8. COMPLESSITÀ DEGLI ALGORITMI

La ***complessità*** ha lo scopo di dividere i problemi in grandi classi che dipendono dal fatto che esista o meno un algoritmo polinomiale per la soluzione.

Le classi che abbiamo visto e che vedremo sono:

- ***Indecidibilità***, ovvero che non esiste algoritmo che risolvi un dato problema;
- ***Classe P***, insieme di problemi per cui esiste un algoritmo polinomiale che li risolve, la cui complessità di tempo è $O(n^k)$;
- ***NP-Completezza***, non si conosce un algoritmo polinomiale e non si riesce a dire che l'algoritmo esiste o meno per un dato problema *(spoiler!!!)*.

I problemi che possiamo risolvere, *in pratica*, sono quelli che ammettono algoritmi aventi ***tempo polinomiale***.

Se abbiamo problemi con tempo polinomiale, gli algoritmi utilizzati possono essere usati anche per risolvere istanze grandi di un altro problema. Da questo abbiamo definito la classe P come classe di problemi che hanno algoritmi efficienti (ovvero con *tempo polinomiale*), per quanto riguarda gli altri problemi non appartenenti a P, non sono noti algoritmi che risolvono tali problemi in tempo polinomiale, forse questi ammettono algoritmi polinomiali che si basano su principi non ancora noti o semplicemente non possono essere risolti in tale modo.

8.1 RIDUZIONI POLINOMIALI

Supponiamo che possiamo risolvere il problema X in tempo polinomiale, se questo problema X si riduce ***efficientemente*** (ovvero in tempo polinomiale) al problema Y, una soluzione efficiente per Y può essere usata per risolvere X efficientemente.

Questa riduzione la si dimostra col concetto di "***riduzione polinomiale***", simile alla riduzione fatta in precedenza ma riguarda la complessità di tempo.

In altre parole, il problema X ***si riduce in tempo polinomiale*** al problema Y se arbitrarie istanze di X possono essere risolte usando un numero polinomiale di passi di computazione, più un numero polinomiale di chiamate ad un oracolo (algoritmo qualsiasi "*black-box*" che risolve istanze di Y in un solo passo) che risolve Y, tutto ciò si denota $\mathbf{X \leq_P Y}$.

Nota: le istanze di Y devono avere dimensione polinomiale, in quanto gli oracoli di Y richiedono tempo per elaborare sia l'input che l'output.

Lo ***scopo*** della riduzione polinomiale è classificare i problemi in accordo alla difficoltà ***relativa***.

Se $\mathbf{X \leq_P Y}$ e Y ammette soluzione in tempo polinomiale, allora anche X ammette soluzione in tempo polinomiale.
Al contrario, se $\mathbf{X \leq_P Y}$ e X non ammette soluzione in tempo polinomiale, allora anche Y non ammette soluzione in tempo polinomiale.
In alcuni casi è possibile stabilire un'uguaglianza tra riduzioni, se $X \leq_P Y$ e $Y \leq_P X$ allora $\mathbf{X \equiv_P Y}$.

RIDUZIONE MEDIANTE EQUIVALENZA SEMPLICE:

Per provare tale riduzione introduciamo due problemi NP noti:

INDEPENDENT-SET:

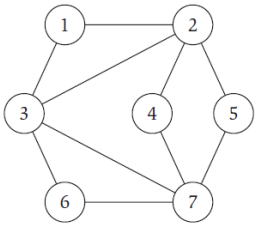
Dato un grafo G = (V, E), diciamo che un insieme di nodi $S \subseteq V$ è indipendente se non ci sono due nodi in S uniti da un arco. Il problema del Independent Set è il seguente: Dato G, trova un Independent Set che sia il più grande possibile.

Ad esempio, la dimensione massima di un Independent Set nella Figura in alto a destra è quattro, ottenuta dal Independent Set a quattro nodi {1, 4, 5, 6}.

Domanda: dato un grafo G = (V, E) e un intero k, esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \geq k$, e per ogni edge, ***almeno*** uno dei suoi estremi è in S?

Es: Esiste un insieme indipendente di dimensione ≥6, nella figura in basso a destra? Si.

Es: Esiste un insieme indipendente di dimensione ≥7, nella figura in basso a destra? No.



VERTEX-COVER:

Dato un grafo G = (V, E), diciamo che un sottoinsieme di nodi $S \subseteq V$ è una Vertex Cover se ogni bordo $e \in E$ ha almeno un'estremità (dell'arco) in S.

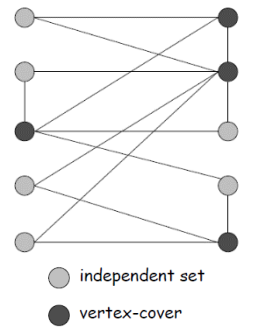
Si noti che in una Vertex Cover, i vertici fanno la "copertura", e gli archi sono gli oggetti che sono "coperti". Ora, è facile trovare grandi Vertex Cover in un grafo, la parte difficile è trovare quelle piccole.

Ad esempio, nella figura in alto a destra, l'insieme di nodi {1, 2, 6, 7} è una Vertex Cover di dimensione 4, mentre l'insieme {2,3,7} è una Vertex Cover di dimensione 3.

