) = (p, Y, \leftarrow)$
- 3) assicura che al primo livello ci sia la configurazione iniziale



### Esempio di trasformazione

Supponiamo di avere la seguente macchina che non si arresta su  $\epsilon$ .



Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che la macchina  $M$  non si arresti su  $\epsilon$ .

Per come abbiamo definito i tipi di piastrella, possiamo aggiungere una nuova riga in cima ogni volta che la macchina  $M$  effettua un passo.

Siccome  $M$  non si arresta, effettuerà infiniti passi.

Questo vuol dire che potremo aggiungere infinite righe  $\Rightarrow$  è possibile tassellare l'intera superficie.

$\Leftarrow$ ) Supponiamo di essere arrivati ad un sistema di tiling  $T$  che ci consente di tassellare l'intera superficie.

Secondo la trasformazione,  $T$  descrive la computazione della macchina  $M$  su  $\epsilon$ .

$T$  non definisce tipi di piastrella per gli stati terminali.

Perciò la computazione non è mai passata per uno stato finale  $\Rightarrow M$  non si arresta su  $\epsilon$ .

Siccome  $\overline{\text{HALT}_\epsilon}$  è riducibile a TILING  $\Rightarrow \text{TILING} \notin R$ .

## Classe CO-RE

Tutti i linguaggi che abbiamo visto fino ad ora hanno una peculiarità: un legame con il loro complemento. I linguaggi che stanno in  $RE$  sono quelli per cui possiamo rispondere sì in tempo finito. Il complemento di tali linguaggi si trova fuori da  $RE$ . Essendo complementi, per questi linguaggi possiamo rispondere no in tempo finito.

Definiamo una classe di decidibilità ulteriore che racchiude tutti quei linguaggi per i quali siamo in grado di rispondere no in tempo finito, ovvero i complementi dei linguaggi in  $RE$ , e la chiamiamo *CO-RE*.

Intuitivamente, TILING  $\in CO-RE$  poiché i tipi di piastrella sono in numero finito, quindi possiamo generare un albero con tutte le possibili combinazioni che sarà anch'esso finito.

Fatto ciò, lo esploriamo interamente e se non troviamo un tassellamento completo rispondiamo no.

Per quanto detto prima,  $\overline{\text{TILING}} \in RE$ .

Ci chiediamo se esistono linguaggi esterni sia ad  $RE$  che a  $CO-RE$ , per i quali non siamo in grado di rispondere nè sì nè no in tempo finito. Per arrivarci, dobbiamo prima comprendere le relazioni tra le classi.

### Teorema

$$R \neq RE$$

### Dimostrazione

$$\exists L \mid L \in RE \wedge L \notin R \implies R \neq RE$$

**Teorema**

$$CO\text{-}RE \cap RE = R$$

**Dimostrazione**

$\subseteq$ ) Sia  $L$  un linguaggio tale che  $L \in CO\text{-}RE \cap RE$ .

Siccome  $L \in CO\text{-}RE \implies \bar{L} \in RE$ .

$L \in RE \wedge \bar{L} \in RE \implies L \in R$ .

$\supseteq$ ) Sia  $L$  un linguaggio tale che  $L \in R$ .

Se  $L \in R \implies \bar{L} \in R$ .

Quindi  $\exists$  una MT che garantisce risposta sia per il sì che per il no.

Pertanto  $L$  sta sia in  $RE$  che in  $CO\text{-}RE$ .

**Teorema**

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

**Dimostrazione**

Supponiamo per assurdo che  $CO\text{-}RE = RE$ .

Poiché  $CO\text{-}RE \cap RE = R \implies RE = R$ , il che è falso per il teorema precedente.

**Teorema**

Siano  $A$  e  $B$  due linguaggi tali che  $A \leq B$ .

Se  $A \notin CO\text{-}RE \implies B \notin CO\text{-}RE$

**Dimostrazione**

Il procedimento è lo stesso del caso  $RE$ .

**Linguaggio dell'arresto universale**

Definiamo  $\text{HALT}_\forall$  come l'insieme delle MT che si arrestano su tutte le stringhe, ovvero

$$\text{HALT}_\forall = \{ M \mid \forall w, M \text{ si arresta su } w \}.$$

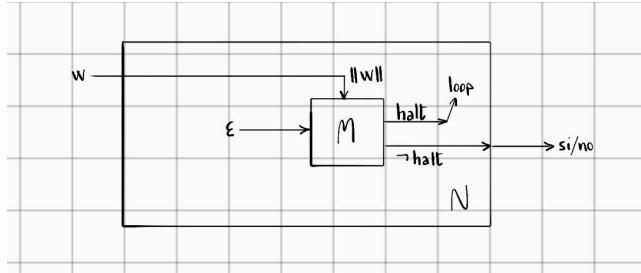
Abbiamo bisogno di due riduzioni: una per dimostrare che  $\text{HALT}_\forall \notin RE$  e una per dimostrare che  $\text{HALT}_\forall \notin CO\text{-}RE$ .

**Prima riduzione**

Per dimostrare che  $\text{HALT}_\forall \notin RE$ , effettuiamo una riduzione a partire da  $\overline{\text{HALT}_\epsilon}$ .

	$\overline{\text{HALT}_\epsilon}$	$\leq$	$\text{HALT}_\forall$
<b>Input</b>	Una macchina $M$	$\rightsquigarrow$	Una macchina $N$
<b>Istanza-sì</b>	$M$ non si arresta processando $\epsilon$		$N$ si arresta $\forall w$
<b>Istanza-no</b>	$M$ si arresta processando $\epsilon$		$N$ non si arresta $\forall w$

La funzione  $f$  restituisce una macchina  $N$  che prende in input  $w$  e si comporta come segue: al suo interno esegue la simulazione di  $M$  su  $\epsilon$  per un numero massimo di step pari alla lunghezza di  $w$ , ossia  $\|w\|$ . Se durante questa simulazione limitata  $M$  si arresta  $\Rightarrow N$  va in loop, se  $M$  non si arresta  $\Rightarrow N$  si ferma e risponde (a caso).



Verifichiamo che la riduzione sia corretta:

$\Rightarrow$ ) Supponiamo che  $M$  non si arresti su  $\epsilon$ .

Questo vuol dire che qualunque sia il limite di passi,  $M$  non si arresterà.

Quindi per ogni input  $w$ , la macchina  $N$  si arresta e risponde.

$\Leftarrow$ ) Supponiamo di partire da un'istanza-no di  $\overline{\text{HALT}}_\epsilon$ .

Questo significa che  $M$  si arresta su  $\epsilon$ , diciamo in  $t$  passi.

Quindi per tutti gli input  $w$  tali che  $\|w\| \geq t$  la macchina  $N$  entra in loop, poiché al suo interno la macchina  $M$  fa in tempo ad arrestarsi su  $\epsilon$ .

$\exists w$  tali che  $N$  non si arresta  $\Rightarrow N$  è un'istanza-no di  $\text{HALT}_\forall$ .

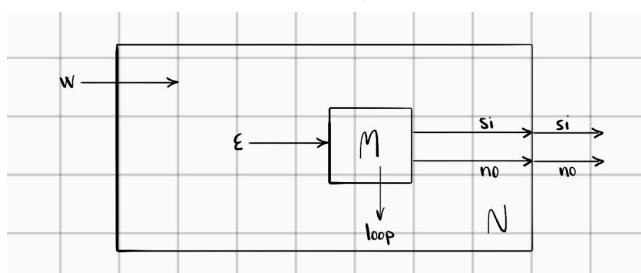
Siccome  $\overline{\text{HALT}}_\epsilon$  è riducibile ad  $\text{HALT}_\forall$   $\Rightarrow \text{HALT}_\forall \notin RE$ .

## Seconda riduzione

Per dimostrare che  $\text{HALT}_\forall \notin CO-RE$ , eseguiamo una riduzione a partire da  $\text{HALT}_\epsilon$ .

	$\text{HALT}_\epsilon$	$\leq$	$\text{HALT}_\forall$
<b>Input</b>	Una macchina $M$	$\rightsquigarrow$	Una macchina $N$
<b>Istanza-sì</b>	$M$ si arresta processando $\epsilon$		$N$ si arresta $\forall w$
<b>Istanza-no</b>	$M$ non si arresta processando $\epsilon$		$N$ non si arresta $\forall w$

La funzione  $f$  restituisce una macchina  $N$  che ignora il proprio input e al suo interno esegue la simulazione di  $M$  su  $\epsilon$ , il cui risultato sarà il risultato di  $N$ .



Verifichiamo la correttezza di questa riduzione:

$\Rightarrow$ ) Supponiamo che  $M$  si arresti su  $\epsilon$ .

Per costruzione, la macchina  $N$  si arresta su qualunque input  $w$ .

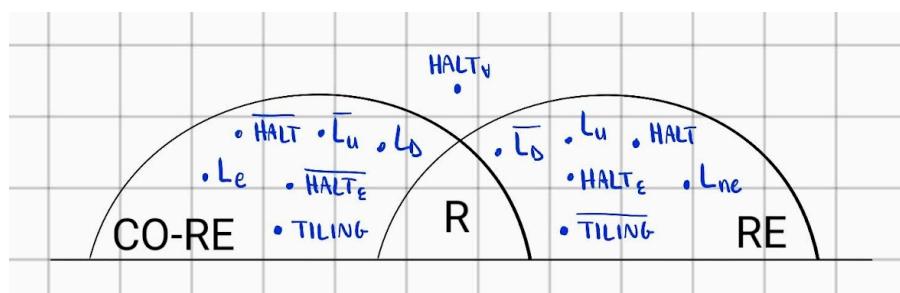
$\Leftarrow$ ) Supponiamo di essere arrivati ad una macchina  $N$  che si arresta su ogni input  $w$ .

Questo vuol dire che la macchina  $M$  al suo interno si arresta su  $\epsilon$ .

Siccome  $\text{HALT}_\epsilon$  è riducibile ad  $\text{HALT}_\forall \implies \text{HALT}_\forall \notin \text{CO-RE}$ .

Da queste due riduzioni segue che  $\text{HALT}_\forall$  è fuori sia da  $\text{RE}$  che da  $\text{CO-RE}$ .

## Classi di decidibilità



## Proprietà di macchine di Turing

I linguaggi  $L_e$  e  $L_{ne}$ , che abbiamo dimostrato essere indecidibili, hanno la peculiarità di contenere codifiche di MT il cui linguaggio possiede una certa proprietà, ovvero essere vuoto o meno.

Una **proprietà**  $P$  di macchine di Turing è un insieme di codifiche di MT che soddisfano determinati vincoli. Una proprietà può fare riferimento a *come è fatta la macchina* oppure al *linguaggio della macchina*.

Il linguaggio della proprietà  $P$  è definito come  $L_P = \{ M \mid M \in P \}$ .

Una proprietà  $P$  è **banale** se contiene tutte le macchine oppure non ne contiene nessuna  $\rightarrow$  si verifica sempre in tempo costante.

Esempi di proprietà di MT:

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

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

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

Le proprietà che riguardano solamente il linguaggio riconosciuto dalla macchina, come  $P_2$  e  $P_e$ , le chiamiamo semantiche.

Una proprietà  $P$  di macchine di Turing si dice **semantica** se:

$$\forall M_1, M_2 \text{ tali che } \mathcal{L}(M_1) = \mathcal{L}(M_2), M_1 \in P \implies M_2 \in P.$$

L'appartenenza o meno di una MT ad una proprietà semantica è caratterizzata soltanto dal linguaggio che essa riconosce. Per questa ragione, le proprietà semantiche le chiamiamo anche proprietà di linguaggi.

La connessione è data dal fatto che possiamo descrivere un linguaggio in maniera compatta tramite la MT che lo riconosce, perciò sia le proprietà semantiche che quelle di linguaggi contengono codici di MT.

Una proprietà  $P$  di linguaggi  $RE$  è un sottoinsieme di  $RE$ , quindi  $P \subseteq RE$  è un insieme di linguaggi.

Una proprietà  $P$  di linguaggi  $RE$  è banale se  $P = \emptyset$  oppure  $P = RE$ .

## Teorema di Rice

### Teorema

Ogni proprietà non banale di linguaggi  $RE$  è indecidibile.

### Dimostrazione

Sia  $P$  una proprietà non banale di linguaggi  $RE$ .

Ci sono due casi possibili:

**1)** Il linguaggio vuoto  $\emptyset \notin P$

Se  $P$  è una proprietà non banale di  $RE \implies P$  non può essere vuota.

Siccome  $\emptyset \notin P$ , deve esistere un linguaggio  $L \in RE \neq \emptyset$  che appartiene a  $P$ .

Se  $L \in RE \implies \exists$  una MT  $M_L$  che accetta il linguaggio  $L$ .

Mostriamo l'indecidibilità del linguaggio  $L_P$  tramite una riduzione da  $L_u$ .

	$L_u$	$\leq$	$L_P$
<b>Input</b>	Una coppia macchina, stringa $(M, w)$	$\rightsquigarrow$	Una macchina $N$
<b>Istanza-sì</b>	$M$ accetta $w$		$\mathcal{L}(N)$ ha la proprietà $P$
<b>Istanza-no</b>	$M$ non accetta $w$		$\mathcal{L}(N)$ non ha la proprietà $P$

Costruzione della macchina  $N$ : per prima cosa,  $M$  esegue su  $w$ . Se  $M$  accetta, viene eseguita  $M_L$  sull'input  $x$  di  $N$ , il cui output sarà l'output di  $N$ .

Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) supponiamo che  $(M, w) \in L_u$ , ovvero  $M \models w$ .

Se  $M \models w \implies N$  accetta il linguaggio  $L$ .

Avendo supposto che  $L \in P \implies N \in L_P$ .

$\Leftarrow$ ) supponiamo che  $(M, w) \notin L_u$ , ovvero  $M \not\models w$ .

Se  $M \not\models w \implies$  la seconda parte della computazione non partirà mai, quindi  $N$  non accetta nulla, ovvero  $\mathcal{L}(N) = \emptyset$ .

Poiché  $\emptyset \notin P \implies N \notin L_P$ .

Siccome  $L_u$  è riducibile a  $L_P \implies L_P$  è indecidibile.

## 2) Il linguaggio vuoto $\emptyset \in P$

Consideriamo  $\bar{P}$ , definito come  $RE \setminus P$ .

Abbiamo che  $\emptyset \notin \bar{P}$ .

Se  $P$  è una proprietà non banale  $\implies$  anche  $\bar{P}$  è non banale.

Quindi  $\bar{P}$  è una proprietà non banale che non contiene il linguaggio vuoto.

Per il caso (1) che abbiamo dimostrato sopra,  $L_{\bar{P}}$  è indecidibile.

Consideriamo ora il linguaggio  $\overline{L_{\bar{P}}}$ , che equivale a  $L_P$  (poiché siamo dentro a  $RE$ ).

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

Se  $\overline{L_{\bar{P}}}$  fosse decidibile, lo sarebbe anche  $L_{\bar{P}}$ , il quale abbiamo dimostrato essere indecidibile.

Per questo motivo  $L_P$  è indecidibile.

Molto utile: se il linguaggio che stiamo considerando è una proprietà di linguaggi non banale, allora è indecidibile. Se la proprietà non è semantica, il teorema non ci dice nulla.

## Applicazioni del teorema

Esempio con i linguaggi  $L_e$  e  $L_{ne}$ :

- $L_e = \{ M \mid \mathcal{L}(M) = \emptyset \}$  definisce una proprietà di linguaggi. È non banale, poiché non contiene i linguaggi non vuoti e  $\emptyset \in L_e$ . Per il teorema di Rice è indecidibile.
- $L_{ne} = \{ M \mid \mathcal{L}(M) \neq \emptyset \}$  definisce una proprietà semantica di macchine. È non banale, poiché  $L_{ne} = \overline{L_e}$ , che è non banale. Per il teorema di Rice è indecidibile.

Consideriamo ora il linguaggio relativo alla proprietà  $P_2$  dell'esempio precedente:

- è una proprietà di macchine, in quanto contiene codifiche di MT
  - è semantica, poiché fa riferimento solo al linguaggio riconosciuto
  - è non banale, in quanto contiene i linguaggi di stringhe pari, ma non quelli di stringhe dispari
- Per il teorema di Rice,  $P_2$  è una proprietà indecidibile.

$P_1$  è una proprietà di macchine, ma non è semantica, quindi non si può applicare il teorema di Rice. Per dimostrare che è non semantica, dobbiamo fornire un controesempio  $\rightarrow$  due MT che riconoscono lo stesso linguaggio, una con 5 stati e una con un numero differente.

Alcune applicazioni del teorema di Rice:

- Data una MT  $M$ , decidere se  $\mathcal{L}(M)$  è finito.

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

È una proprietà di macchine ed è semantica.  $L_a$  contiene il linguaggio vuoto, non contiene il linguaggio di tutte le stringhe, quindi è non banale. Per il teorema è indecidibile.

- Data una MT  $M$ , decidere se  $\mathcal{L}(M)$  è infinito.

$$L_b = \{ M \mid \mathcal{L}(M) \text{ è infinito} \}$$

È una proprietà di macchine ed è semantica.  $L_b$  non contiene il linguaggio vuoto, ma contiene il linguaggio di tutte le stringhe, quindi è non banale. Per il teorema è indecidibile.

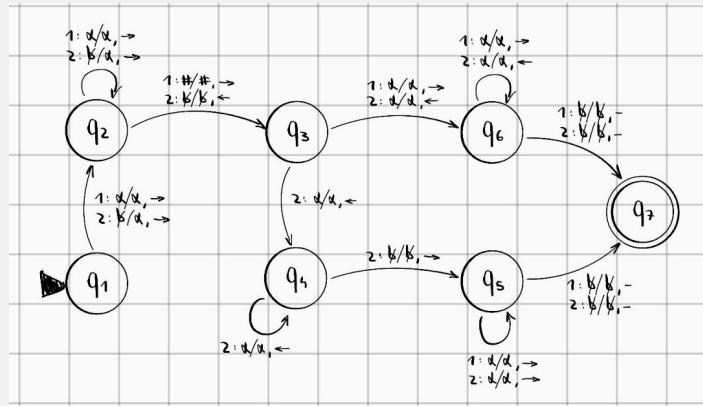
- Data una MT  $M$ , decidere se  $\mathcal{L}(M)$  è un linguaggio accettato solo da macchine con 5 stati.

$$L_c = \{ M \mid \mathcal{L}(M) \text{ è accettato solo da macchine con 5 stati} \}$$

È una proprietà di macchine ed è semantica. È banale, perché un linguaggio riconosciuto da una MT con 5 stati, può essere riconosciuto anche da una macchina con 6 (aggiungendo uno stato inutile), quindi  $L_c$  non contiene nulla.

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

Nastri: 1 = input, 2 =  $W$



$P_1 = \{ M \mid \mathcal{L}(M) = L \}$  è una proprietà di linguaggi non banale, poiché contiene  $L$ , ma non contiene  $\bar{L}$ . Per il teorema è indecidibile.

$P_2 = \{ M \mid \mathcal{L}(M) = L \wedge \forall w \in L, M \text{ la accetta in al più 100 passi} \}$  è una proprietà di macchine, banale perché possiamo scrivere stringhe di lunghezza  $> 100$ .