Domanda: dato un grafo G = (V, E) e un intero k, esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \leq k$, e per ogni edge, ***al più*** uno dei suoi estremi è in S?

Es: Esiste un vertex-cover di dimensione ≤4, nella figura in basso a destra? Si.

Es: Esiste un vertex-cover di dimensione ≤3, nella figura in basso a destra? No.



Non sappiamo come risolvere questi due problemi in tempo polinomiale, ma cosa possiamo dire della loro relativa difficoltà?

Mostriamo ora che sono equivalentemente difficili, ovvero $\mathbf{VERTEX\text{-}COVER \equiv_P INDEPENDENT\text{-}SET}$.

Dimostrazione:

Mostriamo che S è un Independent-Set sse il suo complemento V-S è un Vertex-Cover. Mostrandolo nelle due direzioni:

(⇒) $\mathbf{INDEPENDENT\text{-}SET \leq_P VERTEX\text{-}COVER}$

In primo luogo, supponiamo che S sia un Independent Set. Consideriamo un arco arbitrario e = (u, v), poiché S contiene vertici indipendenti, non è possibile che sia u e sia v siano in S quindi uno di essi deve essere per forza in V-S.

Ne consegue che ogni arco ha almeno un'estremità in V - S, e quindi V - S è una Vertex Cover.

Riassumendo:

Sia S è un qualsiasi independent set, consideriamo un arbitrario edge (u, v).

S indipendente → $u \notin S$ o $v \notin S$ → $u \in V\text{-}S$ o $v \in V\text{-}S$.

Quindi, V-S copre (u, v).

(⇐) $\mathbf{VERTEX\text{-}COVER \leq_P INDEPENDENT\text{-}SET}$

Al contrario, supponiamo che V - S sia una Vertex Cover. Consideriamo due nodi u e v in S, se fossero uniti da un arco *e*, allora nessuna delle estremità di *e* risiederebbe in V - S, contraddicendo la nostra ipotesi che V - S sia una Vertex Cover.

Ne consegue che due nodi in S non sono uniti da un arco *e*, quindi S è un Independent Set.

Riassumendo:

Sia V-S un qualsiasi vertex-cover, consideriamo due nodi $u \in S$ e $v \in S$. Notiamo che $(u, v) \notin E$ poiché V-S è un vertex-cover.

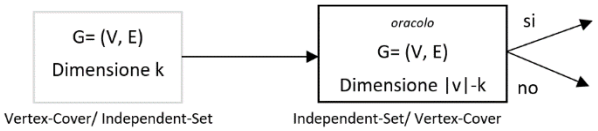
Quindi, non esistono due nodi in S connessi da un edge → S independent set.

VERTEX-COVER ≤P INDEPENDENT-SET

Se abbiamo un oracolo per risolvere l'Independent Set, allora possiamo decidere se G ha una Vertex Cover di dimensioni al massimo k, chiedendo all'oracolo se G ha un Independent Set di dimensioni almeno |v| - k.

INDEPENDENT-SET ≤P VERTEX-COVER

Se abbiamo un oracolo per risolvere la Vertex Cover, allora possiamo decidere se G ha un Independent Set di dimensioni almeno k, chiedendo all'oracolo se G ha una Vertex Cover di dimensioni al massimo |v|-k.



Per riassumere, questo tipo di analisi illustra il nostro piano in generale: anche se non sappiamo come risolvere in modo efficiente né l'Independent Set né la Vertex Cover, le due dimostrazioni ci dicono come potremmo risolvere uno di questi trovando una soluzione efficiente all'altro problema.

RIDUZIONE DA CASO SPECIALE A CASO GENERALE:

L'Independent Set può essere visto come un problema di imballaggio: l'obiettivo è quello di "impacchettare" il maggior numero possibile di vertici, senza a "conflitti" (senza archi che li collegano).

Il Vertex Cover, d'altra parte, può essere visto come un problema di copertura: l'obiettivo è "coprire" tutti gli archi del grafo usando il minor numero possibile di vertici.

Il Vertex Cover è un problema di copertura espresso specificamente nel linguaggio dei grafi, ma c'è un problema di copertura più generale, ovvero il **SET-COVER**, in cui si cerca di coprire un insieme arbitrario di oggetti usando una raccolta di insiemi più piccoli, ed è definito come:

Dato un insieme di elementi {1, 2, …, n} (chiamato universo U) e una raccolta S di m sottoinsiemi (S_1, S_2, …, S_m) la cui unione è uguale ad U, il ***Set Cover Problem*** è identificare le più piccole sotto-raccolte di S la cui unione è uguale all'universo U.

Domanda: Dato un insieme U di elementi, una collezione S_1, S_2, …, S_m di sottoinsiemi di U, e un intero k, esiste una collezione ≤k di questi sottoinsiemi la cui unione è uguale ad U?

Nell'esempio a destra, se k=2 il nostro obiettivo è trovare al più 2 sottoinsiemi la quale la loro unione forma U, in questo caso $S_2 \cup S_6 = U$.

U = { 1, 2, 3, 4, 5, 6, 7 }
k = 2
S_1 = {3, 7} S_4 = {2, 4}
S_2 = {3, 4, 5, 6} S_5 = {5}
S_3 = {1} S_6 = {1, 2, 6, 7}

Intuitivamente, sembra che Vertex Cover sia un caso speciale di Set Cover: in quest'ultimo caso, stiamo cercando di coprire un insieme arbitrario di vertici usando sottoinsiemi arbitrari di vertici stessi, mentre nel primo caso, stiamo specificamente cercando di coprire gli archi di un grafo usando l'insieme degli archi incidente ai vertici, possiamo provare che **VERTEX-COVER ≤P SET-COVER**.

Dimostrazione:

oracolo
Risolve Set-Cover

Supponiamo di avere accesso a un oracolo in grado di risolvere Set Cover e prendere in considerazione un'istanza arbitraria di Vertex Cover, specificata da un grafo G = (V, E) e un numero k.

Il nostro obiettivo è quello di coprire gli archi in E, quindi formuliamo un'istanza di Set Cover in cui l'insieme U è uguale ad E.

Ogni volta che selezioniamo un vertice nel Vertex Cover Problem, copriamo tutti gli archi incidenti ad esso; quindi, per ogni vertice $v \in V$, aggiungiamo un insieme $S_v \subseteq U$ alla nostra istanza Set Cover, costituito da tutti gli archi in G incidenti a v.

Ora ricordiamo che U può essere coperto con al massimo k degli insiemi S_1, …, S_n se e solo se G ha una Vertex Cover di dimensioni al massimo k.

(⇒) Questo può essere dimostrato molto facilmente, ovvero se S_{v1}, …, S_{vl} sono $l \leq k$ insiemi che coprono U, allora ogni arco in G è incidente a uno dei vertici v_1, …, v_l, e quindi l'insieme $\{v_1, …, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$.

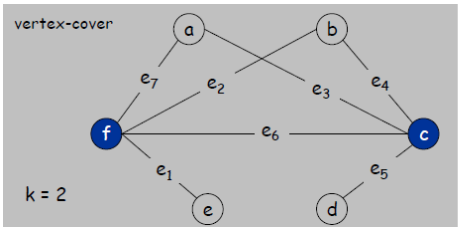
(⇐) Al contrario, se $\{v_1, …, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$, allora gli insiemi S_{v1}, …, S_{vl} coprono U.

Pertanto, data la nostra istanza di Vertex Cover, formuliamo l'istanza di Set Cover descritta sopra e la passiamo al nostro oracolo.

Rispondiamo sì se e solo se l'oracolo risponde sì.

Ricapitolando:

Creiamo un'istanza di SET-COVER: k = k, U = E, S_v = {e ∈ E: e incidente su v}



SET-COVER
U = { 1, 2, 3, 4, 5, 6, 7 }
k = 2
S_a = {3, 7} S_b = {2, 4}
S_c = {3, 4, 5, 6} S_d = {5}
S_e = {1} S_f = {1, 2, 6, 7}

Set-cover è di dimensione ≤k sse vertex-cover è di dimensione ≤k (dimostrata sopra).

RIDUZIONE VIA "GADGETS":

Prima di parlare di questa riduzione introduciamo il ***problema della soddisfacibilità***, uno dei problemi noti NP-Completo (*classe descritta più avanti*).

Si ricordi che le variabili booleane assumono solo valori vero o falso (1 o 0), con queste si possono creare operazioni booleane tramite AND, OR e NOT.

Una ***formula booleana*** è un'espressione che coinvolge variabili e operazioni booleane, per esempio $\varphi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$, ed è ***soddisfacibile*** se qualche assegnamento di 0 e 1 alle variabili fa sì che la formula valga 1, nella formula precedente lo si fa con x=0, y=1 e z=0, quindi diciamo che l'assegnamento ***soddisfa*** φ. Da tutto ciò, diciamo che il ***problema della soddisfacibilità*** consiste nel verificare se una formula booleana è soddisfacibile, denotata **SAT.**

Utilizzeremo un caso speciale di SAT, ovvero 3SAT, in cui tutte le formule booleane sono in una forma speciale. Prima di parlare di questa forma speciale introduciamo altri due concetti che serviranno, ovvero ***letterale*** che è una variabile booleana o il suo negato, e ***clausola C_j*** che consiste in diversi letterali connessi tramite operatore **OR (V)**, ad esempio $C_1 = (\overline{x} \vee y \vee z)$.

Una formula booleana è in ***forma normale congiunta***, ed è detta una ***forma cnf***, se comprende diverse clausole connesse tramite operatore **AND (∧)** ($\varphi = C_1 \wedge C_2 \wedge C_3$), come ad esempio la seguente formula: $\varphi = (\overline{x} \vee y \vee z) \wedge (x \vee \overline{z}) \wedge (y)$.

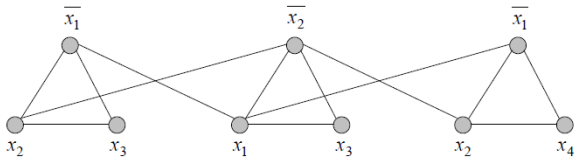
Essa è una ***formula 3cnf*** se tutte le clausole della formula hanno esattamente tre letterali, come: $(\overline{x} \vee y \vee z) \wedge (x \vee y \vee \overline{z}) \wedge (\overline{x} \vee y \vee \overline{z})$.