$P_3 = \{ M \mid M \not\models L \cap \{ 0, 1, \# \}^{100} \}$  (insieme delle MT che non accettano il linguaggio) è una proprietà di macchine, non banale. È una proprietà semantica, quindi per il teorema è indecidibile.

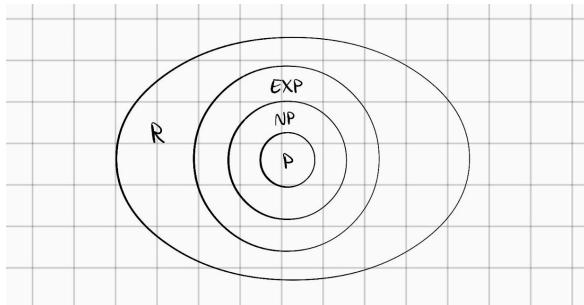
# Complessità strutturale dei problemi

## Introduzione

Da questo momento in poi ci focalizzeremo sui problemi ricorsivi, ossia decidibili, allo scopo di definirne la complessità strutturale. Lavorando dentro  $R$ , avremo sempre garanzia di ricevere una risposta.

Lo studio della complessità nacque da una necessità tecnologica quando apparirono le prime macchine fisiche, poiché all'epoca le risorse computazionali erano limitate (prima non era di interesse). Vennero formulati i concetti di **complessità temporale e spaziale**: il primo è legato al numero di passi eseguiti dalla MT prima di arrestarsi, ovvero il numero di transizioni, il secondo esprime la quantità di celle necessarie per dare risposta (sovrascrivendole in maniera opportuna possiamo ottimizzare l'esecuzione).

Studiare la complessità strutturale significa analizzare come la struttura di un problema influenzi la complessità degli algoritmi che lo risolvono, identificando classi di problemi di complessità simile. Sapendo dove si annida la complessità in un problema, è possibile definire dei vincoli per ottenere delle euristiche più efficienti.



*Quando parliamo di complessità di macchine, linguaggi o algoritmi, intendiamo lo stesso concetto.*

# Complessità temporale

## Definizioni

Sia  $M$  una MT,  $w$  una stringa in input per  $M$ .

Il **computation time** (tempo di esecuzione o computazione) di  $M$  su  $w$  è il numero di passi che  $M$  esegue prima di arrestarsi su  $w$ .

Se la macchina  $M$  è non deterministica, il computation time è dato dalla lunghezza del ramo del computation tree più lungo.

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

La definiamo così, in quanto ci aspettiamo che la macchina impieghi un tempo di esecuzione  $> 0$ , e che esso incrementi all'aumentare della lunghezza dell'input.

Sia  $t(n)$  una time function.

La MT  $M$  ha **running time**  $t(n)$  se per tutte le stringhe  $w$ , eccetto un numero finito, il computation time di  $M$  su  $w$  non eccede  $t(\|w\|)$ .

È un upper-bound funzionale al tempo di esecuzione della macchina.

## Notazione asintotica

### Big-O (o-grande)

Una funzione  $f(n)$  è  $O(g(n))$  se  $\exists c, n_0$  tali che  $\forall n \geq n_0$ ,  $f(n) \leq c \cdot g(n)$ .

Con  $f(n) \in O(g(n))$  intendiamo che  $f$  è o-grande di  $g$ , ovvero che  $f(n)$  non cresce più velocemente di  $g(n) \rightarrow$  upper-bound.

### Big- $\Omega$ (omega-grande)

Una funzione  $f(n)$  è  $\Omega(g(n))$  se  $\exists c, n_0$  tali che  $\forall n \geq n_0$ ,  $f(n) \geq c \cdot g(n)$ .

Con  $f(n) \in \Omega(g(n))$  intendiamo che  $f$  è omega-grande di  $g$ , ovvero che  $f(n)$  cresce più velocemente di  $g(n) \rightarrow$  lower-bound.

### Big- $\Theta$ (theta-grande)

$f(n)$  è  $\Theta(g(n))$  se  $f(n)$  è sia  $O(g(n))$  che  $\Omega(g(n))$ .

$f(n) \in \Theta(g(n))$  significa che  $f$  e  $g$  hanno lo stesso tasso di crescita.

# Complessità di problemi

Il **time complexity upper-bound** di un problema  $P$  è  $O(f(n))$  se esiste almeno un algoritmo che lo risolve avendo complessità  $O(f(n))$ , ovvero se esiste almeno un algoritmo che ci mette al più  $f(n)$ .

Il **time complexity lower-bound** di un problema  $P$  è  $\Omega(f(n))$  se tutti gli algoritmi che lo risolvono hanno complessità  $\Omega(f(n))$ , ovvero se tutti gli algoritmi ci mettono almeno  $f(n)$ .

La complessità temporale di un problema è pertanto legata alla complessità temporale degli algoritmi che lo risolvono.

## Time complexity dell'ordinamento di array

Upper-bound:

- $O(n^2) \rightarrow$  vero, Selection Sort
- $O(n \cdot \log n) \rightarrow$  vero, Merge Sort
- $O(n) \rightarrow$  falso

Lower-bound:

- $\Omega(n) \rightarrow$  vero, tutti ci mettono di più
- $\Omega(n \cdot \log n) \rightarrow$  vero, tutti ci mettono almeno quello
- $\Omega(n^2) \rightarrow$  falso

Diciamo che un problema è **trattabile** (o facile) se la sua complessità è polinomiale. Se il problema non ammette algoritmi polinomiali diciamo che è **difficile**.

## Classi di complessità temporale

Andiamo a definire all'interno di  $R$  delle classi di complessità, ovvero raffiniamo la classe di decidibilità per identificare cosa si risolve facilmente e cosa difficilmente.

Sia  $t(n)$  una time function.

Definiamo la classe **DTIME** come l'insieme di tutti i linguaggi decisi da una MT deterministica in tempo  $O(t(n))$ .

$$\text{DTIME}(t(n)) = \{ L \mid \exists M \text{ deterministica che decide } L \text{ in tempo } O(t(n)) \}$$

$P$  è la classe dei linguaggi che possono essere decisi da una MT deterministica in tempo polinomiale, con un certo esponente fissato.

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

Attenzione: DTIME contiene solo problemi di decisione. *DTIME non contiene ordinamento di array, somma di due numeri, ecc.*

## Esempi di problemi in $P$

- Reachability: dato un grafo diretto, un nodo sorgente e un nodo destinazione, stabilire se esiste un percorso che connette la sorgente alla destinazione.

$\text{REACH} = \{ \langle g, s, t \rangle \mid$

$g$  è un grafo diretto,  $s, t$  sono nodi in  $g$ ,  $\exists$  un percorso da  $s$  a  $t$  in  $g\}$

Con l'algoritmo di Dijkstra abbiamo un tempo  $O(n^2)$ .

- Primes: dato un numero intero in binario, stabilire se è primo.

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

Esiste un algoritmo ma è estremamente lento, tipo  $O(n^6)$ .

Alcuni problemi che non si trovano in  $P$ :

- SAT (soddisfacibilità di formule booleane): data una formula booleana in CNF, stabilire se esiste un assegnamento di verità per le variabili che la rende vera.

Una formula  $\varphi$  in CNF (forma normale congiuntiva) è una congiunzione di un numero finito di clausole  $c_1 \wedge \dots \wedge c_n$ , dove ogni clausola è una disgiunzione di uno o più letterali  $l_1 \vee \dots \vee l_m$ , i quali sono variabili booleane  $x_k$  oppure la loro negazione  $\neg x_k$ .

$\text{SAT} = \{ \varphi \mid \varphi \text{ è in CNF} \wedge \varphi \text{ è soddisfacibile}\}$

- IS (Independent Set): dato un grafo, stabilire se esiste un IS di taglia  $k$  al suo interno.

Un independent set di un grafo è un insieme di nodi che non sono collegati direttamente tra loro. Un nodo da solo è sempre un IS, perciò la formulazione del problema richiede un IS di dimensione  $k$ .

$\text{IS} = \{ \langle g, k \rangle \mid \exists \text{ un IS di taglia } k \text{ in } g\}$

Questi problemi, SAT e IS, tramite MT deterministiche sono decidibili in tempo esponenziale. Se usassimo MT non deterministiche, potremmo "indovinare" la soluzione e poi verificarla, in tempo polinomiale.

Esempio di risoluzione non deterministica:

- SAT: per ogni variabile, indoviniamo se assegnare true o false

- IS: per ogni nodo, indoviniamo se tenerlo o meno nell'IS

Sia  $t(n)$  una time function.

Definiamo la classe **NTIME** come l'insieme di tutti i linguaggi decisi da una MT non deterministica in tempo  $O(t(n))$ .

$\text{NTIME}(t(n)) = \{ L \mid \exists M \text{ non deterministica che decide } L \text{ in tempo } O(t(n))\}$

**NP** è la classe dei linguaggi che possono essere decisi da una MT non deterministica in tempo polinomiale, con un certo esponente fissato.

$$NP = \bigcup_{c \geq 1} \text{NTIME}(n^c)$$

Attenzione:  $NP$  non sta per non-polynomial, significa non-deterministic polynomial time.

## Teoria della complessità temporale

Siccome tutto ciò che può essere deciso in tempo polinomiale da una MT deterministica, può essere deciso in tempo polinomiale anche da una MT non deterministica, abbiamo  $P \subseteq NP$ . Tuttavia non possiamo dire niente riguardo il verso opposto della relazione,  $NP \subseteq P$ , da cui deriverebbe che  $P = NP$ .

A differenza di  $R$  ed  $RE$ , per cui abbiamo la certezza che siano distinti,  $P = NP$  è un problema aperto dell'informatica teorica. La nostra assunzione è che  $P$  e  $NP$  siano distinti, perché nessuno ha mai trovato un algoritmo per risolvere un problema  $NP$  in tempo polinomiale, tuttavia nessuno ha mai dimostrato formalmente che sia impossibile.

## Riduzioni polinomiali

Specializziamo il concetto di riduzione, definendo la riduzione polinomiale, che aggiunge il vincolo per la funzione  $f$  di essere polinomiale nella taglia dell'input.

Siano  $A$  e  $B$  due linguaggi.

Una **riduzione polinomiale** da  $A$  a  $B$  è una funzione  $f : \Sigma^* \rightarrow \Sigma^*$  tale che:

- per ogni stringa  $w$ ,  $w \in A \iff f(w) \in B$
- $f$  è calcolabile in tempo polinomiale

Denotiamo con  $A \leq_P B$  una riduzione polinomiale da  $A$  a  $B$ .

## NP-hardness e NP-completeness

### NP-hardness

Un linguaggio  $L$  è NP-arduo se  $\forall L' \in NP, L' \leq_P L$ , ovvero se tutti i linguaggi di  $NP$  sono riducibili polinomialmente a  $L$ .

Intuitivamente, questo vuol dire che un linguaggio NP-arduo è difficile almeno quanto tutti i linguaggi di  $NP$ .

### NP-completeness

Un linguaggio  $L$  è NP-completo se:

- $L \in NP$
- $L$  è NP-arduo

Un linguaggio NP-completo è un linguaggio NP-arduo che appartiene a  $NP$ , quindi è tra i più difficili di tale classe. I linguaggi fuori da  $R$  sono sicuramente non NP-completi.

Abbiamo visto che all'interno della classe  $NP$  ci sono problemi più difficili di altri. Se anche solo uno di essi fosse risolvibile in tempo polinomiale, avremmo il collasso della classe  $NP$  su  $P$ .

### Teorema

Sia  $L$  un linguaggio NP-completo.

$$L \in P \iff P = NP$$

### Dimostrazione

$\Leftarrow$ ) Assumiamo che  $P = NP$ .

Se  $L$  è un linguaggio NP-completo  $\implies L \in NP$ .

Siccome stiamo assumendo che  $P = NP$ , allora  $L$  sta anche in  $P$ .

$\Rightarrow$ ) Assumiamo che  $L \in P$ .

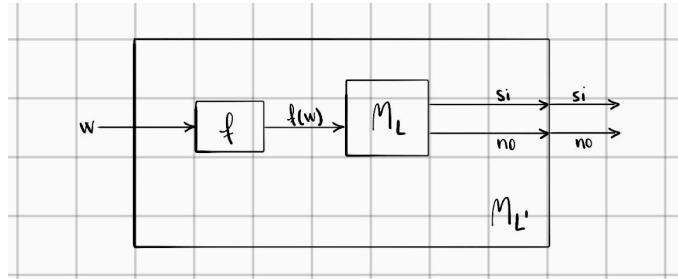
Sappiamo che  $P \subseteq NP$ , quindi dobbiamo dimostrare che  $NP \subseteq P$ .

Siccome  $L$  è un linguaggio NP-completo, allora è anche NP-arduo.

Questo vuol dire che  $\forall L' \in NP, L' \leq_P L$ .

Perciò  $\exists$  una funzione polinomiale  $f$  che mappa istanze di  $L'$  in istanze di  $L$ .

Consideriamo una macchina  $M_{L'}$ , in cui  $w$  viene trasformata in  $f(w)$ , che viene data in input alla macchina  $M_L$ , il cui output sarà l'output finale. Tale macchina sta decidendo il linguaggio  $L'$ .



Ci chiediamo se la macchina  $M_{L'}$  esegua in tempo polinomiale.

1. Inizialmente  $w$  viene trasformata in  $f(w)$ .

Essendo  $f$  polinomiale  $\implies f \in O(n^c)$ , dove  $n = \|w\|, c \geq 1$  fissato.

$\|f(w)\|$  è a sua volta  $O(n^c)$ , poiché  $f$  non ha il tempo di scrivere in output una stringa più lunga.

2. Poiché stiamo assumendo  $L \in P$ , il running time di  $M_L$  è  $O(n^d)$ .

Quindi passando in input  $f(w)$ , il costo totale dell'esecuzione di  $M_L$  è  $O((n^c)^d) = O(n^{c \cdot d})$ .

In conclusione, il running time totale di  $M_{L'}$  è  $O(n^c + n^{c \cdot d})$ , ovvero polinomiale.

Avendo una MT che decide un generico  $L' \in NP$  in tempo polinomiale, abbiamo dimostrato che  $NP \subseteq P$ , da cui  $P = NP$ .

Questo risultato afferma che, se esistesse un linguaggio NP-completo decidibile in tempo polinomiale deterministico, saremmo in grado di risolvere tutti i problemi difficili in tempo polinomiale deterministico.

Non è semplice verificare la NP-completezza, poiché è complicato controllare se un linguaggio sia NP-arduo o meno. Ciò è stato fatto per un linguaggio in particolare: SAT. Esiste una dimostrazione formale del fatto che SAT sia NP-arduo ed NP-completo.

Avendo un linguaggio NP-arduo, possiamo dimostrare che altri linguaggi sono NP-ardui tramite riduzioni. L'intuizione è che se riduciamo  $A$  a  $B$ , allora  $B$  è almeno difficile quanto  $A$ .

Tutto ciò funziona grazie alla **transitività delle riduzioni polinomiali**.

### Teorema

Siano  $A$ ,  $B$  e  $C$  tre linguaggi.

$$\text{Se } A \leq_P B \wedge B \leq_P C \implies A \leq_P C$$

### Dimostrazione

Se  $A \leq_P B \implies \exists$  una funzione  $f$  che trasforma in tempo  $O(n^c)$ , con  $c \geq 1$  fissato.

Se  $B \leq_P C \implies \exists$  una funzione  $g$  che trasforma in tempo  $O(n^d)$ , con  $d \geq 1$  fissato.

Dimostriamo quindi l'esistenza di una riduzione polinomiale da  $A$  a  $C$  definita come  $f \bullet g$ , ovvero  $g(f(w))$ .

Sappiamo che la composizione è fattibile eseguendo due trasformazioni consecutive.

Dobbiamo mostrare che  $g(f(w))$  sia calcolabile in tempo polinomiale.

Siccome le funzioni  $f$  e  $g$  calcolano in tempo polinomiale, l'output da esse restituito non può essere di un ordine di grandezza superiore, poiché non ci sarebbe tempo per scriverlo.

-  $f(w)$  è calcolato in  $O(n^c) \implies \|f(w)\|$  è  $O(n^c)$

-  $g(z)$  è calcolato in  $O(n^d) \implies \|g(z)\|$  è  $O(n^d)$

Quindi componendo le due funzioni,  $g(f(w))$  calcola in tempo  $O((n^c)^d) = O(n^{c \cdot d})$ , ovvero ancora polinomiale.

### Teorema

Sia  $A$  un linguaggio NP-arduo.

Se  $B$  è un linguaggio tale che  $A \leq_P B \implies B$  è NP-arduo.

### Dimostrazione

Secondo la definizione,  $B$  è NP-arduo se tutti i linguaggi di  $NP$  sono riducibili ad esso.

Sappiamo che  $A$  è un linguaggio NP-arduo per ipotesi, perciò ogni linguaggio  $L' \in NP$  è riducibile polinomialmente ad  $A$ .

Sempre per ipotesi, sappiamo che  $A$  si riduce polinomialmente a  $B$ .

Quindi, per la transitività delle riduzioni polinomiali, tutti i linguaggi  $L' \in NP$  sono riducibili polinomialmente a  $B$ .

Sfrutteremo questo risultato per dimostrare che determinati problemi sono NP-ardui.

# NP-completezza dei linguaggi

Per stabilire se un linguaggio è NP-completo dobbiamo verificare due condizioni:

- appartenenza a  $NP$
- NP-hardness

## 3SAT

3SAT è una specializzazione di SAT, in cui una clausola può contenere al massimo 3 letterali.

Data una formula booleana in 3CNF, stabilire se è soddisfacibile, ovvero se esiste un assegnamento di verità per le variabili che la rende vera.

$3SAT \in NP$  poiché tramite una MT non deterministica possiamo indovinare l'assegnamento e poi verificarlo, in tempo polinomiale.

Per dimostrare che sia NP-arduo, riduciamo SAT a 3SAT.

	SAT	$\leq_P$	3SAT
<b>Input</b>	Una formula booleana $\varphi$ in CNF	$\rightsquigarrow$	Una formula booleana $\psi$ in 3CNF
<b>Istanza-sì</b>	$\varphi$ è soddisfacibile		$\psi$ è soddisfacibile
<b>Istanza-no</b>	$\varphi$ non è soddisfacibile		$\psi$ non è soddisfacibile

Consideriamo una formula generica  $\varphi = c_1 \wedge \dots \wedge c_m$ .

Effettuiamo iterativamente una trasformazione sulle clausole  $c_i = l_1 \vee \dots \vee l_k$ , per arrivare ad una formula  $\varphi'$ :

- se la clausola  $c_i$  contiene al più 3 letterali, la ricopiamo così com'è.
- se la clausola  $c_i$  contiene più di 3 letterali, a partire da essa generiamo due clausole:  $c'_i = (l_1 \vee l_2 \vee h_i)$  e  $c''_i = (l_3 \vee \dots \vee l_k \vee \neg h_i)$ , dove  $h_i$  è una nuova variabile introdotta appositamente. In questo modo, la seconda clausola ha un letterale in meno rispetto a prima. Ripetiamo questo processo finché ci sono clausole con più di 3 letterali. Raggiunta una formula contenente solo clausole con al più 3 letterali, abbiamo ottenuto la nostra  $\psi$ .

Dimostriamo che questa trasformazione è polinomiale.

Supponiamo che la formula di partenza abbia  $m$  clausole. Dopo ogni trasformazione, nel caso peggiore ci troviamo  $2m$  clausole, di cui solo la metà possono avere più di 3 letterali. Quindi ad ogni iterazione il numero di clausole aumenta al massimo di una quantità  $m \implies$  polinomiale.

Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $\varphi$  sia soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento di verità  $\sigma$  che rende vera la formula  $\varphi$ .

Definiamo un assegnamento  $\tau_\sigma$ , basato su  $\sigma$ , tale che  $\psi$  sia soddisfatta.

I letterali presenti in  $\sigma$  mantengono lo stesso valore, dobbiamo definire il valore delle variabili  $h_i$ .

Siccome  $\sigma$  soddisfa  $\varphi$ ,  $\exists$  un letterale  $l_j$  con valore true in ciascuna clausola  $c_i$ . Se nella trasformazione, tale  $l_j$  finisce in  $c'_i$  poniamo  $h_i$  a false, altrimenti se finisce in  $c''_i$  poniamo  $h_i$  a true.

In questo modo entrambe le clausole generate da  $c_i$  saranno soddisfatte  $\Rightarrow$  tutte le clausole di  $\psi$  saranno soddisfatte  $\Rightarrow \psi$  è soddisfacibile.

$\Leftarrow$ ) Supponiamo di essere arrivati ad una formula  $\psi$  soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento di verità  $\tau$  che rende vere tutte le clausole di  $\psi$ .

Queste clausole possono essere di due tipi:

- ricopiate esattamente come erano in  $\varphi$ , quindi  $\exists$  un letterale  $l_j$  in  $\tau$  che era true anche nella formula  $\varphi$ .

- prese in coppia formano una clausola di  $\varphi$ . Siccome  $h_i$  rende true una sola delle due clausole di  $\psi$ , vuol dire che nell'altra clausola è presente un letterale true appartenente anche alla clausola di  $\varphi$  originaria.

Perciò stavamo partendo da una formula  $\varphi$  che a sua volta era soddisfacibile.

Siccome SAT è riducibile polinomialmente a 3SAT  $\Rightarrow$  3SAT è NP-arduo.

$3SAT \in NP \wedge 3SAT$  è NP-arduo  $\Rightarrow$  3SAT è NP-completo.

## Independent Set (IS)

Dato un grafo, stabilire se esiste un IS di taglia  $k$  al suo interno.

Un independent set è un insieme di nodi che non sono collegati direttamente tra loro. Un nodo da solo è sempre un IS, perciò la formulazione del problema richiede un IS con almeno  $k$  elementi.

$$IS = \{ \langle G, k \rangle \mid \exists \text{ un IS di taglia } k \text{ in } G \}$$

$IS \in NP$  poiché tramite una MT non deterministica possiamo indovinare l'insieme di  $k$  nodi che compone l'IS. La fase di guess impiega tempo lineare, quella di check impiega al massimo tempo quadratico, perciò la macchina esegue in tempo polinomiale.

Per dimostrare che IS è NP-arduo, riduciamo 3SAT a IS.

	<b>3SAT</b>	$\leq_P$	<b>IS</b>
<b>Input</b>	Una formula booleana $\phi$ in 3CNF	$\rightsquigarrow$	Una coppia grafo, numero di nodi $(G, k)$
<b>Istanza-sì</b>	$\phi$ è soddisfacibile		$\exists$ un IS di taglia $k$ in $G$
<b>Istanza-no</b>	$\phi$ non è soddisfacibile		$\nexists$ un IS di taglia $k$ in $G$

Idea della riduzione: assegnare true ad una determinata variabile significa inserire un determinato nodo nell'IS.

Per ogni letterale di ciascuna clausola, creiamo un nodo nel grafo.

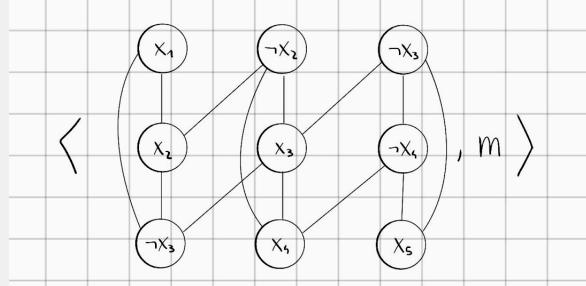
Per evitare che nell'IS finiscano variabili opposte ( $x_i$  e  $\neg x_i$ ), colleghiamo tra di loro i relativi nodi con un arco.

Con questa trasformazione, non abbiamo la certezza che, avendo un IS, la formula  $\phi$  sia soddisfatta, in quanto non ci sono vincoli sul da dove vengano presi i letterali. Per far sì che nell'IS ci finisca un letterale di ogni clausola, colleghiamo tra loro i nodi relativi alla stessa clausola di  $\phi$ .

La taglia  $k$  dell'IS sarà quindi pari al numero di clausole in  $\phi$ .

Esempio di trasformazione

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$$



Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $\phi$  sia soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento  $\sigma$  che rende true almeno un letterale per ogni clausola.

Sia  $S_\sigma$  l'insieme dei nodi che compongono l'IS.

Per ciascuna clausola di  $\phi$ , prendiamo un letterale vero secondo  $\sigma$  ed inseriamo il relativo nodo all'interno di  $S_\sigma$ .

Sia  $m$  il numero di clausole di  $\phi$ . Dentro  $S_\sigma$  ci saranno tanti nodi quante sono le clausole, ovvero  $|S_\sigma| = m$ , in particolare un nodo per clausola.

Supponiamo per assurdo che  $S_\sigma$  non sia un IS.

Quindi dentro  $S_\sigma$  ci sono due nodi collegati tra loro.

Siccome abbiamo preso un solo nodo per clausola, tali due nodi sono collegati perché uno è il negato dell'altro.

Ma questa è una contraddizione in quanto  $\sigma$  è un assegnamento coerente  $\implies S_\sigma$  è un IS.

Da cui  $S_\sigma$  è un IS di taglia  $k = m$ .

$\Leftarrow$ ) Supponiamo di essere arrivati ad un grafo contenente un IS  $S$  di taglia  $k = m$ .

Quindi  $S$  contiene un solo nodo per ciascuna clausola.

Sia  $\sigma_S$  un assegnamento di verità ottenuto dai nodi presenti in  $S$ .

Abbiamo la certezza che  $\sigma_S$  sia consistente, per costruzione di  $S$ .

Supponiamo per assurdo che  $\sigma_S$  non soddisfi  $\phi$ .

Quindi c'è una clausola in cui tutti i letterali sono false.

Ma questo è impossibile poiché vorrebbe dire che di tale clausola non abbiamo preso nessun letterale nella costruzione di  $S$ .

Da cui la formula  $\phi$  è soddisfatta dall'assegnamento  $\sigma_S$ .

Siccome 3SAT è riducibile polinomialmente a IS  $\implies$  IS è NP-arduo.

IS  $\in NP \wedge$  IS è NP-arduo  $\implies$  IS è NP-completo.

## Vertex Cover (VC)

Dato un grafo, stabilire se al suo interno esiste un VC di taglia al più  $k$ .

Un vertex cover è un insieme di nodi che tocca tutti gli archi del grafo, ovvero ogni arco ha un estremo nel VC. L'insieme di tutti i nodi è sempre un VC, perciò la formulazione del problema richiede un VC con massimo  $k$  elementi.

$$VC = \{ < G, k > \mid \exists \text{ un VC di taglia al più } k \text{ in } G \}$$

VC  $\in NP$  poiché tramite una MT non deterministica possiamo indovinare l'insieme con al massimo  $k$  nodi che compone il VC e verificare che sia corretto, in tempo polinomiale.

Per dimostrare che sia NP-arduo, riduciamo IS a VC.

	IS	$\leq_P$	VC
Input	Una coppia grafo, numero di nodi $(G, k)$	$\rightsquigarrow$	Una coppia grafo, numero di nodi $(H, l)$
Istanza-sì	$\exists$ un IS di taglia $k$ in $G$		$\exists$ un VC di taglia al più $l$ in $H$
Istanza-no	$\nexists$ un IS di taglia $k$ in $G$		$\nexists$ un VC di taglia al più $l$ in $H$

La riduzione si basa sul seguente lemma:

### Lemma

Sia  $G = \langle V, E \rangle$  un grafo, sia  $S \subseteq V$  un insieme di nodi.

$S$  è un Independent Set  $\iff \bar{S}$  è un Vertex Cover

### Dimostrazione

$\Rightarrow$ ) Sia  $S$  un IS.

Supponiamo per assurdo che  $\bar{S}$  non sia un VC.

Questo vuol dire che in  $G$  c'è un arco i cui estremi non stanno in  $\bar{S}$  (entrambi).

Ma se gli estremi non stanno in  $\bar{S}$ , allora stanno in  $S$ .

Essendo collegati tra loro,  $S$  non sarebbe un IS  $\Rightarrow \bar{S}$  è un VC.

$\Leftarrow$ ) Sia  $\bar{S}$  un VC.

Supponiamo per assurdo che  $S$  non sia un IS.

Questo significa che ci sono due nodi in  $S$  collegati tra loro da un arco.

Se  $\bar{S}$  non contiene nessuno dei due estremi, allora  $\bar{S}$  non è un VC  $\Rightarrow S$  è un IS.

Il grafo  $H$  resta tale e quale a  $G$ , mentre la taglia  $l$  diventa il numero di nodi meno  $k$ .

Quindi  $H = G$  e  $l = |V| - k$ . Ovviamente la trasformazione è polinomiale.

Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $G$  abbia un IS  $S$  di taglia  $k$ .

Per il lemma precedente,  $\bar{S}$  è un VC di  $G$ .

Siccome  $H = G$ ,  $\bar{S}$  è un VC anche di  $H$ .

La taglia  $l$  del VC è il numero di nodi del grafo, tolti i nodi dell'IS.

$\Leftarrow$ ) Supponiamo di essere arrivati a un grafo  $H$  avente un VC  $C$  di taglia  $l$ .

Per il lemma precedente,  $\bar{C}$  è un IS di  $H$ .

Siccome  $G = H$ ,  $\bar{C}$  è un IS anche di  $G$ , di taglia  $k$  pari al numero di nodi del grafo, tolti i nodi del VC.

Siccome IS è riducibile polinomialmente a VC  $\Rightarrow$  VC è NP-arduo.

VC  $\in NP \wedge$  VC è NP-arduo  $\Rightarrow$  VC è NP-completo.

### Clique

Dato un grafo, stabilire se al suo interno esiste una clique di taglia  $k$ .

Una clique è una sezione del grafo completamente connessa. Un grafo vuoto è sempre una clique, perciò la formulazione del problema richiede una clique di almeno  $k$  elementi.

CLIQUE = { $\langle G, k \rangle \mid \exists$  una clique di taglia  $k$  in  $G$ }

CLIQUE  $\in NP$  poiché tramite una MT non deterministica possiamo indovinare l'insieme di  $k$  nodi che forma la clique e verificare che sia corretto, in tempo polinomiale.

Per dimostrare che CLIQUE è NP-arduo, eseguiamo una riduzione a partire da IS.

	<b>IS</b>	$\leq_P$	<b>CLIQUE</b>
<b>Input</b>	Una coppia grafo, numero di nodi $(G, k)$	$\rightsquigarrow$	Una coppia grafo, numero di nodi $(H, l)$
<b>Istanza-sì</b>	$\exists$ un IS di taglia $k$ in $G$		$\exists$ una clique di taglia $l$ in $H$
<b>Istanza-no</b>	$\nexists$ un IS di taglia $k$ in $G$		$\nexists$ una clique di taglia $l$ in $H$

Sia  $G = \langle V, E \rangle$  un grafo. Denotiamo con  $\bar{G} = \langle V, \bar{E} \rangle$  il grafo in cui i nodi restano gli stessi, mentre gli archi sono l'opposto: se una coppia di nodi non è collegata in  $G$ , lo sarà in  $\bar{G}$ , e viceversa.

La riduzione si basa sul seguente lemma:

### Lemma

Sia  $G$  un grafo.

$$S \text{ è un IS di } G \iff S \text{ è una clique di } \bar{G}$$

### Dimostrazione

$\Rightarrow$ ) Sia  $S$  un IS di  $G$ .

Supponiamo per assurdo che  $S$  non sia una clique di  $\bar{G}$ .

Questo vuol dire che in  $S$  ci sono due nodi che in  $\bar{G}$  non sono collegati.

Ma allora tali due nodi in  $G$  sono collegati  $\implies S$  non è un IS.

Contraddizione, quindi  $S$  è una clique di  $\bar{G}$ .

$\Leftarrow$ ) Sia  $S$  una clique di  $\bar{G}$ .

Supponiamo per assurdo che  $S$  non sia un IS di  $G$ .

Questo significa che due nodi contenuti in  $S$  sono collegati in  $G$ .

Ma allora tali due nodi in  $\bar{G}$  sono scollegati  $\implies S$  non è una clique.

Contraddizione, quindi  $S$  è un IS di  $G$ .

Il grafo  $H$  si ottiene complementando  $G$ , mentre la taglia  $l$  resta uguale a  $k$ .

Quindi  $H = \bar{G}$  e  $l = k$ . Tale trasformazione è polinomiale.

Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $G$  abbia un IS  $S$  di taglia  $k$ .

Per il lemma precedente,  $S$  è una clique di  $\bar{G}$  di taglia  $k$ .

Quindi per definizione di  $H$  e  $l$ ,  $S$  è una clique di  $H$  di taglia  $l$ .

$\Leftarrow$ ) Supponiamo di essere arrivati a un grafo  $H$  avente una clique  $S$  di taglia  $l$ .

Per il lemma precedente,  $S$  è un IS di  $\bar{H}$  di taglia  $l$ .

Quindi per definizione di  $H$  e  $l$ ,  $S$  è un IS di  $G$  di taglia  $k$ .

Siccome IS è riducibile polinomialmente a CLIQUE  $\implies$  CLIQUE è NP-arduo.

CLIQUE  $\in NP \wedge$  CLIQUE è NP-arduo  $\implies$  CLIQUE è NP-completo.

## Dominating Set (DS)

Dato un grafo, stabilire se al suo interno esiste un DS di taglia al più  $k$ .

Un dominating set è un insieme di nodi tale per cui i nodi che non ne fanno parte vi sono collegati tramite un arco. L'insieme di tutti i nodi è sempre un DS, perciò la formulazione del problema richiede un DS con al massimo  $k$  elementi.

$$\text{DS} = \{ \langle G, k \rangle \mid \exists \text{ un DS di taglia al più } k \text{ in } G \}$$

$\text{DS} \in NP$  poiché tramite una MT non deterministica possiamo indovinare l'insieme con al massimo  $k$  nodi che compone il DS per poi verificare che sia corretto, in tempo polinomiale.

Un VC è sempre un DS, ma non viceversa. Sfruttiamo questo legame per mostrare che DS è NP-arduo, tramite una riduzione da VC.

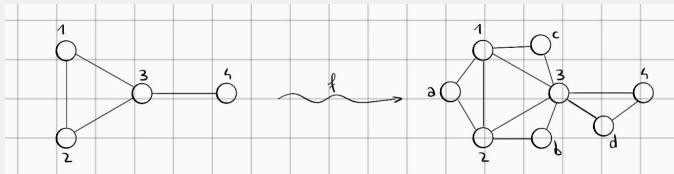
	<b>VC</b>	$\leq_P$	<b>DS</b>
<b>Input</b>	Una coppia grafo, numero di nodi $(G, k)$	$\rightsquigarrow$	Una coppia grafo, numero di nodi $(H, l)$
<b>Istanza-sì</b>	$\exists$ un VC di taglia al più $k$ in $G$		$\exists$ un DS di taglia al più $l$ in $H$
<b>Istanza-no</b>	$\nexists$ un VC di taglia al più $k$ in $G$		$\nexists$ un DS di taglia al più $l$ in $H$

Se lasciassimo il grafo invariato, non sarebbe garantito che avere un DS in  $H$  implichi l'esistenza di un VC in  $G$ , quindi  $H$  deve essere alterato.

Per ogni arco in  $G$ , aggiungiamo un nodo in  $H$  e lo colleghiamo agli estremi dell'arco di  $G$ .

La taglia  $l$  resta uguale a  $k$ , quindi  $l = k$ .

Esempio di trasformazione



Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $G$  abbia un VC  $C$  di taglia  $k$ .

Questo significa che da  $C$  riusciamo a raggiungere tutti gli archi di  $G$ .

Siccome i nodi che abbiamo aggiunto per costruire  $H$  sono tutti collegati in corrispondenza degli archi originari di  $G$ , riusciamo a raggiungerli da  $C$ .

Quindi  $C$  è un DS di  $H$  di taglia  $l = k$ .

$\Leftarrow$ ) Supponiamo di essere arrivati a un grafo  $H$  avente un DS  $D$  di taglia  $l$ .

Questo  $D$ , per come è costruito  $H$ , può contenere nodi originari o aggiunti.

In  $D$ , sostituiamo ogni nodo aggiunto con uno dei due originari a cui è collegato, ottenendo un  $D'$  con solo nodi originari.

Siccome  $D'$  è un DS di  $H$ , deve raggiungere tutti i nodi che "mimano" gli archi di  $G$ , pertanto  $D'$  deve essere un VC in  $G$  di taglia  $k = l$ .

Siccome VC è riducibile polinomialmente a DS  $\implies$  DS è NP-arduo.

DS  $\in NP \wedge$  DS è NP-arduo  $\implies$  DS è NP-completo.

## Certificati

Fino ad ora, per stabilire se un linguaggio appartenesse a  $NP$ , abbiamo descritto una MT non deterministica che, in tempo polinomiale, indovina una soluzione e ne verifica la correttezza.

È dimostrabile che la fase di check può essere più semplice di polinomiale, usando la logica del prim'ordine.

Ci chiediamo se tutti i linguaggi di  $NP$  possano essere decisi eseguendo guess and check in quest'ordine, oppure se esistono linguaggi per cui il non determinismo viene applicato alla fine.

Un **certificato** è la prova (fin'ora indovinata) che un'istanza di un problema sia un'istanza-sì. È conciso, perché non può essere più grande di polinomiale, ed è polinomialmente verificabile, in quanto il check viene effettuato in tempo polinomiale.

La nostra definizione della classe  $NP$  si basa sulle MT non deterministiche, macchine che nessuno sarà mai in grado di costruire fisicamente. Sono state quindi pensate definizioni differenti. Vogliamo arrivare a definire  $NP$  come i linguaggi le cui istanze-sì ammettono certificati concisi e polinomialmente verificabili.

Sia  $\mathcal{R}$  una relazione binaria, definita come  $\mathcal{R} \subseteq \Sigma^* \times \Sigma^*$ , composta da coppie di stringhe  $\langle x, y \rangle$ .