Questo problema di verificare se una formula booleana è in 3cnf, viene definito **3-SAT= {⟨φ⟩| φ è una formula 3cnf soddisfacibile}.** *Domanda*:

Dato un insieme di clausole C_1, …, C_k, ciascuno di lunghezza 3, su un insieme di variabili $X = \{x_1, …, x_n\}$, esiste un assegnamento di verità soddisfacente?

Mettiamo in relazione la difficoltà computazionale tra ***3-SAT***, che riguarda l'impostazione delle variabili booleane in presenza di vincoli, e ***Indipendent-Set***, che riguarda la selezione dei vertici indipendenti in un grafo.
Per risolvere un'istanza di 3-SAT, usando un oracolo che risolve Indipendent-Set, abbiamo bisogno di una codifica di tutti i vincoli booleani in nodi e archi di un grafo, in modo che la soddisfacibilità corrisponda all'esistenza di trovare un insieme indipendente, dimostriamo **3-SAT≤$_P$Indipendent-Set**.
Dimostrazione:
Abbiamo un oracolo per Independent-Set e vogliamo risolvere un'istanza di 3-SAT composta da variabili $X = \{x_1, \dots, x_n\}$ e clausole $C_1, \dots, C_k$.
Da quanto detto, dobbiamo trovare un modo di prendere la formula booleana da risolvere e convertirla in un grafo, affinché l'oracolo che già abbiamo risolvi il nostro problema. Diciamo che due termini sono in "conflitto" se una variabile x_i ha in comune un arco col suo negato $\overline{x_i}$.
Un modo di risolvere un'istanza di 3-SAT è il seguente: scegliere un termine da ciascuna clausola, non quelli in "conflitto", e uguagliarli ad 1, in modo da soddisfare tutte le clausole.
Iniziamo a costruire il grafo G che contiene 3 vertici per ogni clausola, uno per ogni letterale, connettiamo i 3 letterali in una clausola (che forma un triangolo), ed infine connettiamo ogni letterale ad ogni suo negato, risultando come segue:

G



k = 3

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Prima di procedere, considera come appaiono gli insiemi indipendenti di dimensione k in questo grafo: Non è possibile selezionare due vertici dallo stesso triangolo, ma questo è proprio il nostro obiettivo, ovvero quello di scegliere un termine in ciascuna clausola, che poi verrà assegnato a 1, soddisfacendo la nostra formula booleana.
Da tutto questo si enuncia che: **G contiene un insieme indipendente di dimensione k=|φ| sse φ è soddisfacibile**, in altre parole supponiamo che l'istanza 3-SAT originale sia soddisfacibile se e solo se il grafo G, che abbiamo costruito, ha un insieme indipendente di dimensioni ***almeno*** k.
Dimostriamolo nei due versi:

(⇐) In primo luogo, supponiamo che il nostro grafo G abbia un insieme indipendente S di dimensioni almeno k. Quindi la dimensione di S è esattamente k, e deve consistere in un nodo per ogni triangolo. Ora, sosteniamo che esiste un'assegnazione di verità v per le variabili nell'istanza 3-SAT con la proprietà che le etichette di tutti i nodi in S valgono 1. Ecco come possiamo costruire tale assegnazione v. Se un nodo in S fosse etichettato $\overline{x_i}$ e un altro fosse etichettato x_i, allora ci sarebbe un arco tra questi due nodi, contraddicendo la nostra ipotesi, ovvero che S sia un insieme indipendente. Pertanto, se x_i appare come un'etichetta di un nodo in S, impostiamo $v(x_i)=1$, altrimenti impostiamo $v(x_i)=0$. Costruendo v in questo modo, tutte le etichette dei nodi in S valgono 1.
Riassumendo:
Sia S l'insieme indipendente di dimensione k:
- S deve contenere esattamente un vertice di ogni triangolo
- Non può contenere un letterale ed il suo negato

Poniamo questi letterali di S a 1, l'assegnazione di verità è consistente e tutte le clausole sono soddisfatte.

(⇒⇐) Al contrario, se l'istanza 3-SAT è soddisfacibile, allora ogni triangolo nel grafo contiene almeno un nodo che ha valore 1. Sia S un insieme costituito da uno di questi nodi da ciascun triangolo. Sosteniamo che S è indipendente, poiché se ci fosse un arco tra due nodi u, v ∈ S, allora le etichette di u e v dovrebbero essere in conflitto (uno 0 e l'altro 1), ma questo non è possibile, poiché entrambi i vertici valgono 1.
Riassumendo:
Data un'assegnazione soddisfacente, selezioniamo un letterale che vale 1 da ogni triangolo.
Questo è un insieme indipendente di dimensione k.

Per concludere, poiché G ha un Indipendent-Set di dimensioni almeno k se e solo se l'istanza 3-SAT originale è soddisfacente, la riduzione è completa.