Diciamo che  $\mathcal{R}$  è **polynomialmente bilanciata** se  $\|y\| \leq \|x\|^c$  con  $c$  fissato.

Ovvero la lunghezza della stringa in seconda posizione non eccede un polinomio della lunghezza della stringa in prima posizione. La costante  $c$  indica quanto  $y$  può essere più grande di  $x$ .

Diciamo che  $\mathcal{R}$  è **polynomialmente decidibile** se  $\mathcal{R}$  si può decidere in tempo polinomiale deterministico.

Ovvero se esiste una MT deterministica che può decretare in tempo polinomiale se una coppia appartiene alla relazione o meno.

## Teorema

Sia  $L$  un linguaggio.

$L \in NP \iff \exists$  una relazione binaria  $\mathcal{R}_L$  polinomialmente bilanciata e decidibile, tale che

$$L = \{x \mid \langle x, y \rangle \in \mathcal{R}_L\}$$

### Dimostrazione

$\Leftarrow$ ) Supponiamo che  $\exists$  una relazione  $\mathcal{R}_L$  polinomialmente bilanciata e decidibile, tale che

$$L = \{x \mid \langle x, y \rangle \in \mathcal{R}_L\}$$

Se  $\mathcal{R}_L$  è polinomialmente decidibile  $\implies \exists$  una MT  $M$  deterministica che in tempo polinomiale è in grado di decidere se una coppia  $\langle x, y \rangle \in \mathcal{R}_L$ .

Tramite una MT  $N$  non deterministica indoviniamo il certificato  $y$ , che sarà di lunghezza

$O(\|x\|^c)$  poiché non avrebbe tempo di scriverne uno più lungo.

Sempre dentro a  $N$ , la coppia  $\langle x, y \rangle$  viene passata alla macchina  $M$ , il cui output diventa l'output di  $N$ .

$\exists$  una MT non deterministica che decide  $L$  in tempo polinomiale  $\implies L \in NP$ .

$\Rightarrow$ ) Sia  $L \in NP$ .

Mostreremo che la fase di guess può essere effettuata tutta all'inizio, senza necessità di farla negli stadi avanzati della computazione.

Se  $L \in NP \implies \exists$  una MT non deterministica  $M$  che decide  $L$  in tempo polinomiale.

Se  $M$  decide  $L$  in tempo polinomiale, il ramo di computazione più lungo è al più polinomiale nella lunghezza dell'input.

Sia  $p_1(n)$  tale polinomio, dove  $n$  è la taglia dell'input.

Se la macchina lavora in tempo  $p_1$ , non avrà mai il tempo di scrivere sul nastro una stringa più lunga di  $p_1(n)$  simboli, quindi ciascuna descrizione istantanea di  $M$  avrà lunghezza polinomiale.

Possiamo scrivere la sequenza di configurazioni di  $M$  come una stringa unica, attestante che la MT accetti la stringa in input.

Il numero totale di configurazioni è al massimo  $p_1(n)$ , ognuna di lunghezza al più polinomiale  $\implies$  tale stringa, che avrà il ruolo di certificato, ha taglia polinomiale.

Abbiamo quindi generato un certificato che non dipende dalla struttura del problema, bensì dalla computazione della macchina che decide il linguaggio. Questo certificato ha senso, in quanto le stringhe non appartenenti a  $L$  non hanno una computazione accettante con cui generarlo.

Il certificato è conciso, ovvero di lunghezza polinomiale nella lunghezza della stringa  $x \in L$   $\implies \mathcal{R}_L$  è polinomialmente bilanciata.

Il certificato è polinomialmente verificabile, in quanto è la descrizione della computazione della macchina, quindi per verificarne la correttezza è sufficiente controllare che le configurazioni si susseguano legalmente rispetto alla funzione di transizione di  $M \implies \mathcal{R}_L$  è polinomialmente decidibile.

In conclusione, il linguaggio  $L \in NP$  se è possibile inventare una relazione binaria, con in prima posizione le stringhe  $w \in L$  e in seconda posizione il certificato che ne attesta l'appartenenza.

Esempio per Independent Set

$$IS = \{ x \mid \langle x, y \rangle \in \mathcal{R}_{IS} \}$$

Dove  $\mathcal{R}_{IS}$  è una relazione binaria in cui:

- $x$  è una coppia grafo, numero di nodi  $(G, k)$
- $y$  è un IS di taglia  $k$

Ecco dunque la nuova caratterizzazione di  $NP$ :

I linguaggi di  $NP$  sono quei linguaggi le cui soluzioni possono essere verificate in tempo polinomiale deterministico.

Intuitivamente, i problemi in  $P$  sono quelli per cui può essere calcolata facilmente una soluzione, i problemi in  $NP$  sono quelli per cui la soluzione può soltanto essere verificata facilmente.

La questione  $P = NP$  si traduce quindi in: "tutti i problemi la cui soluzione può essere verificata facilmente, sono anche problemi la cui soluzione è calcolabile facilmente?"

## Colorabilità di un grafo

Dato un grafo, stabilire se è colorabile con al più  $k$  colori.

Colorare un grafo significa assegnare ad ogni nodo un colore, facendo in modo che nodi adiacenti abbiano colori diversi.

$$COL = \{ \langle G, k \rangle \mid \text{è possibile colorare i nodi di } G \text{ con al più } k \text{ colori} \}$$

$COL \in NP$  in quanto tramite una MT non deterministica possiamo indovinare l'assegnamento dei colori ai nodi e poi verificare che sia corretto, in tempo polinomiale.

Per dimostrare che sia NP-arduo, riduciamo EXACT-3SAT alla colorabilità.

	<b>3SAT</b>	$\leq_P$	<b>COL</b>
<b>Input</b>	Una formula booleana $\phi$ in 3CNF	$\rightsquigarrow$	Una coppia grafo, numero di colori $(G, k)$
<b>Istanza-sì</b>	$\phi$ è soddisfacibile		Il grafo $G$ può essere colorato con al più $k$ colori
<b>Istanza-no</b>	$\phi$ non è soddisfacibile		Il grafo $G$ non può essere colorato con al più $k$ colori

Da un lato dobbiamo stabilire se una variabile è true o false, dall'altro dobbiamo stabilire il colore da dare ad un nodo.

Partiamo generando una clique con 3 nodi, per cui serviranno almeno 3 colori. Un colore rappresenterà i nodi true (verde), un colore rappresenterà i nodi false (rosso) e l'ultimo colore sarà "altro" (blu).

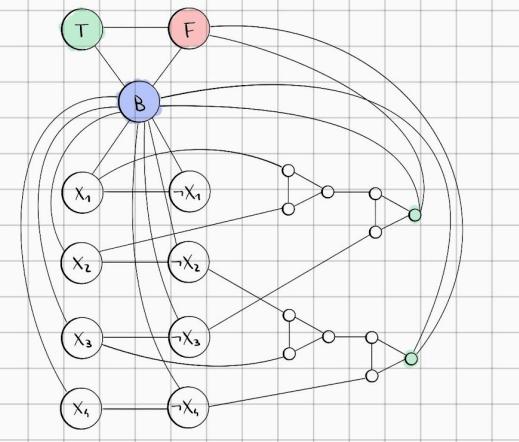
Per ogni variabile di  $\phi$ , generiamo due nodi:  $x_i$  e  $\neg x_i$ . Siccome dovranno avere colori diversi tra loro, li colleghiamo. Per far sì che possano assumere solo il colore verde o rosso, li colleghiamo tutti al nodo blu.

Ora dobbiamo simulare l'OR tramite colorazione. Lo facciamo costruendo una struttura apposita (vedi disegno esempio). Per forzare l'output di queste strutture ad essere true (verde), lo colleghiamo ai colori rosso e blu.

Il numero di colori  $k$  definitivo è 3.

### Esempio di trasformazione

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$$



Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che la formula  $\phi$  sia soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento di verità  $\sigma$  che rende true tutte le clausole di  $\phi$ .

Per la trasformazione, otteniamo un grafo  $G$  in cui la clique è colorata con i tre colori, esattamente in quel modo. Diamo poi ad ogni nodo corrispondente alle variabili un colore in base a  $\sigma$ . Siccome  $\phi$  è soddisfacibile, siamo in grado di colorare coerentemente la parte restante del grafo (quella degli OR).

Perciò  $G$  è colorabile con al più  $k = 3$  colori.

$\Leftarrow$ ) Supponiamo di essere arrivati a un grafo  $G$  colorabile con  $k = 3$  colori.

Costruiamo un assegnamento di verità  $\sigma$ , assegnando true o false alle variabili in base al colore che assumono i corrispondenti nodi nel grafo  $G$ .

Tale assegnamento è consistente poiché nodi relativi alla stessa variabile sono collegati.

Supponiamo per assurdo che  $\sigma$  non soddisfi la formula  $\phi$ .

Questo significa che una clausola di  $\phi$  è interamente false.

Quindi tutti i nodi corrispondenti ai letterali di tale clausola sarebbero rossi, portando l'output dell'OR ad essere rosso, ma ciò è impossibile per il collegamento alla clique  $\implies \sigma$  soddisfa  $\phi$ .

Siccome EXACT-3SAT è riducibile polinomialmente a COL  $\implies$  COL è NP-arduo.

COL  $\in NP \wedge$  COL è NP-arduo  $\implies$  COL è NP-completo.

## Exact Cover (EC)

Dato un insieme di oggetti  $U = \{ u_1, \dots, u_n \}$ , chiamato insieme universo, ed una famiglia  $F = \{ S_1, \dots, S_m \}$  tale che  $S_i \subseteq U$ , stabilire se tra gli insiemi in  $F$  esiste una partizione di  $U$ . Una partizione di  $U$  è una selezione di insiemi  $S_i \in F$  disgiunti tra loro, la cui unione è precisamente  $U$ .

$$\text{EC} = \{ \langle U, F \rangle \mid \exists \text{ una partizione di } U \text{ formata da insiemi } S_i \in F \}$$

Tramite una MT non deterministica possiamo indovinare gli insiemi di  $F$  che compongono una partizione di  $U$  e poi verificarne la correttezza, in tempo polinomiale, perciò  $\text{EC} \in NP$ .

Per dimostrare che sia NP-arduo, effettuiamo una riduzione a partire da EXACT-3SAT.

	<b>3SAT</b>	$\leq_P$	<b>EC</b>
<b>Input</b>	Una formula booleana $\phi$ in 3CNF	$\rightsquigarrow$	Una coppia insieme universo, famiglia $(U, F)$
<b>Istanza-sì</b>	$\phi$ è soddisfacibile		In $F \exists$ una partizione di $U$
<b>Istanza-no</b>	$\phi$ non è soddisfacibile		In $F \nexists$ una partizione di $U$

Consideriamo una formula  $\phi = c_1 \wedge \dots \wedge c_n$ , in cui le clausole sono tipo  $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ .

L'insieme universo  $U_\phi$  conterrà:

- un oggetto  $x_i$  per ciascuna variabile all'interno di  $\phi$
- un oggetto  $c_j$  per ogni clausola della formula  $\phi \rightarrow$  per indicare se la clausola è soddisfatta
- un oggetto  $p_{j,k}$  per ogni letterale che compare nella clausola  $j$

La famiglia  $F_\phi$  conterrà:

- un insieme singoletto  $\{ p_{j,k} \}$  per ogni oggetto  $p_{j,k}$
- due insiemi per ogni variabile  $x_i$ :  $T_{i,T}$ , con dentro l'oggetto  $x_i$  assieme a tutti gli oggetti  $p_{j,k}$  legati ai letterali che risultano false quando  $x_i$  è true, e  $T_{i,F}$ , con dentro l'oggetto  $x_i$  assieme a tutti gli oggetti  $p_{j,k}$  legati ai letterali che risultano false quando  $x_i$  è false  $\rightarrow$  per simulare gli assegnamenti di verità
- per ciascuna clausola  $c_j$ , le coppie  $\{ c_j, p_{j,1} \}, \{ c_j, p_{j,2} \}, \{ c_j, p_{j,3} \}$

Esempio: trasformazione di  $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$$U_\phi = \{ x_1, x_2, x_3, x_4, c_1, c_2, p_{1,1}, p_{1,2}, p_{1,3}, p_{2,1}, p_{2,2}, p_{2,3} \}$$

$$\begin{aligned} F_\phi = & \{ \{ p_{1,1} \}, \{ p_{1,2} \}, \{ p_{1,3} \}, \{ p_{2,1} \}, \{ p_{2,2} \}, \{ p_{2,3} \}, T_{1,T}, T_{2,T}, T_{3,T}, T_{4,T}, \\ & T_{1,F}, T_{2,F}, T_{3,F}, T_{4,F}, \{ c_1, p_{1,1} \}, \{ c_1, p_{1,2} \}, \{ c_1, p_{1,3} \}, \{ c_2, p_{2,1} \}, \{ c_2, p_{2,2} \}, \{ c_2, p_{2,3} \} \} \\ T_{1,T} = & \{ x_1, p_{2,1} \}, T_{2,T} = \{ x_2, p_{1,2} \}, T_{3,T} = \{ x_3 \}, T_{4,T} = \{ x_4 \} \\ T_{1,F} = & \{ x_1, p_{1,1} \}, T_{2,F} = \{ x_2, p_{2,2} \}, T_{3,F} = \{ x_3, p_{1,3} \}, T_{4,F} = \{ x_4, p_{2,3} \} \end{aligned}$$

Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che  $\phi$  sia soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento di verità  $\sigma$  che rende vera  $\phi$ .

Costruiamo una partizione di  $U_\phi$  basandoci su  $\sigma$ , prendendo i seguenti elementi di  $F_\phi$ :

- $T_{i,T}$  se  $\sigma[x_i]$  è true,  $T_{i,F}$  se  $\sigma[x_i]$  è false

- le coppie  $\{c_j, p_{j,k}\}$  tali che il letterale  $l_{j,k}$  è true in  $\sigma$  e non appare nei  $T_{i,T}$  o  $T_{i,F}$

- tutti i singoletti  $\{p_{j,k}\}$  rimasti fuori

Con il primo passaggio copriamo tutti gli  $x_i$  e gli oggetti legati ai letterali false. Con il secondo copriamo tutti i  $c_j$  e gli oggetti legati ai letterali true. Infine, con il terzo, copriamo i  $p_{j,k}$  non considerati.

Pertanto  $\exists$  una partizione di  $U_\phi$  in  $F_\phi$ .

$\Leftarrow$ ) Supponiamo di essere arrivati ad una famiglia  $F_\phi$  contenente una partizione  $\mathcal{P}$  di  $U_\phi$ .

Vogliamo costruire un assegnamento  $\sigma$  partendo da  $\mathcal{P}$ :

$\sigma_{\mathcal{P}}[x_i]$  sarà true se  $T_{i,T} \in \mathcal{P}$ , sarà false se  $T_{i,F} \in \mathcal{P}$

Tutte le variabili  $x_i$  sono considerate, poiché in  $\mathcal{P}$  c'è un  $T$  per ognuna di esse. Inoltre,  $\sigma_{\mathcal{P}}$  è consistente, poiché in  $\mathcal{P}$  non possono esserci due  $T$  per la stessa variabile.

Supponiamo per assurdo che  $\sigma_{\mathcal{P}}$  non soddisfi  $\phi$ .

Questo significa che  $\exists$  una clausola  $c_j$  in cui tutti i letterali assumono valore false.

Ma ciò implica che dentro a  $\mathcal{P}$  non abbiamo l'oggetto  $c_j$ .

Impossibile,  $\mathcal{P}$  non sarebbe una partizione  $\implies \sigma_{\mathcal{P}}$  soddisfa  $\phi$ .

Siccome EXACT-3SAT è riducibile polinomialmente a EC  $\implies$  EC è NP-arduo.

EC  $\in NP \wedge$  EC è NP-arduo  $\implies$  EC è NP-completo.

## Knapsack (problema dello zaino)

Dato un insieme di oggetti  $O = \{1, \dots, n\}$ , dove ogni oggetto ha un peso  $w_i$  e un valore  $v_i$ , stabilire se esiste un sottoinsieme di oggetti il cui peso totale non superi  $W$  e il cui valore totale sia almeno  $K$ .

$$KNAP = \{ \langle O, w, v, W, K \rangle \mid \exists S \subseteq O, \sum_{i \in S} w_i \leq W \wedge \sum_{i \in S} v_i \geq K \}$$

Knapsack  $\in NP$  poiché tramite una MT non deterministica possiamo indovinare il sottoinsieme di oggetti e verificarne il suo peso e valore, in tempo polinomiale.

Effettuiamo una riduzione da EC per dimostrare che Knapsack è NP-arduo.

	<b>EC</b>	$\leq_P$	<b>Knapsack</b>
<b>Input</b>	Una coppia insieme universo, famiglia $(U, F)$	$\rightsquigarrow$	Un insieme di oggetti, uno di pesi, uno di valori e due soglie intere $(O, w, v, W, K)$
<b>Istanza-sì</b>	In $F \exists$ una partizione di $U$		$\exists S \subseteq O$ tale che la somma dei pesi non superi $W$ e la somma dei valori sia almeno $K$
<b>Istanza-no</b>	In $F \nexists$ una partizione di $U$		$\nexists S \subseteq O$ tale che la somma dei pesi non superi $W$ e la somma dei valori sia almeno $K$

Facciamo in modo che gli oggetti abbiano un valore pari al peso, quindi  $\forall i w_i = v_i$ . Perciò le soglie saranno identiche, ovvero  $W = K$ . Di conseguenza, dovendo valere contemporaneamente sia un  $\leq$  che un  $\geq$ , possiamo ridurci a cercare un sottoinsieme il cui peso sia esattamente  $W$ .

Descriviamo la composizione degli insiemi appartenenti ad  $F$  tramite la codifica in binario di un numero: la j-esima cifra sarà 1 se il j-esimo elemento di  $U$  appartiene all'insieme, altrimenti sarà 0. La soglia  $W$  è data dal numero con soli 1.

Il nostro obiettivo è quindi selezionare i numeri in modo che sommandoli il risultato sia  $W$ .

Eseguendo la somma abbiamo però il problema del riporto. La soluzione è interpretare i numeri in una base tale che non ci possa essere riporto, ovvero, assumendo di generare  $m$  numeri, base  $m + 1$ .

$$w_i = \sum_{j \in F_i} 1 \cdot (m+1)^{n-j}$$

Esempio di trasformazione

$$U = \{1, 2, 3, 4\}, F = \{\{3, 4\}, \{2, 4\}, \{2, 3, 4\}\}$$

$$\{3, 4\} = (0011)_2 = w_1$$

$$\{2, 4\} = (0101)_2 = w_2$$

$$\{2, 3, 4\} = (0111)_2 = w_3$$

$$W = (1111)_2$$

In questo caso interpretiamo i numeri in base 4.

Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che dentro  $F$  ci siano dei sottoinsiemi che costituiscono una partizione di  $U$ . Le stringhe binarie rappresentanti tali sottoinsiemi non possono avere 1 in comune sulle colonne, poiché i sottoinsiemi sono disgiunti.

Inoltre, su ogni colonna c'è almeno un 1, dato che i sottoinsiemi coprono tutto  $U$ .

Quindi sommando i rispettivi numeri  $w_i$  otteniamo una stringa di soli 1 (nella base specifica)

$\implies \exists$  un sottoinsieme di  $O$  il cui peso totale è esattamente  $W$ .

$\Leftarrow$ ) Supponiamo di essere arrivati ad una specifica istanza-sì di Knapsack.

Questo significa che  $\exists$  un sottoinsieme dei numeri  $w_i$  la cui somma è esattamente  $W$ .

Tale  $W$  è il numero la cui rappresentazione in base  $m + 1$  è formata da soli 1.

Siccome abbiamo al più  $m$  numeri  $w_i$ , in base  $m + 1$  non potremo mai generare un riporto sommando solo degli 1.

Perciò se il risultato di tale somma è composto da soli 1, vuol dire che in ogni colonna c'è precisamente una cifra 1.

Questo significa che i  $w_i$  considerati rappresentano insiemi di  $F$  tra loro disgiunti, la cui unione equivale all'insieme universo.

$\exists$  una partizione di  $U$  in  $F \implies$  partivamo da un'istanza-sì di EC.

Siccome EC è riducibile polinomialmente a Knapsack  $\implies$  Knapsack è NP-arduo.

$\text{Knapsack} \in NP \wedge \text{Knapsack è NP-arduo} \implies \text{Knapsack è NP-completo.}$

Dato che Knapsack si può esprimere come un problema di PLI (Programmazione Lineare Intera), tutta la PLI è NP-ardua.

## Teorema di Cook

Fino ad ora, per dimostrare che un linguaggio fosse NP-arduo, abbiamo sfruttato la transitività delle riduzioni partendo da linguaggi NP-ardui. Il linguaggio alla radice di questo procedimento è SAT, il quale si può dimostrare formalmente essere NP-arduo. Per fare ciò, bisogna mostrare che tutti i linguaggi di  $NP$  sono riducibili a SAT.

La classe  $NP$  contiene un numero infinito di linguaggi, pertanto effettuare infinite riduzioni non è una strada percorribile. Mostreremo che per un qualsiasi linguaggio  $L \in NP$  esiste una riduzione verso SAT. Non saremo in grado di dettagliare tale riduzione poiché non conosciamo  $L$ , tuttavia forniremo uno scheletro sufficientemente preciso affinché se  $L \in NP \implies \exists$  una riduzione da  $L$  a SAT, da cui segue che SAT è NP-arduo.

Nello svolgimento della prova useremo una sintassi booleana un po' più ricca, comunque equivalente alla CNF, come affermato dalle seguenti proprietà:

- Distributività  $\rightarrow (A_1 \wedge \dots \wedge A_n) \vee (B_1 \wedge \dots \wedge B_m) \equiv (A_1 \vee B_1) \wedge (A_1 \vee B_2) \wedge \dots \wedge (A_1 \vee B_m) \wedge (A_2 \vee B_1) \wedge \dots \wedge (A_n \vee B_m)$
- De Morgan  $\rightarrow \neg(A_1 \wedge \dots \wedge A_n) \equiv \neg A_1 \vee \dots \vee \neg A_n$

- Implicazione (1)  $\rightarrow \phi \implies \psi \equiv \neg\phi \vee \psi$
- Implicazione (2)  $\rightarrow \phi \implies \psi_1 \wedge \psi_2 \equiv (\phi \implies \psi_1) \wedge (\phi \implies \psi_2)$

Esempio di utilizzo delle proprietà

$$\begin{aligned}
 A \wedge B &\implies (C \wedge D) \vee (E \wedge F) \equiv \\
 A \wedge B &\implies (C \vee E) \wedge (C \vee F) \wedge (D \vee E) \wedge (D \vee F) \equiv \\
 (A \wedge B \implies C \vee E) &\wedge (A \wedge B \implies C \vee F) \wedge (A \wedge B \implies D \vee E) \wedge \\
 (A \wedge B \implies D \vee F) &\equiv \\
 (\neg A \vee \neg B \vee C \vee E) &\wedge (\neg A \vee \neg B \vee C \vee F) \wedge (\neg A \vee \neg B \vee D \vee E) \wedge \\
 (\neg A \vee \neg B \vee D \vee F)
 \end{aligned}$$

## SAT

Data una formula booleana in CNF, stabilire se è soddisfacibile, ovvero se esiste un assegnamento di verità per le variabili che la rende vera.

$SAT \in NP$  poiché tramite una MT non deterministica possiamo indovinare l'assegnamento e poi verificarlo, in tempo polinomiale.

Sia  $L$  un generico linguaggio di  $NP$ , lo vogliamo ridurre polinomialmente a SAT.

	$L$	$\leq_P$	SAT
<b>Input</b>	Una stringa $w$	$\rightsquigarrow$	Una formula booleana $\phi$ in CNF
<b>Istanza-sì</b>	$w \in L$		$\phi$ è soddisfacibile
<b>Istanza-no</b>	$w \notin L$		$\phi$ non è soddisfacibile

L'unica cosa che sappiamo di  $L$  è che appartiene a  $NP$ .

Se  $L \in NP \implies \exists$  una MT non deterministica  $M$  che in tempo polinomiale decide  $L$ .

Assumiamo che il running time di  $M$  sia  $p(n)$ , dove  $n = \|w\|$ .

Per semplificare la prova, facciamo alcune assunzioni sulla macchina  $M$  (non tolgo generalità):

- non scrive mai  $\not b$
- ha un nastro semi-infinito, quindi c'è una cella 0
- ci mette esattamente  $p(n)$  passi su tutti i rami (quelli più corti eseguono delle operazioni nulle, lasciando invariata la configurazione finale)

La riduzione trasforma la stringa  $w$  in una formula  $\phi_w$  che simula il comportamento della macchina  $M$  su  $w$  tramite delle variabili proposizionali.

Attenzione: la funzione  $f$  è creata a partire da  $M$ , ma in input prende soltanto la stringa  $w \rightarrow$  la denotiamo con  $f_M(w)$ .

Le variabili proposizionali che usiamo per descrivere le configurazioni della macchina sono:

- $q_{i,k}$  vale true  $\iff$  al passo  $i$ , la macchina  $M$  si trova nello stato  $q_k$
- $h_{i,j}$  vale true  $\iff$  al passo  $i$ , la testina di  $M$  si trova sulla  $j$ -esima cella del nastro
- $t_{i,j,l}$  vale true  $\iff$  al passo  $i$ , la  $j$ -esima cella del nastro contiene il simbolo  $\alpha_l \in \Gamma$

Innanzitutto verifichiamo che il numero di variabili generate sia polinomiale.

Sia  $r$  il numero di stati della macchina  $M$ ,  $s$  il numero di simboli dell'alfabeto  $\Gamma$ .

- $q_{i,k} \rightarrow$  una per ogni stato, per ogni passo =  $p(n) \cdot r$
- $h_{i,j} \rightarrow$  una per ogni cella, per ogni passo =  $p(n)^2$
- $t_{i,j,l} \rightarrow$  una per ogni simbolo in ogni cella, per ogni passo =  $p(n)^2 \cdot s$

Il risultato finale è una somma di polinomi, perciò la taglia della formula è polinomiale.

Siccome la formula  $\phi_w$  deve simulare il funzionamento della macchina  $M$  su  $w$  tramite le variabili proposizionali, la costruiamo suddivisa logicamente in pezzi, ognuno dei quali imporrà determinati vincoli sulla formula.

Il primo pezzo, **CONSISTENCY** ( $C$ ), si occupa della consistenza dell'assegnamento di verità alle variabili: per ogni passo  $i$ , la macchina è in un solo stato e la sua testina legge un solo simbolo da una sola cella. Un pezzo **START** ( $S$ ) deve codificare il fatto che la macchina comincia la computazione dallo stato iniziale, avendo sul nastro l'intera stringa  $w$ . C'è poi un pezzo **NEXT STEP** ( $N$ ) che codifica i passi legali secondo la funzione di transizione della macchina. Infine, il pezzo **FINAL STEP** ( $F$ ) codifica la transizione verso gli stati finali.

$$\phi_w = C \wedge S \wedge N \wedge F$$

**CONSISTENCY -  $C$ :**

Questo pezzo di formula ci dice che ad ogni passo la macchina si trova in al più uno stato e la sua testina legge al più un simbolo da al più una cella. Il vincolo che sia esattamente 1 consegnerà dai pezzi successivi.

$$C \equiv \bigwedge_{\substack{i, k, k' \\ k \neq k'}} (q_{i,k} \implies \neg q_{i,k'}) \wedge \bigwedge_{\substack{i, j, j' \\ j \neq j'}} (h_{i,j} \implies \neg h_{i,j'}) \wedge \bigwedge_{\substack{i, j, l, l' \\ l \neq l'}} (t_{i,j,l} \implies \neg t_{i,j,l'})$$

**START -  $S$ :**

Il contenuto di nostro interesse si trova sul nastro fra la posizione 0 e la posizione  $p(n)$ , siccome la macchina non ha il tempo di leggere oltre.

Questo pezzo codifica che al passo zero la macchina si trova nello stato  $q_0$ , la sua testina è sulla cella 0 e legge il simbolo iniziale della stringa  $w$ , mentre nelle celle successive ci sono i restanti simboli di  $w$ . Codifica inoltre che superato  $n$  ci sono soltanto simboli  $\not\in \Gamma$  fino a  $p(n)$ .

$$S \equiv q_{0,0} \wedge h_{0,0} \wedge t_{0,0,w_0} \wedge t_{0,1,w_1} \wedge \dots \wedge t_{0,n-1,w_{n-1}} \wedge t_{0,n,\not\in\Gamma} \wedge t_{0,n+1,\not\in\Gamma} \wedge \dots \wedge t_{0,p(n),\not\in\Gamma}$$

## NEXT STEP - $N$ :

Questo pezzo codifica l'evoluzione del contenuto del nastro secondo la funzione di transizione.

Sul nastro possiamo distinguere due aree: la cella dove sta la testina e tutte le altre celle.

**INERZIA ( $N^I$ )**: nelle celle dove non sta la testina, il contenuto resta invariato da un passo all'altro.

$$N^I \equiv \bigwedge_{i,j,l} (\neg h_{i,j} \wedge t_{i,j,l} \implies t_{i+1,j,l})$$

**TRANSIZIONE ( $N^H$ )**: la cella dove si trova la testina, da un passo all'altro cambia contenuto in base alla funzione di transizione. Questo sotto-pezzo codifica ogni entry della funzione di transizione del tipo  $\delta(q_k, \alpha_l) = \{(q_{k'}, \alpha_{l'}, \rightarrow), (q_{k''}, \alpha_{l''}, \leftarrow)\}$ .

$$N^H \equiv \bigwedge_{i,j,l} (q_{i,k} \wedge h_{i,j} \wedge t_{i,j,l} \implies (q_{i+1,k'} \wedge h_{i+1,j+1} \wedge t_{i+1,j,l'}) \vee \\ (q_{i+1,k''} \wedge h_{i+1,j-1} \wedge t_{i+1,j,l''}))$$

**PADDING ( $N^P$ )**: tutti i branch devono avere lunghezza  $p(n)$ , sia accettanti che non. Se sono più corti devono iterare sullo stato finale senza fare nulla.

$$N^P \equiv \bigwedge_{i,j,l} (q_{i,F} \wedge h_{i,j} \wedge t_{i,j,l} \implies q_{i+1,F} \wedge h_{i+1,j} \wedge t_{i+1,j,l}) \wedge \\ \bigwedge_{i,j,l} \bigwedge_{\substack{q_k \in Q, \alpha_l \in \Gamma \\ \delta(q_k, \alpha_l) = \emptyset}} (q_{i,k} \wedge h_{i,j} \wedge t_{i,j,l} \implies q_{i+1,k} \wedge h_{i+1,j} \wedge t_{i+1,j,l})$$

La seconda parte di  $N^P$  serve per i branch non accettanti.

Concludendo,  $N \equiv N^I \wedge N^H \wedge N^P$ .

## FINAL STEP - $F$ :

Questo pezzo codifica che all'ultimo passo lo stato è accettante (assumiamo che sia uno solo).

$$F \equiv q_{p(n),F}$$

In conclusione, la funzione  $f_M$  prende in input una stringa  $w$  e restituisce una formula booleana in CNF  $\phi_w$  molto specifica, la quale descrive il funzionamento della macchina  $M$  (fissata) su  $w$  tramite delle variabili proposizionali opportune. Questa formula, pur essendo gigantesca, è di taglia polinomiale nella grandezza dell'input.

Per come è stata definita la trasformazione, imponendo dei vincoli alle variabili, gli assegnamenti true o false devono necessariamente replicare il comportamento di  $M$  su  $w$ .

Pertanto  $w \in L \iff M \models w \iff \phi_w$  è soddisfacibile.

Siccome ogni linguaggio  $L \in NP$  è riducibile polinomialmente a SAT  $\implies$  SAT è NP-arduo.

SAT  $\in NP \wedge$  SAT è NP-arduo  $\implies$  SAT è NP-completo.

## Classe CO-NP

Analogamente a quanto avviene per la calcolabilità, indichiamo con  $CO-NP$  la classe di complessità temporale che contiene i linguaggi complemento di quelli in  $NP$ .

Nota:  $CO-NP \neq \overline{NP}$ , in quanto la seconda rappresenta tutto ciò che non appartiene a  $NP$ .

L'intuizione è speculare rispetto a quella di  $NP$ : un linguaggio appartiene a  $CO-NP$  se le sue istanze-no sono caratterizzate da un certificato conciso, ovvero siamo in grado di rispondere no indovinando qualcosa.

Alcuni linguaggi che si trovano in  $CO-NP$ :

- UNSAT: data una formula booleana in CNF, stabilire se è insoddisfacibile, ovvero se non esiste nessun assegnamento di verità che la rende vera.

$$UNSAT = \{ \phi \mid \phi \text{ è una formula booleana in CNF non soddisfacibile} \}$$

Indovinando un assegnamento che soddisfa  $\phi$  possiamo rispondere no, mentre dovremmo provarli tutti per poter rispondere sì.

- TAUTOLOGY: data una formula booleana in CNF, stabilire se è una tautologia, ovvero se è soddisfatta da qualunque assegnamento di verità.

$$TAUT = \{ \phi \mid \phi \text{ è una formula booleana in CNF sempre soddisfatta} \}$$

Possiamo rispondere no indovinando un assegnamento che rende  $\phi$  falsa. Al contrario, per rispondere sì dovremmo provare tutti i possibili assegnamenti.

All'interno della classe  $CO-NP$  ci sono linguaggi CO-NP-ardui, ovvero linguaggi tosti almeno quanto tutti i linguaggi di  $CO-NP$ . Un linguaggio  $L$  è CO-NP-arduo se  $\bar{L}$  è NP-arduo.

## Relazioni tra classi

A differenza dei risultati di calcolabilità, molte delle questioni riguardanti le relazioni tra  $CO-NP$ ,  $NP$  e  $P$  restano ancora aperte e irrisolte.

Per prima cosa, non sappiamo se  $NP$  e  $CO-NP$  siano due classi distinte  $\rightarrow NP \stackrel{?}{=} CO-NP$ . Ipotizziamo che siano distinte poiché nessuno è mai riuscito a dimostrare il contrario.

### Teorema

$$NP = CO-NP \iff \exists \text{ un linguaggio } L \text{ NP-completo tale che } L \in CO-NP$$

### Dimostrazione

$\Rightarrow$ ) Supponiamo che  $NP = CO-NP$ .

Se  $L$  è NP-completo  $\Rightarrow L \in NP \Rightarrow L \in CO-NP$ .

$\Leftarrow$ ) Supponiamo che  $\exists L$  NP-completo tale che  $L \in CO-NP$ .

Considerazioni: se  $L$  è NP-completo  $\Rightarrow L \in NP$ , da cui  $\bar{L} \in CO-NP$ . Similmente, se  $L \in CO-NP \Rightarrow \bar{L} \in NP$ . Perciò sia  $L$  che  $\bar{L}$  stanno in entrambe le classi.

-  $NP \subseteq CO-NP$

Sia  $L' \in NP$  un linguaggio qualsiasi.

Siccome  $L$  è NP-completo per ipotesi, abbiamo che  $L' \leq_P L$  (per definizione).

Questo significa che  $\exists$  una funzione di trasformazione  $f : \Sigma^* \rightarrow \Sigma^*$ ,

tale che  $\forall w, w \in L' \iff f(w) \in L$ .

Ciò è equivalente a dire che  $\forall w, w \notin L' \iff f(w) \notin L$ ,

che a sua volta equivale a  $\forall w, w \in \bar{L}' \iff f(w) \in \bar{L}$ .

Quindi possiamo affermare che  $\exists$  una riduzione da  $\bar{L}'$  a  $\bar{L}$ .

Secondo le considerazioni iniziali  $\bar{L} \in NP$ .

$\exists$  una riduzione da  $\bar{L}'$  a  $\bar{L} \implies \bar{L}' \in NP \implies L' \in CO-NP$ .

-  $CO-NP \subseteq NP$

Sia  $L' \in CO-NP$  un linguaggio qualsiasi.

$L' \in CO-NP \implies \bar{L}' \in NP$ .

Siccome  $L$  è NP-completo per ipotesi, abbiamo che  $\bar{L}' \leq_P L$ .

Equivalentemente a prima, consideriamo i complementi:  $L' \leq_P \bar{L}$ .

Secondo le considerazioni iniziali,  $\bar{L} \in NP \implies L' \in NP$ .

Vediamo ora come si relaziona  $P$  rispetto a queste due classi.

### Teorema

$$P \subseteq NP \cap CO-NP$$

### Dimostrazione

Sia  $L$  un generico linguaggio tale che  $L \in P$ .

Poiché  $P \subseteq NP \implies L \in NP$ .

Consideriamo  $\bar{L}$ .

Siccome  $L \in P \implies \exists$  una MT  $M$  deterministica che in tempo polinomiale decide  $L$ .

Se invertiamo  $M$ , essa deciderà  $\bar{L}$ .

Quindi  $\bar{L} \in P \implies \bar{L} \in NP \implies L \in CO-NP$ .

Non sappiamo se l'intersezione tra  $NP$  e  $CO-NP$  sia esattamente  $P$  oppure se  $P$  sia un sottoinsieme stretto di tale intersezione.

Semmai  $P = NP$ , allora anche  $CO-NP$  coinciderebbe con  $P$ . Non sapendolo, potrebbe accadere che  $NP = CO-NP$  e che  $P$  sia un sottoinsieme di queste due classi equivalenti.

## Linguaggi nel limbo

Attualmente non esiste alcun linguaggio NP-completo che appartenga a  $CO-NP$ , altrimenti avremmo che  $NP = CO-NP$ . Questo vuol dire che nell'intersezione tra  $NP$  e  $CO-NP$  esistono linguaggi che non sono NP-completi, pertanto non sono NP-ardui. Di tali linguaggi, quelli che non appartengono nemmeno a  $P$  compongono un limbo di problemi nè troppo difficili nè semplici.

Uno di questi è il problema della fattorizzazione: dato un numero intero, produrre una sua fattorizzazione in numeri primi. La variante decisionale a cui siamo interessati richiede di stabilire se esiste almeno un fattore primo minore di  $k$ .