TRANSITIVITÀ DELLE RIDUZIONI:

Se X≤$_P$Y e Y≤$_P$Z, allora X≤$_P$Z

Dimostrazione:
Dato un oracolo per Z, mostriamo come risolvere un'istanza di X, essenzialmente componiamo solo i due algoritmi implicati da X≤$_P$Y e Y≤$_P$Z.
Eseguiamo l'algoritmo per X usando un oracolo per Y; ma ogni volta che viene chiamato l'oracolo per Y, lo simuliamo in un numero polinomiale di passaggi usando l'algoritmo che risolve istanze di Y, ma in tal caso stiamo usando un oracolo per Z.
Ad esempio, poiché abbiamo dimostrato che **3-SAT ≤$_P$ Independent Set ≤$_P$ Vertex Cover ≤$_P$ Set Cover**, possiamo concludere che **3-SAT ≤$_P$ Set Cover**.

8.2 CERTIFICAZIONE EFFICIENTE E DEFINIZIONE DI NP

Quando abbiamo parlato di Indipendent-Set Problem abbiamo detto che un grafo contiene un insieme indipendente di dimensioni almeno k, per provarlo si può benissimo esibire un esempio di grafo in modo tale che contiene k insiemi indipendenti. Allo stesso modo, se un'istanza 3-SAT è soddisfacibile, basta prendere una formula booleana in forma 3cnf e assegnare i vari valori alle variabili in modo che tale formula risulti 1.
Il problema è il contrasto tra la ***ricerca*** di una soluzione e il ***controllo*** di una soluzione proposta.
Per Independent-Set o 3-SAT, non conosciamo un algoritmo a tempo polinomiale per trovare tali soluzioni, ma il controllo di una soluzione proposta a questi problemi può essere facilmente eseguito in tempi polinomiali.
Da ciò possiamo dire che esistono due diversi problemi, un ***problema di decisione***, ovvero controllare se esiste una soluzione al problema, e un ***problema di ricerca***, ovvero trovare l'effettiva soluzione al problema, noi ci occuperemo di problemi relativi al primo tipo.
Questo ci porta al **Self-Reducibility**, il quale afferma che un **problema di ricerca ≤$_P$ problema di decisione**, ciò si applica a tutti i problemi NP-Completi che vedremo più avanti, giustificando la focalizzazione sui problemi di decisione.

PROBLEMI E ALGORITMI:

L'input per un problema computazionale verrà codificato come una stringa binaria finita s, ed indichiamo la lunghezza con |s|. Identificheremo un problema decisionale X con l'insieme di stringhe su cui la risposta è "sì". Un algoritmo A per un problema di decisione riceve una stringa di input se restituisce il valore "sì" o "no": indicheremo questo valore restituito con A(s).

Diciamo che l'algoritmo A risolve il problema X se per tutte le stringhe s, abbiamo A(s) = si, se e solo se s∈X.

Ricapitolando: Un ***problema di decisione*** è caratterizzato da:

- X insieme di stringhe provenienti dal nostro problema decisionale
- Un istanza di X è una stringa s
- Algoritmo A risolve il problema X: A(s) = si sse s∈X.

Come sempre, diciamo che A ha un tempo di esecuzione polinomiale se esiste una ***funzione polinomiale*** p(·) in modo che per ogni stringa di input s, l'algoritmo A termina su s al più in p(|s|) passi.

Ricapitolando: ***Tempo polinomiale***:

Algoritmo A ha tempo polinomiale se per ogni stringa s, A(s) termina in al più p(|s|) "passi", dove p() è qualche polinomio.

CERTIFICAZIONE EFFICIENTE:

Un "***algoritmo di controllo***", chiamato "***certificatore***", per un problema X ha una struttura diversa da un algoritmo che cerca effettivamente di risolvere il problema. Per molti problemi non è noto se esiste un algoritmo polinomiale di soluzione.

Un certificatore non serve a risolvere il problema, ma permette di verificare se l'istanza di un problema è una istanza "si" con l'aiuto di un altro elemento, ovvero il ***certificato*** denotato con t.

Per "***controllare***" una soluzione, abbiamo bisogno della stringa di input s, nonché di una stringa di "***certificato***" separata t che contenga la prova che s è un'istanza "si" di X.

Tecnicamente ciò significa che esiste un algoritmo di verifica C(s, t) che, usando t, controlla in tempo polinomiale l'esistenza di una soluzione per s avente la proprietà richiesta.

Quindi diciamo che C è un ***certificatore efficiente*** per un problema X se valgono le seguenti proprietà:

- *C* è un algoritmo a tempo polinomiale che accetta due argomenti di input s e t.
- Esiste una funzione polinomiale p in modo che per ogni stringa s, abbiamo s ∈ X se e solo se esiste una stringa t tale che |t|≤p(|s|) e C(s, t)= sì.

Algoritmo C(s, t) è un ***certificatore*** per il problema X se per ogni stringa s, s ∈ X sse esiste una stringa t (certificato) tale che C(s, t) = si.
Nota: Il ***certificatore*** non determina se s∈X, semplicemente controlla se una data dimostrazione t che s∈X, quindi ***vede se il certificato è corretto***.

Esempio: Un certificatore efficiente C può essere utilizzato come componente principale di un algoritmo "bruteforce" per un problema X: su un input s, prova tutte le stringhe t di lunghezza ≤ p(|s|) e vede se C(s, t)=sì per una di queste stringhe.