$\text{FACTOR} = \{ \langle n, k \rangle \mid n \text{ è un intero che ha almeno un fattore primo } p \leq k \}$

Esempio di fattorizzazione

$$175 = 5 \cdot 5 \cdot 7$$

$\langle 175, 6 \rangle \in \text{FACTOR}$

$\langle 175, 4 \rangle \notin \text{FACTOR}$

Molti sistemi crittografici si basano su questo problema, ad esempio RSA.

### Teorema

$\text{FACTOR} \in NP \cap CO-NP$

#### Dimostrazione

$\text{FACTOR} \in NP$ )

Indoviniamo un fattore  $p \leq k$ , poi controlliamo che  $p$  sia primo e che  $p$  divida  $n$ .

$p$  è limitato da  $n$ , quindi è polinomiale nella taglia dell'input. La divisione tra  $n$  e  $p$  si può fare in tempo polinomiale nella taglia degli operandi. Il test di primalità su  $p$  si fa in tempo polinomiale deterministico (c'è un algoritmo).

Questo ci permette di dire che  $\text{FACTOR} \in NP$ .

$\text{FACTOR} \in CO-NP$ )

Lo verifichiamo mostrando che  $\overline{\text{FACTOR}} \in NP$ .

$\overline{\text{FACTOR}} = \{ \langle n, k \rangle \mid n \text{ è un intero i cui fattori primi sono tutti maggiori di } k \}$

Indoviniamo tutta la fattorizzazione  $p_1 \cdot \dots \cdot p_m$  di  $n$ , la quale è unica, poi controlliamo che

$p_1, \dots, p_m$  siano tutti primi, che moltiplicati tra loro diano  $n$  e che il più piccolo sia maggiore di  $k$ . Il numero di fattori indovinati è polinomiale nella taglia di  $n$ , poiché alla peggio è un esponente di 2 (il numero primo minimo). Il test di primalità si fa in tempo polinomiale, in quanto è una somma di polinomi (per l'algoritmo citato prima). La moltiplicazione si può fare in tempo polinomiale nella taglia degli operandi. Il controllo che il fattore più piccolo sia maggiore di  $k$  si fa in tempo polinomiale, scorrendoli tutti.

Questo ci dice che  $\overline{\text{FACTOR}} \in NP \implies \text{FACTOR} \in CO-NP$ .

Inoltre,  $\text{FACTOR} \notin P$  perché nessuno ha mai trovato un algoritmo polinomiale (non è detto che non esista).

## Classe EXP

$EXP$  è la classe dei linguaggi che possono essere decisi da una MT deterministica in tempo esponenziale.

$$EXP = \bigcup_{c \geq 1} \text{DTIME}(2^{nc})$$

Sappiamo che  $P \subset EXP$ . L'inclusione stretta è garantita dal teorema della gerarchia temporale, il quale afferma che se due classi di complessità temporale si differenziano per un fattore esponenziale, allora esse sono distinte.

Siccome possiamo simulare le MT non deterministiche in tempo esponenziale deterministico, la classe  $NP \subseteq EXP$ .

Tuttavia non sappiamo se sia un'inclusione stretta oppure se  $NP = EXP$ .

Similmente, anche  $CO-NP \subseteq EXP$ . Questo perché le classi deterministiche sono chiuse per complemento. Quindi se  $L \in CO-NP \implies \bar{L} \in NP \implies \bar{L} \in EXP \implies L \in EXP$ .

## Classe NEXP

$NEXP$  è la classe dei linguaggi che possono essere decisi da una MT non deterministica in tempo esponenziale. NEXP sta per non deterministic exponential time.

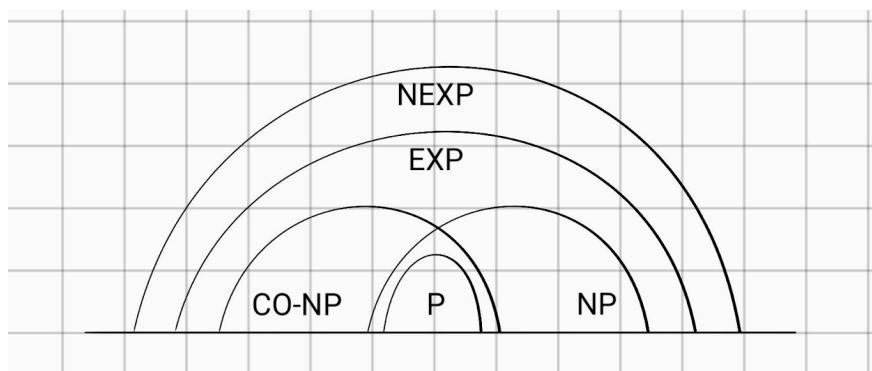
$$NEXP = \bigcup_{c \geq 1} \text{NTIME}(2^{nc})$$

Un linguaggio  $L \in NEXP$  se le sue istanze-sì sono caratterizzate da un certificato di taglia esponenziale, verificabile da una MT deterministica in tempo polinomiale nella taglia combinata dell'input e del certificato.

Sappiamo per certo che  $EXP \subseteq NEXP$ , tuttavia anche qui non sappiamo se l'inclusione è stretta oppure se  $EXP = NEXP$ .

Inoltre, per il teorema della gerarchia temporale, sappiamo che  $NP \subset NEXP$ .

## Classi di complessità temporale



# Complessità spaziale

Quando si parla di complessità, il fatto che ad essere vincolato sia il tempo oppure lo spazio fa la differenza. Questo perché il tempo vincola implicitamente anche lo spazio, mentre quest'ultimo può essere riutilizzato un numero arbitrario di volte.

## Macchine di Turing space-bounded

Per poter classificare classi di complessità spaziale piccole dobbiamo introdurre un modello raffinato di macchina di Turing, sempre equivalente alle MT precedenti. Questo nuovo modello non ha un nome in particolare ed è sempre un decisore.

La macchina in questione ha due nastri, su cui la testina si muove avanti e indietro:

- input → nastro di input, soltanto lettura
- work-tape → nastro di lavoro, lettura e scrittura

Ci serve questo modello perché il vincolo sullo spazio riguarderà solo il work-tape, altrimenti la taglia dell'input sovrasterebbe il limite.

Sia  $M$  una MT space-bounded.

Il **computation space** della macchina  $M$  su input  $w$  è il numero di celle distinte viste da  $M$  sul work-tape mentre processa  $w$ . Per una macchina non deterministica è il massimo computation space tra tutti i rami di computazione.

Sia  $s : \mathbb{N} \rightarrow \mathbb{N}$  una funzione.  $s$  è una **space function** se è strettamente positiva e non decrescente.

Sia  $s(n)$  una space function.

Il **running space** della macchina  $M$  è  $s(n)$  se per ogni stringa  $w$ , eccetto un numero finito, il computation space di  $M$  su  $w$  è limitato da  $s(\|w\|)$ .

## Classi di complessità spaziale

Sia  $s(n)$  una space function.

Definiamo la classe **DSPACE** come l'insieme di tutti i linguaggi decisi da una MT deterministica avente running space  $O(s(n))$ .

$$\text{DSPACE}(s(n)) = \{ L \mid \exists M \text{ deterministica che decide } L \text{ in spazio } O(s(n)) \}$$

Sia  $s(n)$  una space function.

Definiamo la classe **NSPACE** come l'insieme di tutti i linguaggi decisi da una MT non deterministica avente running space  $O(s(n))$ .

$$\text{NSPACE}(s(n)) = \{ L \mid \exists N \text{ non deterministica che decide } L \text{ in spazio } O(s(n)) \}$$

$L$ , anche detta *LOGSPACE*, è la classe dei linguaggi che possono essere decisi da una MT deterministica in spazio logaritmico.

$$L = \text{DSPACE}(\log_2 n)$$

$NL$ , anche detta *Non deterministic LOGSPACE*, è la classe dei linguaggi che possono essere decisi da una MT non deterministica in spazio logaritmico.

$$NL = \text{NSPACE}(\log_2 n)$$

La relazione che lega queste due classi è  $L \subseteq NL$ . Anche in questo caso non sappiamo se  $L$  è inclusa strettamente in  $NL$  oppure se  $L = NL$ .

D'ora in poi con  $\log n$  faremo sempre riferimento al logaritmo in base 2.

Esempio

Consideriamo il linguaggio  $L = \{ 0^n 1^n \mid n > 0 \}$ .

Usare la strategia formulata inizialmente richiederebbe spazio lineare (andare in cima e in fondo cancellando man mano i caratteri).

Possiamo fare meglio: sul work-tape contiamo in binario gli 0 commando, poi sottraendo contiamo gli 1. In questo modo la rappresentazione binaria prende spazio logaritmico, quindi  $L \in LOGSPACE$ .

Vediamo adesso un problema appartenente ad  $NL$ : raggiungibilità sul grafo.

$\text{REACH} = \{ \langle G, s, t \rangle \mid G = \langle V, E \rangle \text{ è un grafo diretto,}$

$s, t \in V, \exists \text{ un percorso da } s \text{ a } t \text{ in } G \}$

Normalmente lo risolveremmo tramite BFS o DFS, i quali richiedono spazio polinomiale.

Possiamo utilizzare un algoritmo più "economico": disponendo di una MT non deterministica, indoviniamo gradualmente il passo successivo nel percorso, poi controlliamo che l'arco esista e che non comporti cicli. In questo modo, dobbiamo memorizzare solo il nodo attuale ed il nodo seguente. Per garantire l'assenza di cicli, ci serve in memoria un contatore di nodi visitati. Se tale contatore supera il numero totale di nodi nel grafo si rifiuta la computazione a causa di un loop.

```
REACH(G = <V,E>, s, t):
    p = s
    n = 1
    (a) if (p == t) then ACCEPT
        p' = GUESS di un nodo in G
        if (<p,p'> non è in E) then REJECT
        p = p'
        n = n + 1
        if (n <= |V|) then GOTO (a) else REJECT
```

Sul work-tape quindi memorizziamo il nodo attuale, il nodo seguente ed il contatore, ognuno dei quali prende spazio logaritmico nella taglia dell'input. Segue che  $\text{REACH} \in NL$ .

# Teoria della complessità spaziale

Nell'oscurità delle nostre conoscenze, è stata formulata una nozione di NL-completezza. Prima di arrivarci, dobbiamo definire il concetto di riduzione vincolata sul piano spaziale.

## Riduzioni LOGSPACE

Intuitivamente, sono riduzioni in cui la funzione  $f$  è calcolabile da un trasduttore avente spazio logaritmico sul work-tape. Lo spazio sul nastro di output non è vincolato poiché, essendo di sola scrittura, il suo contenuto non viene considerato nella valutazione del next step da parte della funzione di transizione.

Siano  $A$  e  $B$  due linguaggi.

Esiste una **riduzione LOGSPACE** da  $A$  a  $B$ , denotata con  $A \leq_L B$ , se  $\exists$  una funzione  $f : \Sigma^* \rightarrow \Sigma^*$  tale che:

- per ogni stringa  $w$ ,  $w \in A \iff f(w) \in B$
- $f$  è calcolabile da un trasduttore in spazio logaritmico

Tutti i risultati validi per le riduzioni polinomiali sono validi anche per le riduzioni LOGSPACE.

## NL-completeness

Un linguaggio  $L$  è NL-completo se:

- $L \in NL \rightarrow$  membership
- $\forall L' \in NL \exists$  una riduzione  $L' \leq_L L \rightarrow$  NL-hardness

Esiste un risultato, simile a quanto dimostrato per  $NP$ , che afferma quanto segue:

Sia  $\tilde{L}$  un linguaggio NL-completo.  $\tilde{L} \in L \iff L = NL$ .

## REACH

Dimostriamo che REACH è NL-completo facendo uso della definizione.

Sia  $L$  un generico linguaggio di  $NL$ .

	$L$	$\leq_L$	REACH
<b>Input</b>	Una stringa $w$	$\rightsquigarrow$	Una tripla grafo, nodo sorgente, nodo destinazione $(G, s, t)$
<b>Istanza-sì</b>	$w \in L$		$\exists$ in $G$ un percorso da $s$ a $t$
<b>Istanza-no</b>	$w \notin L$		$\nexists$ in $G$ un percorso da $s$ a $t$

Inizialmente sappiamo solo che  $L \in NL$ , perciò  $\exists$  una MT  $M$  che decide  $L$  in spazio logaritmico.

Il grafo  $G$  che costruiamo avrà un nodo per ogni configurazione di  $M$  nella computazione su  $w$ . Gli archi conserveranno ciascun nodo ai suoi legal successor. I nodi sorgente e destinazione saranno rispettivamente la configurazione iniziale e finale. In questo modo esisterà un percorso solo se la computazione è accettante.

La funzione  $f$  deve generare tanti nodi quante sono le possibili configurazioni della MT  $M$ . Ogni nodo è identificato da una label che descrive la configurazione tramite i campi di interesse (non ci serve memorizzare il contenuto del nastro di input poiché esso è immutabile):

- Contenuto del work-tape  $\rightarrow$  per definizione di  $M$  prende spazio  $O(\log n)$ , dove  $n = \|w\|$
  - Posizione della testina sul nastro di input,  $h_1 \rightarrow$  serve spazio  $O(\log n)$  poiché il nastro è lungo  $n$
  - Posizione della testina sul work-tape,  $h_2 \rightarrow$  serve spazio  $O(\log(\log n))$  poiché il work-tape è  $O(\log n)$
  - Stato in cui si trova la macchina,  $q \rightarrow$  il numero di stati è fissato, quindi prende spazio costante
- In totale lo spazio di cui necessitiamo per memorizzare una singola label è  $O(\log n)$ .

Non possiamo mantenere sul work-tape il grafo  $G$  interamente, in quanto avremmo bisogno di spazio polinomiale.

Per questo motivo, la funzione  $f$  scrive una label alla volta sul work-tape, controlla se è sensata (potrebbe essere inconsistente), se lo è la scrive in output altrimenti la scarta. Fatto ciò, sul nastro di output avremo tutti e soli i nodi di  $G$ . A questi nodi si aggiunge  $v^*$ , che sarà il nodo destinazione  $t$ . Il nodo sorgente  $s$  sarà la configurazione iniziale.

Per definire gli archi, ripartiamo dai nodi generati considerandoli a coppie. Per ogni coppia, se il secondo nodo è un legal successor del primo scriviamo la coppia in output, altrimenti passiamo alla successiva.

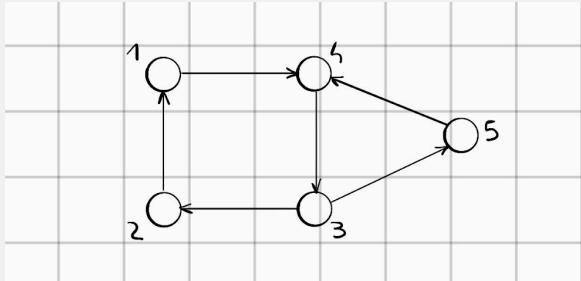
Come ultima fase, esploriamo tutti i nodi e connettiamo le configurazioni accettanti di  $M$  a  $v^*$ .

Per come è costruito  $G$ , al suo interno  $\exists$  un percorso da  $s$  a  $t \iff M \models w$ .

Siccome ogni linguaggio  $L \in NL$  è riducibile LOGSPACE a REACH  $\implies$  REACH è NL-arduo.  
 $REACH \in NL \wedge$  REACH è NL-arduo  $\implies$  REACH è NL-completo.

Sebbene non siamo in grado di dimostrare che  $REACH \in L$ , possiamo mostrare che  $REACH \in \text{DSPACE}(\log^2 n)$ , ovvero che REACH è decidibile in spazio polilogaritmico.

Focalizziamoci su un esempio.



$$s = 2, t = 5$$

In particolare, questa è un'istanza-sì:  $s \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow t$

Osserviamo che se esiste un percorso per andare da  $s$  a  $t$ , allora deve esistere un nodo intermedio  $u$  tale che esistano due percorsi  $s \rightarrow u$  e  $u \rightarrow t$ .

Applichiamo ciò al nostro caso: per verificare l'esistenza di un percorso da  $s$  a  $t$ , troviamo un nodo  $u$  a metà strada e poi controlliamo che ci siano due percorsi  $s \rightarrow u$  e  $u \rightarrow t$ .

Questo approccio è di tipo ricorsivo:

- caso base  $s = t \implies$  rispondiamo sì
- caso base  $t$  è un successivo di  $s \implies$  rispondiamo sì
- caso induttivo  $\implies$  proviamo tutti i nodi  $u$  per vedere se esistono i percorsi  $s \rightarrow u$  e  $u \rightarrow t$

Definiamo quindi il seguente algoritmo `exists-path`, che ritorna true se in  $G$  esiste un percorso  $s \rightarrow t$  di lunghezza al più  $k$ .

```
exists-path(G = <V, E>, s, t, k):  
    if (k == 0) {  
        if (s == t) then ACCEPT else REJECT  
    }  
    if (k == 1) {  
        if (<s,t> è in E) then ACCEPT else REJECT  
    }  
    for-each (u in V) {  
        if (exists-path(G,s,u,k/2) AND exists-path(G,u,t,k/2)) then ACCEPT  
    }  
    REJECT
```

La prima chiamata è `exists-path(G, s, t, |E|)`. Non c'è non-determinismo in quanto non stiamo indovinando, stiamo provando tutti i nodi.

Valutiamo lo spazio di cui necessita tale algoritmo. Sostanzialmente, ci servono due aree per memorizzare  $s$  e  $t$ , entrambe di taglia logaritmica, più una per sapere chi è il nodo  $u$  attuale, sempre di taglia logaritmica. In totale ci serve spazio  $O(\log n)$  per una singola esecuzione.

Siccome l'algoritmo è ricorsivo, dobbiamo considerare anche lo stack di chiamate. Se ci troviamo nel caso induttivo, per ogni nodo  $u$  effettuiamo due chiamate, le quali non vengono eseguite contemporaneamente. Quindi, siccome ad ogni chiamata dimezziamo  $k$ , nello scenario peggiore in memoria avremo  $\log |E|$  chiamate.

In conclusione, la complessità spaziale di `exists-path` è  $O(\log |E|) \cdot O(\log n) = O(\log^2 n)$  poiché  $E$  è parte dell'input.

## Teorema di Savitch

Vediamo un risultato molto importante che caratterizza le classi di complessità spaziale, chiamato teorema di Savitch. Questo teorema afferma che: tutto ciò che è possibile fare in spazio  $s(n)$  non deterministico, lo si può fare in spazio  $s(n)^2$  deterministico. Quindi, a differenza delle classi temporali, dove il passaggio da non deterministico a deterministico comporta un salto esponenziale, quando lavoriamo con classi spaziali il salto è solo quadratico.

### Teorema

Sia  $s(n)$  una space function tale che  $s(n) \in \Omega(\log n)$ .

Allora  $\text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s(n)^2)$

### Dimostrazione

Sia  $L \in \text{NSPACE}(s(n))$ .

Questo vuol dire che  $\exists$  una MT  $M$  non deterministica che decide  $L$ , la quale ha running space  $s(n)$ .