Ma l'esistenza di C non ci fornisce alcun modo chiaro per progettare un algoritmo efficiente che risolva effettivamente X; dopo tutto, spetta ancora a noi trovare una stringa t che farà sì che C(s, t) dica "sì" e ci sono molte possibilità esponenziali per t.

CLASSE DEI PROBLEMI - NP:

Definiamo NP come l'insieme di tutti i problemi per i quali esiste un ***certificatore efficiente***. L'atto di cercare una stringa t che farà sì che un certificatore efficiente accetti gli input s, viene spesso visto come una ricerca non deterministica nello spazio delle possibili dimostrazioni t; per questo motivo, NP è stato nominato acronimo di " ***Non-deterministic Polynomial-time*** ".

Adesso possiamo ridefinire alcune classi di problemi già viste:

- **P**: Problema di decisione per cui esiste un *algoritmo polinomiale*.
- **EXP**: Problema di decisione per cui esiste un *algoritmo esponenziale*.
- **NP**: Problema di decisione per cui esiste *certificatore polinomiale*.

Da quanto appena detto possiamo osservare che la classe **P⊆NP**.

Dimostrazione:

Consideriamo un problema X ∈ P, questo significa che esiste un algoritmo in tempo polinomiale A che risolve X. Per dimostrare che X ∈ NP, dobbiamo dimostrare che esiste un certificatore efficiente C per X, progettiamo C come segue:

Quando C viene eseguito con una coppia di input (s, t), il certificatore C restituisce semplicemente il valore di A(s). C è un certificatore efficiente per X perché ha un tempo di esecuzione polinomiale, poiché esegue A (che ha un tempo polinomiale). Quindi, se una stringa s∈X, allora per ogni t di lunghezza al massimo p(|s|), abbiamo C(s, t)=sì. D'altra parte, se s∉X, allora per ogni t di lunghezza al massimo p(|s|), abbiamo C(s, t) = no.

Riassumendo:

Consideriamo un qualsiasi problema X in P

Per Definizione, esiste un algoritmo polinomiale A(s) che risolve X.

Certificato: t = ε, mentre il certificatore: C(s, t) = A(s).

Adesso possiamo verificare che alcuni problemi introdotti precedentemente appartengono ad NP, si tratta di determinare in che modo un certificatore efficiente, per ciascuno di essi, utilizzerà una stringa t "certificato":

Ricordiamo che un numero è composto se è il prodotto di due numeri maggiori di 1, ovvero è composto se non è primo.

Il primo problema polinomialmente verificabile è il **COMPOSITES**, ovvero dato un intero s, questo verifica se è composto. Abbiamo:

Il ***certificato***:

Un fattore non banale t di s. Nota che tale certificato esiste sse s è composito. Inoltre |t|≤|s|.

Ed il ***certificatore***:

```
boolean C(s, t) {
   se (t ≤ 1 or t ≥ s)
      Restituisci  false
   else se (s è a multiplo di t)
      Restituisci  true
   else
      Restituisci  false
}
```

Una prova di tale problema può essere: Istanza s=437,669 → Certificato t=541 o 809 (437,669 = 541 × 809)

In conclusione, **COMPOSITES** è in NP.

Il secondo problema polinomialmente verificabile è il **SAT/3-SAT**, ovvero data una formula CNF φ, verifica se esiste un'assegnazione che la soddisfa.
Il ***certificato*** sarà un assegnamento di valori di verità alle n variabili booleane.
Il ***certificatore*** invece controlla che ogni clausola in φ ha almeno un letterale ad 1.
Una prova di tale problema può essere:

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$$

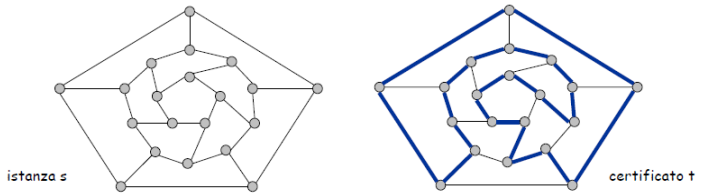
istanza s

$$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1$$

certificato t

In conclusione, **SAT/3-SAT** è in NP.

Il terzo problema polinomialmente verificabile è **HAM-CYCLE**, ovvero dato un grafo non orientato G=(V,E), verifica se esiste un ciclo semplice C che visita ogni nodo esattamente una volta.
Il ***certificato*** è una permutazione di n nodi.
Il ***certificatore*** controlla che la permutazione contiene ogni nodo in V esattamente una volta, e che esiste un arco tra ogni coppia di nodi adiacenti nella permutazione.
Una prova di tale problema può essere:



In conclusione, **HAM-CYCLE** è in NP.

Tra l'altro, non sappiamo se NP è contenuta in una classe di complessità deterministica più piccola, il miglior metodo deterministico attuale noto per decidere se un linguaggio è in NP, è vedere se utilizza un tempo esponenziale. In altre parole, possiamo dimostrare che **NP⊆EXP**.

Dimostrazione:

Consideriamo un qualsiasi problema X in NP.
Per definizione, esiste un certificatore polinomiale C(s, t) per X.
Per risolvere X su input s, usiamo C(s, t) su tutte le stringhe t con $|t| \leq p(|s|)$.
Restituisci si, se C(s, t) restituisce si per una qualsiasi stringa.