Analizziamo il computation graph di  $M$ , ovvero un grafo avente come nodi le configurazioni di  $M$  ed archi che connettono i legal successor. Dobbiamo assicurarci che la taglia di tale grafo non ecceda i limiti. Sappiamo che la label di un nodo è formata da:

- work-tape  $\rightarrow$  per definizione di  $M$  prende spazio  $O(s(n))$
- testina sull'input,  $h_1 \rightarrow$  serve spazio  $O(\log n)$  poiché il nastro è lungo  $n$
- testina sul work-tape,  $h_2 \rightarrow$  serve spazio  $O(\log s(n))$  poiché il work-tape è  $O(s(n))$
- stato della macchina,  $q \rightarrow$  prende spazio costante poiché non dipende dall'input

Siccome per ipotesi  $s(n)$  è almeno  $\log n$ , una singola label richiede spazio  $O(s(n))$ .

Sia  $l = |\Gamma|$ . Il numero di nodi del grafo è  $O(l^{s(n)})$ . Il numero di archi del grafo è al più quadratico nel numero di nodi, quindi  $O(l^{2 \cdot s(n)})$ .

Complessivamente, la taglia del computation graph è  $O(l^{2 \cdot s(n)})$ .

Su questo grafo lanciamo l'algoritmo deterministico `exists-path` e verifichiamo se esiste un percorso dalla configurazione iniziale ad una configurazione accettante. Tale procedura richiede spazio  $O(\log^2 m)$ , dove  $m$  è la taglia del grafo.

Mettendo insieme i pezzi, lo spazio necessario per decidere deterministicamente  $L$  risulta essere  $O((\log l^{2 \cdot s(n)})^2) = O((2 \cdot s(n) \cdot \underbrace{\log l}_{\text{cost.}})^2) = O(s(n)^2)$ .

Segue che  $L \in \text{DSPACE}(s(n)^2) \implies \text{NSPACE}(s(n)) \subseteq \text{DSPACE}(s(n)^2)$ .

## Classi polinomiali

A questo punto possiamo definire  $PSPACE$  ed  $NPSPACE$ .

$PSPACE$  è la classe dei linguaggi che possono essere decisi da una MT deterministica in spazio polinomiale, con un certo esponente fissato.

$$PSPACE = \bigcup_{c \geq 1} \text{DSPACE}(n^c)$$

$NPSPACE$  è la classe dei linguaggi che possono essere decisi da una MT non deterministica in spazio polinomiale, con un certo esponente fissato.

$$NPSPACE = \bigcup_{c \geq 1} \text{NSPACE}(n^c)$$

La relazione che lega queste due classi è  $PSPACE \subseteq NPSPACE$ , ma per il teorema di Savitch esse sono equivalenti  $\rightarrow PSPACE = NPSPACE$ .

Il teorema di Savitch è interessante proprio per questo: per uno stesso ordine di grandezza, da polinomiale a crescere, le classi spaziale deterministica e spaziale non deterministica sono equivalenti.

L'intuizione alla base è che il tempo una volta passato non lo possiamo riavere, mentre lo spazio lo possiamo riutilizzare. Quindi simulando una macchina non deterministica tramite una deterministica, lo spazio utilizzato resta lo stesso.

## Relazioni tra spazio e tempo

### $LOGSPACE$ e $PTIME$

Supponiamo di avere una MT deterministica  $M$  che decide un linguaggio in spazio logaritmico. Il numero di possibili configurazioni di una MT è  $O(2^m)$ , dove  $m$  è la taglia di una configurazione. Nel caso di una macchina in  $LOGSPACE$  diventa  $2^{\log m} = O(m)$ , ovvero una quantità polinomiale.

Sappiamo che  $M$  si arresta sempre, quindi non passa mai due volte per la stessa configurazione, altrimenti ci sarebbe un loop.

Perciò, attraversando un numero di configurazioni al più polinomiale,  $M$  avrà a disposizione tempo polinomiale, da cui  $LOGSPACE \subseteq PTIME$ .

### $NLOGSPACE$ e $PTIME$

Sappiamo che il problema REACH, appartenente a  $NLOGSPACE$ , è risolvibile in tempo polinomiale tramite BFS. Siccome REACH è NL-arduo, tutti i problemi di  $NLOGSPACE$  possono essere ridotti ad esso in tempo polinomiale, da cui  $NLOGSPACE \subseteq PTIME$ .

## $NPTIME$ e $NPSPACE = PSPACE$

Supponiamo di avere una MT non deterministica che decide un linguaggio in tempo polinomiale. Durante la computazione, tale macchina non può visitare una quantità di celle più grande di polinomiale, pertanto  $NPTIME \subseteq NPSPACE = PSPACE$ .

## $CO-NPTIME$ e $PSPACE$

Sia  $L$  un linguaggio tale che  $L \in CO-NPTIME$ .

Per definizione,  $\bar{L} \in NPTIME \implies \bar{L} \in PSPACE$ .

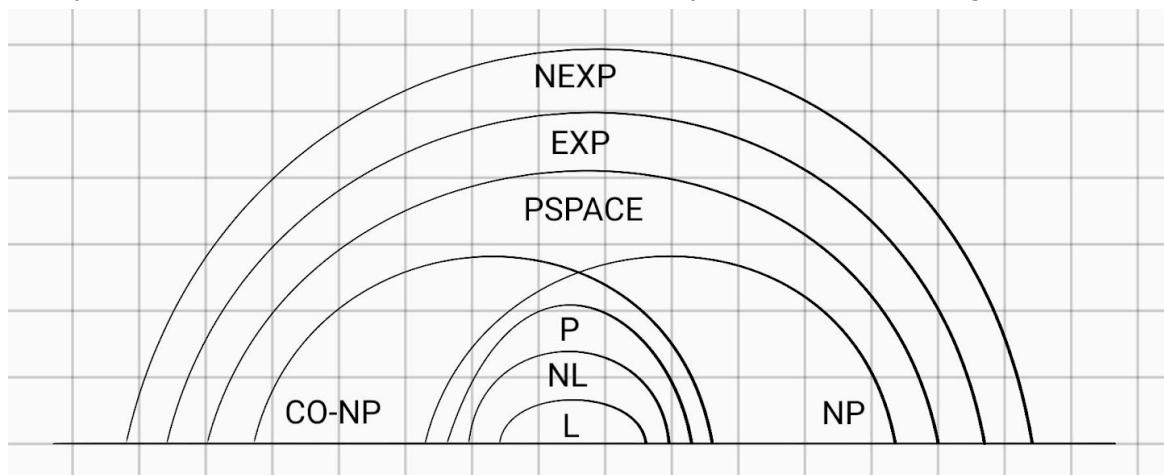
Siccome  $PSPACE$  è una classe deterministica, è chiusa per complemento, perciò anche  $L \in PSPACE$ , da cui  $CO-NPTIME \subseteq PSPACE$ .

## $PSPACE$ , $EXPTIME$ e $NEXPTIME$

Per lo stesso ragionamento affrontato prima ( $2^{\text{conf}} \dots$ ), una MT in  $PSPACE$  avrà a disposizione tempo esponenziale, da cui  $PSPACE \subseteq EXPTIME \subseteq NEXPTIME$ .

## Schema finale delle classi

Complessivamente, lo schema delle classi di complessità diventa il seguente.



All'interno della classe  $P$  esistono dei problemi chiamati  $P$ -completi, ovvero tra i più difficili di  $P$ . Riteniamo che essi non siano in  $LOGSPACE$ , ma nessuno è mai riuscito a dimostrarlo. I problemi  $P$ -completi sono inerentemente sequenziali, nel senso di non-parallelizzabili, mentre i problemi in  $LOGSPACE$  sono parallelizzabili.

# Oracoli

In certe situazioni ci troviamo di fronte a problemi per cui una singola verifica non è sufficiente: non basta indovinare una soluzione → è necessario compiere più passi logici.

Prendiamo come esempio il problema del Vertex Cover: dato un grafo  $G$ , stabilire se al suo interno esiste un VC di taglia al più  $k$ .

$$\text{VC} = \{ \langle G, k \rangle \mid \exists \text{ un VC di taglia al più } k \text{ in } G \}$$

Consideriamo ora una variante chiamata MIN-COVER, in cui vogliamo stabilire se la taglia minima di un VC in  $G$  sia esattamente  $k$ .

$$\text{MIN-COVER} = \{ \langle G, k \rangle \mid \text{i VC di taglia minima in } G \text{ hanno taglia } k \}$$

In questo caso non è sufficiente indovinare un VC di taglia  $k$ , dobbiamo assicurarsi che non ne esista uno di taglia inferiore. In altre parole, la verifica di un'istanza di MIN-COVER richiede due controlli: che esista un VC di taglia  $k$  e che non esista un VC di taglia  $k - 1$ . Possiamo quindi riformulare il problema come segue.

$$\text{MIN-COVER} = \{ \langle G, k \rangle \mid \langle G, k \rangle \in \text{VC} \wedge \langle G, k - 1 \rangle \notin \text{VC} \}$$

Se avessimo la possibilità di chiamare un decisore per VC, potremmo risolvere MIN-COVER agilmente.

## Macchine di Turing ad oracolo

Introduciamo a questo scopo le macchine di Turing ad oracolo, le quali simulano il fatto che una MT possa chiamare una subroutine. Intuitivamente, la macchina chiamante chiede all'oracolo di decidere una certa istanza, poi in base alla risposta prosegue la sua computazione. La macchina chiamante può essere sia deterministica che non deterministica.

Sostanzialmente, un oracolo è un decisore di un linguaggio. Una MT ad oracolo dispone di un nastro di sola scrittura, che chiamiamo oracle-tape, su cui il chiamante scrive la domanda da porre all'oracolo. Ciò che viene scritto sul nastro è quindi un'istanza del linguaggio di cui l'oracolo è decisore, e la risposta ci dirà se è istanza-sì o istanza-no.

Denotiamo una generica MT ad oracolo con  $M^?$ , perché la funzione di transizione di  $M$  è indipendente dal comportamento dell'oracolo. Denotiamo con  $M^L$  una MT che ha come oracolo un decisore per il linguaggio  $L$ .

Una MT ad oracolo ha 3 stati particolari per effettuare le domande:  $q_?$ ,  $q_{yes}$  e  $q_{no}$ .

Il chiamante scrive sull'oracle-tape la domanda, poi si sposta nello stato  $q_?$ . A questo punto, l'oracolo risponde ed il chiamante si muove sullo stato  $q_{yes}$  oppure  $q_{no}$  in base alla risposta dell'oracolo. Il procedimento di risposta è un'unica transizione, quindi non c'è perdita di tempo. Assumiamo che l'oracle-tape si svuoti automaticamente nel momento in cui riceviamo risposta.

# Classi ad oracolo

Con questa nozione possiamo definire una serie di classi ad oracolo.

Sia  $C$  una generica classe di complessità.

Definiamo con  $P^C$  l'insieme dei linguaggi che possono essere decisi in tempo polinomiale da una MT ad oracolo deterministica che interroga un oracolo per un linguaggio  $L \in C$ .

Sia  $C$  una generica classe di complessità.

Similmente, definiamo  $NP^C$  come l'insieme dei linguaggi che possono essere decisi in tempo polinomiale da una MT ad oracolo non deterministica che interroga un oracolo per un linguaggio  $L \in C$ .

Intuitivamente, all'interno di queste due classi possiamo effettuare una quantità polinomiale di domande all'oracolo.

MIN-COVER  $\in P^{NP}$ , poiché è decidibile in tempo polinomiale facendo due domande ad un decisore per  $VC \in NP$ .

$NP \subseteq P^{NP}$

Sia  $L \in NP$ . Possiamo costruire una MT ad oracolo che prende un'istanza di  $L$ , la passa all'oracolo che decide  $L$ , la cui risposta sarà la risposta della macchina.

Tale macchina decide  $L \implies L \in P^{NP} \implies NP \subseteq P^{NP}$ .

$CO-NP \subseteq P^{NP}$

Sia  $L \in CO-NP \implies \bar{L} \in NP$ . Possiamo costruire una MT ad oracolo  $M^{\bar{L}}$ , che prende un'istanza di  $L$ , la passa all'oracolo che decide  $\bar{L}$ , poi risponde l'inverso dell'oracolo.

Tale macchina decide  $L \implies L \in P^{NP} \implies CO-NP \subseteq P^{NP}$ .

## Gerarchia polinomiale

Il fatto che possano esistere MT aventi come oracolo una macchina che a sua volta fa uso di oracoli, e così via, dà origine alla gerarchia polinomiale.

La gerarchia polinomiale è un insieme di classi di complessità che si colloca al di sopra di  $NP$ .

Le classi che fanno parte della gerarchia polinomiale si chiamano  $\Sigma_i^P$ ,  $\Pi_i^P$  e  $\Delta_i^P$ , con  $i \geq 0$ , e sono così definite:

- $\Sigma_0^P = P$ , mentre per  $i \geq 1$   $\Sigma_i^P = NP^{\Sigma_{i-1}^P}$
- per  $i \geq 0$   $\Pi_i^P = CO-\Sigma_i^P$ , ovvero i linguaggi complemento di quelli in  $\Sigma_i^P$
- per  $i \geq 0$   $\Delta_i^P = P^{\Sigma_i^P}$

$\Delta_2^P = P^{NP}$

Quindi MIN-COVER  $\in \Delta_2^P$

Praticamente la  $i$  rappresenta il numero di classi in catena:  $\Sigma$  ha come base  $NP$ ,  $\Pi$  sono i complementi di sigma,  $\Delta$  ha come base  $P$  (tutti "elevati" alla  $NP$ ).

### Relazioni tra le classi della gerarchia polinomiale

$$\Sigma_i^P \subseteq \Delta_{i+1}^P = P^{\Sigma_i^P}$$

Dimostrazione simile a  $NP \subseteq P^{NP}$ .

$$\Pi_i^P \subseteq \Delta_{i+1}^P = P^{\Sigma_i^P}$$

Dimostrazione simile a  $CO-NP \subseteq P^{NP}$ .

$$\Delta_i^P \subseteq \Sigma_i^P$$

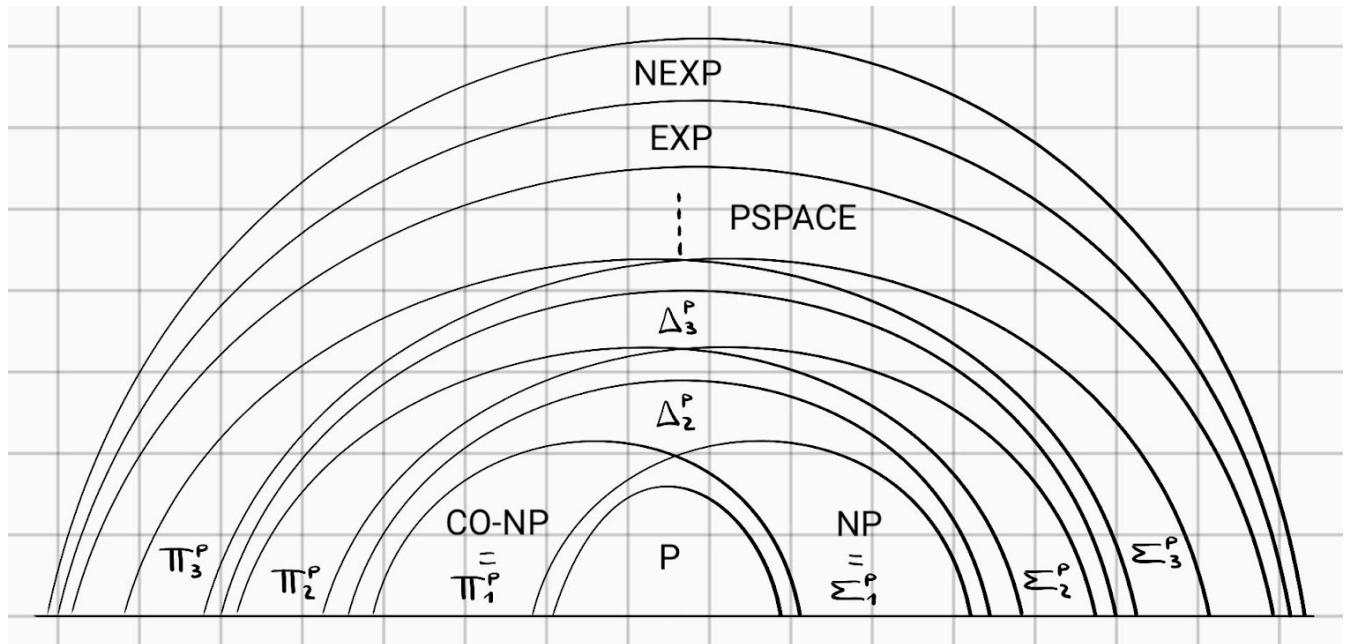
Entrambe hanno oracoli in  $\Sigma_{i-1}^P$ , quindi la classe deterministica è contenuta in quella non deterministica.

$$\Delta_i^P \subseteq \Pi_i^P$$

Sia  $L \in \Delta_i^P = P^{\Sigma_{i-1}^P}$ . Allora  $\exists$  una MT ad oracolo  $M^?$  deterministica che decide  $L$  in tempo polinomiale interrogando un oracolo in  $\Sigma_{i-1}^P$ . Quindi invertendo gli stati di accettazione e rifiuto otteniamo una MT  $\bar{M}^?$ , con le stesse caratteristiche, che decide  $\bar{L}$ .

Questo significa che  $\bar{L} \in P^{\Sigma_{i-1}^P} = \Delta_i^P$ .

Sappiamo che  $\Delta_i^P \subseteq \Sigma_i^P \implies L \in \Pi_i^P$ .



Allo stato attuale delle conoscenze, riteniamo che ci siano infiniti livelli nella gerarchia polinomiale, tuttavia non è mai stato dimostrato.

Ciò che è stato dimostrato è che la gerarchia polinomiale  $\subseteq PSPACE$ .

Intuitivamente, i problemi dentro alla gerarchia polinomiale sono i problemi dei giochi.

Il problema SAT che abbiamo affrontato fin'ora può essere formulato come: esiste un assegnamento di verità  $\bar{x}$  che soddisfi la formula  $\phi$ , ovvero  $\exists \bar{x} \mid \phi(\bar{x}) = \text{true}$ . Per questa ragione lo chiamiamo  $\exists$ -SAT.

Allo stesso modo possiamo definire  $\exists\forall$ -SAT, in cui l'assegnamento di verità  $\bar{x}$  deve soddisfare  $\phi$  per ogni valore delle variabili  $y$ .

Si può dimostrare che  $\exists\forall$ -SAT è  $\Sigma_2^P$ -completo.

$\exists\forall\exists$ -SAT è  $\Sigma_3^P$ -completo e così via.

La metafora dei giochi sarebbe: è vero che esiste una mia mossa tale che per ogni mossa dell'avversario esiste una mia mossa tale che ... e alla fine vinco?

## Problemi di ottimizzazione e classi funzionali

Definiamo la **classe funzionale  $FP$**  come l'insieme delle funzioni calcolate da trasduttori deterministici in tempo polinomiale. In sostanza, sono le funzioni per le quali abbiamo algoritmi deterministici.