LA QUESTIONE P=NP?

La domanda se P=NP è uno dei maggiori problemi irrisolti dell'informatica teorica, se queste classi fossero uguali, qualsiasi problema polinomialmente verificabile sarebbe anche polinomialmente decidibile. La maggior parte dei ricercatori ritengono che non siano uguali, perché molte persone hanno investito, senza successo, sforzi per trovare algoritmi aventi tempo polinomiale per problemi in NP.
Tuttavia, non possiamo dimostrare che nessuno di questi problemi richieda più del tempo polinomiale per risolverlo. In effetti, non possiamo dimostrare che in NP vi siano problemi che non appartengono a P. Quindi al posto di un teorema concreto, possiamo solo fare una domanda:

C'è un problema in NP che non appartiene a P? P = NP?

8.3 NP-COMPLETEZZA

In assenza di progressi sulla domanda P = NP, le persone si sono rivolte a una domanda correlata ma più accessibile: quali sono i problemi più difficili in NP? Le riduzioni polinomiali ci offrono un modo per affrontare questa domanda e ottenere informazioni sulla struttura di NP.

Approfondimento (Tesi di Cook-Levin):
Un problema L è NP completo se sta in NP e se ∀L' (appartenente anch'esso a NP), L'≤L, ovvero se L è il più difficile problema in NP.
In altre parole, possiamo dire che un linguaggio L è NP-completo se i seguenti enunciati sono veri:

1. L è in NP
2. Per ogni Linguaggio L' in NP esiste una riduzione polinomiale di L' a L

si può dire sia un caso particolare della *Cook-completezza* che definisce un problema NP-completo se, dato un *oracolo* per il problema P, ovvero un meccanismo in grado di rispondere ad una qualsiasi domanda sull'appartenenza di una stringa a P in un'unità di tempo, è possibile riconoscere in tempo polinomiale un qualsiasi linguaggio in NP. La definizione di *Cook-completezza* risulta essere più generale tanto da includere i complementi dei problemi NP-completi nella classe dei problemi NP-completi.
"Riducibile" quindi significa che per ogni problema L, c'è una *riduzione polinomiale*, ovvero un algoritmo deterministico che trasforma istanze $l \in L$ in istanze $c \in C$, così che la risposta a c è sì se e solo se la risposta a l è sì. Per provare che un problema NP A è infatti un problema NP-completo è sufficiente mostrare che un problema NP-completo già conosciuto si riduce a A.
Una conseguenza di questa definizione è che se avessimo un algoritmo di tempo polinomiale per C, potremmo risolvere tutti i problemi in NP in tempo polinomiale, questa definizione è stata data da Stephen Cook.
Un problema che soddisfa la condizione 2 ma non necessariamente la condizione 1 è detto NP-hard. Informalmente, un problema NP-hard è "almeno difficile come" qualsiasi problema NP-completo, e forse anche più difficile.

Il problema X ***si riduce in modo polinomiale*** (Cook) al problema Y se istanze arbitrarie del problema X possono essere risolte usando: Un numero polinomiale di passi di computazione standard, più un numero polinomiale di chiamate ad oracolo che risolve il problema Y.
Il problema X ***si trasforma in modo polinomiale*** (Karp) al problema Y se dato un qualsiasi input x∈X, possiamo costruire un input y tale che x è istanza "sì" di X sse y è istanza "sì" di Y.
Nota: Le trasformazioni polinomiali sono riduzioni polinomiali con una sola chiamata all'oracolo per Y, esattamente alla fine dell'algoritmo per X. Quasi tutte le precedenti riduzioni erano di questa forma.

DEFINIZIONE NP-COMPLETO: Probabilmente il modo più naturale per definire un problema "più difficile" X, è attraverso le seguenti due proprietà:

(i) X ∈ NP;

(ii) per tutti Y ∈ NP, Y $\leq_P$ X.

In altre parole, richiediamo che ogni problema in NP *possa essere ridotto a X*. Chiameremo tale X un problema NP-completo.

Supponiamo che X sia un problema NP-completo, allora X è risolvibile in tempo polinomiale se e solo se **P=NP**.

Dimostrazione:

(⇐) Chiaramente, se P = NP, allora X può essere risolto in tempo polinomiale poiché appartiene a NP.

(⇒) Al contrario, supponiamo che X possa essere risolto in tempo polinomiale.
Se Y è un altro problema in NP, allora Y $\leq_P$ X, e quindi Y può essere risolto in tempo polinomiale.
Ciò implica NP⊆P, ma già sappiamo che P⊆NP, e quindi P=NP.

Una conseguenza di questa dimostrazione è: Se c'è un problema in NP che non può essere risolto in tempo polinomiale, allora nessun problema NP completo può essere risolto in tempo polinomiale.

CIRCUIT SATISFIABILITY, il primo problema NP-Completo:

Il primo problema NP-Completo è il **CIRCUIT-SAT**, chiamato anche "*soddisfacibilità dei circuiti*".

Per dimostrare che un problema è NP-completo, si deve mostrare come quest'ultimo potrebbe codifica qualsiasi problema in NP. Questa è una questione molto più complicata di quanto riscontrato nelle riduzioni polinomiali precedenti, in cui abbiamo cercato di codificare specifici problemi individuali in termini di altri problemi già noti.