Consideriamo il problema  $FMIN-COVER(G) = \min\{ |V| \mid$

$V$  è un Vertex Cover di  $G \}$ . Vogliamo trovare la taglia del VC più piccolo: non è un problema di decisione, bensì di calcolo.

Supponiamo di volerlo risolvere, disponendo di un oracolo che decide Vertex Cover.

Per ogni  $k$  da 1 a crescere, chiediamo all'oracolo se esiste un VC di quella dimensione. Il primo  $k$  per cui l'oracolo risponde sì è la taglia minima del VC.

Possiamo quindi dire che  $FMIN-COVER \in FP^{NP}$ , perché un trasduttore, avendo a disposizione un decisore per VC, può calcolare l'algoritmo in tempo polinomiale (alla peggio valutiamo  $k$  fino al numero di nodi nel grafo).

L'algoritmo è ottimizzabile usando la ricerca binaria, in questo modo il chiamante pone un numero logaritmico di domande all'oracolo. Denotiamo ciò con  $FP^{NP}[O(\log n)]$

Qui viene fuori il legame tra problemi di decisione e problemi di ricerca. Se fossimo in grado di calcolare la taglia minima del VC in tempo polinomiale deterministico, allora anche Vertex Cover decisionale sarebbe decidibile in tempo polinomiale deterministico, ergo  $VC \in P$ .

Ma VC è NP-completo, quindi sarebbe  $P = NP$ . Per questo motivo, la variante di ricerca di un problema è almeno difficile quanto la sua versione decisionale.

## Traveling Sales Person (TSP, commesso viaggiatore)

Prima di affrontare il problema in questione ci servono due definizioni.

Un **grafo pesato**  $G$  è una tripla  $\langle V, E, \lambda \rangle$ , composta da nodi, archi ed una funzione di peso.

La funzione  $\lambda : E \rightarrow \mathbb{N}$  mappa ogni arco ad un intero che corrisponde al suo peso.

Un **Hamiltonian cycle** è un percorso nel grafo che parte ed arriva allo stesso nodo, passando esattamente una volta su tutti i nodi del grafo. Il peso di un HC è la somma dei pesi degli archi traversati.

Dato un grafo non orientato, vogliamo calcolare il peso dell'HC di peso minimo.

$$FTSP = \min\{ \lambda(\pi) \mid \pi \text{ è un Hamiltonian Cycle di } G = \langle V, E, \lambda \rangle \}$$

Supponiamo di avere un oracolo per la variante decisionale del problema, che dato un grafo non orientato, sia in grado di stabilire se vi sia un Hamiltonian Cycle di peso al più  $k$ .

$$TSP = \{ \langle G, k \rangle \mid G = \langle V, E, \lambda \rangle \text{ ammette un HC di peso al più } k \}$$

Per risolvere FTSP potremmo chiedere all'oracolo se esista un HC di peso  $k$  in  $G$ , per  $k$  che va da 0 alla somma totale dei pesi del grafo. Il primo  $k$  per cui l'oracolo risponde sì è la soluzione, altrimenti l'HC non esiste (se non risponde mai sì).

Tramite questa procedura poniamo all'oracolo una quantità di domande proporzionale al valore della somma dei pesi, il quale è esponenziale nella taglia della rappresentazione. Facendo un numero esponenziale di domande, sarebbe  $FTSP \in FEXP^{NP}$ , ma a questo punto non ci servirebbe l'oracolo.

Necessitiamo quindi di una risoluzione più economica: usiamo la ricerca binaria. In questo modo poniamo all'oracolo una quantità di domande logaritmica rispetto allo spazio del dominio, il quale è esponenziale nella taglia dell'input. Segue che la quantità complessiva è polinomiale, quindi  $FTSP \in FP^{NP}$ .

Riteniamo che non sia possibile risolvere FTSP in tempo polinomiale deterministico senza fare uso di oracoli, poiché TSP decisionale è NP-completo. Infatti, se fosse possibile, come detto per VC, avremmo  $P = NP$ .

Dimostriamo quindi la NP-completezza di TSP.

Tramite una MT non deterministica indoviniamo un Hamiltonian Cycle in  $G$ , la cui taglia è necessariamente polinomiale poiché composto da tutti e soli i nodi dell'input. Controlliamo che tutti i nodi siano collegati, che sorgente e destinazione coincidano, che il peso complessivo non ecceda  $k$  e che siano visitati tutti i nodi una volta sola. Tutto ciò è fattibile in tempo polinomiale, pertanto  $TSP \in NP$ .

Per dimostrare che TSP è NP-arduo dobbiamo fare alcuni passaggi intermedi, componendo una catena di riduzioni:  $3SAT \leq_P DHC \leq_P HC \leq_P TSP$ .

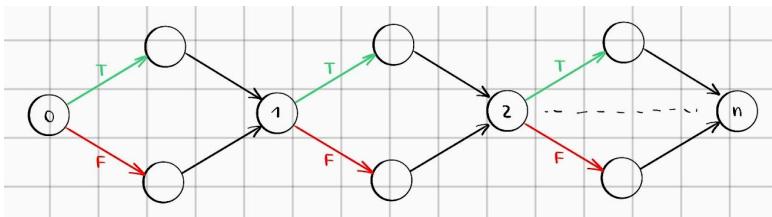
## Prima riduzione: 3SAT $\leq_P$ DHC

Il problema DHC, Directed Hamiltonian Cycle, richiede di stabilire se all'interno di un grafo orientato è presente un HC.

$$\text{DHC} = \{ G \mid G \text{ è un grafo orientato che ammette un HC} \}$$

	<b>3SAT</b>	$\leq_P$	<b>DHC</b>
<b>Input</b>	Una formula booleana $\phi$ in 3CNF	$\rightsquigarrow$	Un grafo orientato $G$
<b>Istanza-sì</b>	$\phi$ è soddisfacibile		$G$ ammette un HC
<b>Istanza-no</b>	$\phi$ non è soddisfacibile		$G$ non ammette un HC

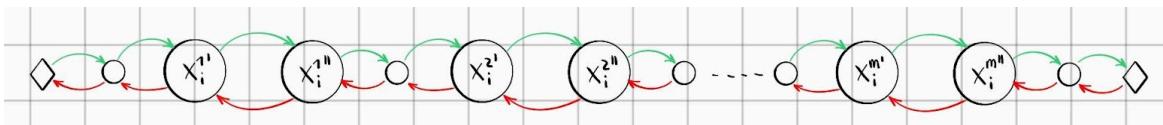
Per una formula  $\phi$  avente  $n$  variabili, il numero di possibili assegnamenti di verità è  $2^n$ . Per poter simulare una quantità esponenziale di percorsi, abbiamo bisogno di un grafo in cui ogni nodo ha due archi uscenti per raggiungere il successivo. Il nostro obiettivo è mappare l'assegnamento true o false per una variabile alla scelta di uno o l'altro arco per un nodo.



Sia  $\phi$  una formula con  $m$  clausole  $c_1 \wedge \dots \wedge c_m$ , ognuna delle quali contiene 3 variabili su  $n$  totali,  $\bar{x} = \{x_1, \dots, x_n\}$ . Assumiamo che in una clausola non appaia mai un letterale ed il suo negato, altrimenti sarebbe sempre verificata.

Per ogni variabile  $x_i$  di  $\phi$  costruiamo una catena di nodi:

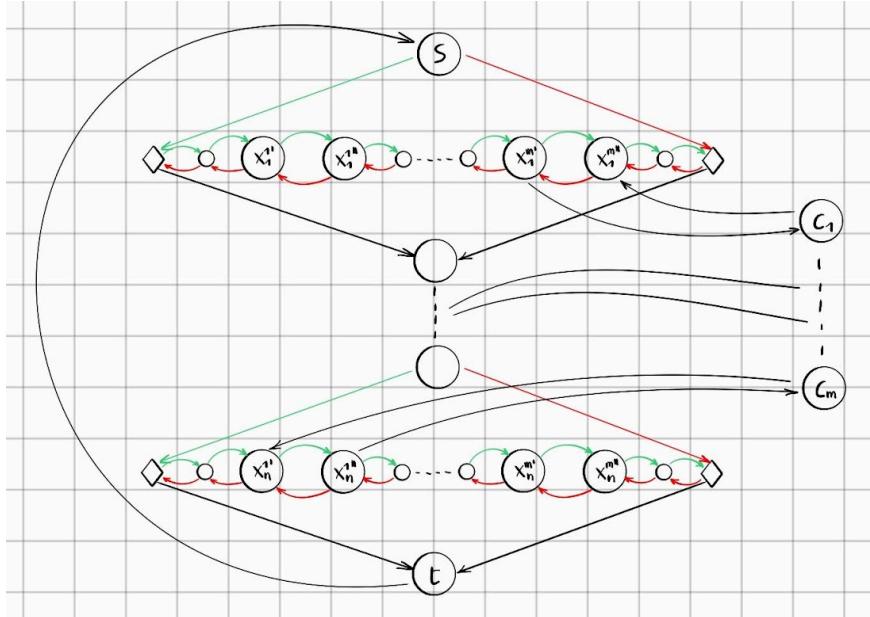
- generiamo una coppia di nodi per ogni clausola, quindi  $m$  coppie
- aggiungiamo dei nodi intermedi prima e dopo le coppie (tondini)
- inseriamo un nodo speciale all'inizio e alla fine della catena (diamanti)
- colleghiamo con archi verdi i nodi dal primo all'ultimo, poi con archi rossi dall'ultimo al primo (i colori servono solo visivamente)



A questo punto costruiamo il grafo  $G$ . Abbiamo un nodo di partenza  $s$ , collegato con un arco verde ed uno rosso rispettivamente ai diamanti iniziale e finale della catena di  $x_1$ . Dai due diamanti ci ricongiungiamo ad un nodo intermedio, da cui ripetiamo il procedimento per ogni catena  $x_i$  fino ad arrivare ad un nodo finale  $t$ . Infine colleghiamo il nodo  $t$  al nodo  $s$ .

Ci serve ancora un pezzo per legare la soddisfacibilità alla presenza dell'HC.

Aggiungiamo lateralmente  $m$  nodi, uno per clausola. Se la variabile  $x_i$  appare come letterale positivo nella clausola  $c_j$ , prendiamo la  $j$ -esima coppia della  $i$ -esima catena e connettiamo il nodo primo al nodo  $c_j$  e il nodo  $c_j$  al nodo secondo. Se la variabile  $x_i$  appare come letterale negativo in  $c_j$  invertiamo il verso del collegamento: dal nodo secondo al nodo  $c_j$  e dal nodo  $c_j$  al nodo primo.



Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo che la formula  $\phi$  sia soddisfacibile.

Questo vuol dire che  $\exists$  un assegnamento  $\sigma$  che verifica tutte le clausole.

Mostriamo che  $G_\phi$  ammette un HC.

L'HC parte da  $s$ . Ad ogni bivio, per raggiungere la catena  $i$ -esima percorriamo l'arco verde se  $x_i$  in  $\sigma$  è true, altrimenti quello rosso se  $x_i$  in  $\sigma$  è false. Mentre attraversiamo una catena seguiamo tutti i detour possibili verso nodi  $c_j$ . Quando arriviamo a  $t$  torniamo a  $s$ .

Siamo sicuri di visitare tutti i nodi  $c_j$  poiché  $\sigma$  rende true almeno una variabile per ogni clausola, quindi prima o poi effettueremo il detour verso il nodo corrispondente.

Abbiamo quindi la certezza che  $G_\phi$  ammetta un HC.

$\Leftarrow$ ) Supponiamo di essere arrivati ad un grafo  $G$  che ammette un HC  $\pi$ .

Dimostriamo che, per come è costruito  $G$ ,  $\pi$  non può passare da una catena all'altra percorrendo detour differenti (non salta dei piani), ovvero che se dal nodo  $x_i^{j'}$  si va al nodo  $c_j$ , allora dopo  $c_j$  si va in  $x_i^{j''}$ .

Supponiamo per assurdo che dopo aver visitato  $c_j$ ,  $\pi$  vada ad un nodo di un'altra catena.

Siccome  $\pi$  è un HC, deve visitare tutti i nodi una sola volta, quindi sarebbe costretto a visitare il secondo nodo della coppia arrivandoci dal tondino alla sua destra. Ma così facendo, dopo non potremmo proseguire, perciò non è possibile.

Il verso opposto, ovvero che se dal nodo  $x_i^{j''}$  si va al nodo  $c_j$ , allora dopo  $c_j$  si va in  $x_i^{j'}$ , si dimostra simmetricamente.

Questo significa che un HC sensato visita prima tutta una catena  $x_i$ , eventualmente passando per qualche  $c_j$ , e solo dopo passa alla catena successiva.

Costruiamo quindi un assegnamento  $\sigma$  per  $\phi$  tramite l'HC  $\pi$ . Consideriamo i nodi di scelta (bivio): se per raggiungere la catena  $i$ -esima percorriamo l'arco verde, poniamo  $x_i$  true, se percorriamo l'arco rosso, poniamo  $x_i$  false.

Siccome  $\pi$  è un HC, visitiamo tutti i nodi  $c_j$ . Per come è costruito  $G$ , significa che per ogni clausola stiamo assegnando true ad un letterale positivo oppure false ad un letterale negato, quindi tutte le clausole sono verificate.

Questo vuol dire che  $\sigma$  soddisfa  $\phi \implies \phi$  è soddisfacibile.

Siccome 3SAT è riducibile polinomialmente a DHC  $\implies$  DHC è NP-arduo.

### Seconda riduzione: DHC $\leq_P$ HC

Il problema HC, Hamiltonian Cycle, richiede di stabilire se all'interno di un grafo non orientato è presente un Hamiltonian cycle.

$\text{HC} = \{ G \mid G \text{ è un grafo non orientato che ammette un Hamiltonian cycle} \}$

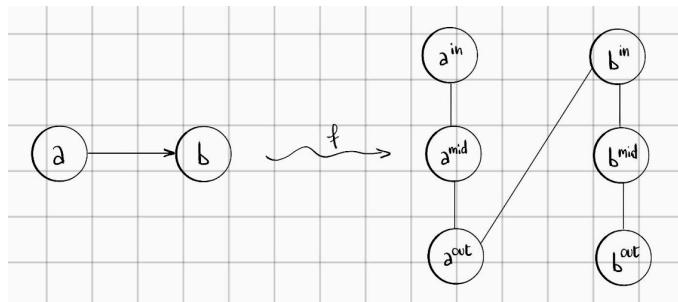
	DHC	$\leq_P$	HC
<b>Input</b>	Un grafo orientato $G$	$\rightsquigarrow$	Un grafo non orientato $H$
<b>Istanza-sì</b>	$G$ ammette un HC		$H$ ammette un HC
<b>Istanza-no</b>	$G$ non ammette un HC		$H$ non ammette un HC

Notiamo che la definizione dei due problemi è molto simile. Il nostro obiettivo è rappresentare la direzione degli archi tramite nodi aggiuntivi.

Per ogni nodo  $v$  in  $G$ , generiamo 3 nodi in  $H$ :  $v^{\text{in}}$ ,  $v^{\text{mid}}$  e  $v^{\text{out}}$ .

Queste triple le colleghiamo  $v^{\text{in}} - v^{\text{mid}} - v^{\text{out}}$ .

Per rappresentare un generico arco  $a \rightarrow b$  di  $G$ , connettiamo in  $H$  il nodo  $a^{\text{out}}$  al nodo  $b^{\text{in}}$ .



Verifichiamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo di avere in  $G$  un HC orientato  $\pi$ , tracciante un percorso  $v_1 \rightarrow \dots \rightarrow v_n$ .

Associando ad ogni nodo  $v$  la tripla corrispondente in  $H$  otteniamo equivalentemente un HC:

$v_1 \rightarrow \dots \rightarrow v_n \rightsquigarrow v_1^{\text{in}} \rightarrow v_1^{\text{mid}} \rightarrow v_1^{\text{out}} \rightarrow \dots \rightarrow v_n^{\text{in}} \rightarrow v_n^{\text{mid}} \rightarrow v_n^{\text{out}}$ .

Abbiamo la certezza di poter collegare le triplette tra loro poiché se due nodi sono collegati in  $G$ , allora per costruzione lo sono anche le triplette in  $H$ .

Pertanto  $H$  ammette un Hamiltonian cycle.

$\Leftarrow$ ) Supponiamo di essere arrivati ad un grafo  $H$  avente un HC  $\pi$ .

Per come è costruito  $H$ , la sequenza di apici attraversata da  $\pi$  deve per forza essere  $\text{in} \rightarrow \text{mid} \rightarrow \text{out}$  oppure  $\text{out} \rightarrow \text{mid} \rightarrow \text{in}$ , per ogni tripletta di nodi.

Assumendo di trovarci nel primo caso, possiamo costruire un HC per  $G$  considerando solo i nodi con apice  $\text{in}$ . Abbiamo la certezza che i nodi siano connessi correttamente in  $G$ , poiché in  $H$  c'è un arco tra  $\text{out}$  e  $\text{in}$ .

Questo prova che il grafo orientato di partenza  $G$  avesse un HC.

Siccome DHC è riducibile polinomialmente ad HC  $\implies$  HC è NP-arduo.

### Terza riduzione: HC $\leq_P$ TSP

Arriviamo quindi al nostro problema originale: TSP.

$\text{TSP} = \{ \langle G, k \rangle \mid G = \langle V, E, \lambda \rangle \text{ ammette un HC di peso al più } k \}$

	<b>HC</b>	$\leq_P$	<b>TSP</b>
<b>Input</b>	Un grafo non orientato $G$	$\rightsquigarrow$	Una coppia grafo pesato, peso $(H, k)$
<b>Istanza-sì</b>	$G$ ammette un HC		$H$ ammette un HC di peso al più $k$
<b>Istanza-no</b>	$G$ non ammette un HC		$H$ non ammette un HC di peso al più $k$

Il grafo  $H$  è uguale identico al grafo  $G$  (nodi e archi), dobbiamo solo aggiungere i pesi: assegniamo peso 1 a tutti gli archi. A questo punto il peso massimo  $k$  sarà pari al numero di nodi in  $G$ .

Dimostriamo la correttezza della riduzione:

$\Rightarrow$ ) Supponiamo di avere un grafo  $G$  avente un HC  $\pi$ .

Siccome la struttura di  $H$  è uguale a quella di  $G$ ,  $\pi$  è un HC anche per  $H$ .

Per definizione, l'HC attraversa un numero di archi pari al numero di nodi, quindi il peso di  $\pi$  è  $k$ . Ciò vuol dire che  $H$  ammette un HC di peso al più  $k$ .

$\Leftarrow$ ) Supponiamo di essere arrivati ad un grafo  $H$  avente un HC di peso al più  $k$ .

Siccome la struttura di  $G$  è uguale a quella di  $H$ ,  $G$  ammette un HC.

Siccome HC è riducibile polinomialmente a TSP  $\implies$  TSP è NP-arduo.

$\text{TSP} \in NP \wedge \text{TSP è NP-arduo} \implies \text{TSP è NP-completo.}$

That's all folks! Fine degli appunti del corso.