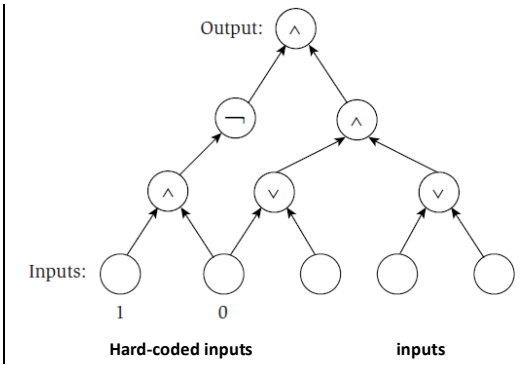
Nel 1971, Cook e Levin mostrarono indipendentemente come farlo per problemi molto naturali in NP. Forse la scelta del problema più naturale per un primo problema NP-completo è il seguente ***Circuit Satisfiability Problem***.

Per specificare questo problema, dobbiamo precisare cosa intendiamo per circuito. Consideriamo gli operatori booleani standard che abbiamo usato per definire il problema di soddisfazione: ∧ (AND), ∨ (OR) e ¬ (NOT). La nostra definizione di circuito è progettata per rappresentare un circuito fisico costruito da nodi che implementano questi operatori. Quindi definiamo un circuito K come un grafo aciclico marcato e diretto come il seguente:

- Le foglie in K sono etichettate con una delle costanti 0 o 1 o con il nome di una variabile distinta. I nodi di quest'ultimo tipo saranno indicati come gli input al circuito.
- Ogni altro nodo è etichettato con uno degli operatori booleani ∧, ∨ o ¬; i nodi etichettati con ∧ o ∨ avranno due archi in entrata e i nodi etichettati con ¬ avranno un arco in entrata.
- Esiste un singolo nodo senza archi in uscita che rappresenterà l'output: il risultato che viene calcolato dal circuito, ovvero la radice dell'albero.

Alle due foglie a sinistra sono assegnati i valori 1 e 0 e le tre foglie successive costituiscono gli input.

Ad esempio, se in input abbiamo 101, otteniamo nel penultimo livello 011 per i rispettivi nodi di quel, per i nodi del secondo livello otteniamo 11 ed infine otterremo 1 sulla radice, la quale rappresenta il nostro output (che ha soddisfatto la formula iniziale essendo =1).



Ora, il problema di soddisfazione del circuito è il seguente: Ci viene dato un circuito e dobbiamo decidere se esiste un'assegnazione di valori in input che fa sì che l'output restituisca il valore 1 (in tal caso, diremo che il circuito dato è soddisfatto).

Nel nostro esempio, abbiamo appena visto, tramite l'assegnazione 101 in input, che il circuito è soddisfatto ed è quindi istanza "sì".

Possiamo vedere il teorema di Cook e Levin nel modo seguente: **CIRCUIT-SAT è NP-Completo.**

Come già discusso, la dimostrazione di questo teorema richiede che consideriamo un problema arbitrario X in NP e mostrare che **X$\leq_P$CIRCUIT-SAT**.

Non descriveremo la dimostrazione in dettaglio, ma in realtà non è poi così difficile basta seguire l'idea di base:

Un qualsiasi algoritmo che prende in input un numero fissato n di bits e produce una risposta si/no può essere rappresentato con tale circuito inoltre, se l'algoritmo prende tempo polinomiale, allora il circuito è di dimensione polinomiale.

Questo circuito è equivalente ad un algoritmo, nel senso che il suo output è esattamente 1 sugli input per i quali l'algoritmo produce sì.

(*Si noti che un algoritmo non ha problemi a gestire input di lunghezze variabili, ma un circuito è strutturalmente codificato con una dimensione fissa*).

Dimostrazione(***idea***):

Consideriamo un problema X in NP, questo ha un certificatore polinomiale C(s, t) per definizione.

Per determinare se s è in X, dobbiamo sapere se esiste un certificato t di lunghezza p(|s|) tale che C(s, t) = si. Consideriamo C(s, t) come un algoritmo su |s|+p(|s|) bits (input s, certificato t) e convertiamolo in circuito di dimensione polinomiale K con |s|+p(|s|) foglie.

- i primi |s| bits (foglie) sono ***hard-coded*** con s (bits fissi)
- restanti p(|s|) bits (foglie) rappresentano i bits di t

Quindi, il circuito K è soddisfacibile se e solo se C(s, t) = si, ovvero se il circuito restituisce 1.

Ora osserviamo semplicemente che s ∈ X se e solo se esiste un modo per impostare i bit di ingresso su K in modo che il circuito produca 1.

Ciò stabilisce che **X$\leq_P$CIRCUIT-SAT**.

Esempio:

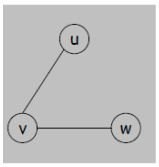
Supponiamo di avere il seguente problema: Dato un grafico G, contiene un indipendent-set (problema in NP) a due nodi?

Vediamo come un'istanza di questo problema può essere risolta costruendo un'istanza equivalente di CIRCUIT-SAT.

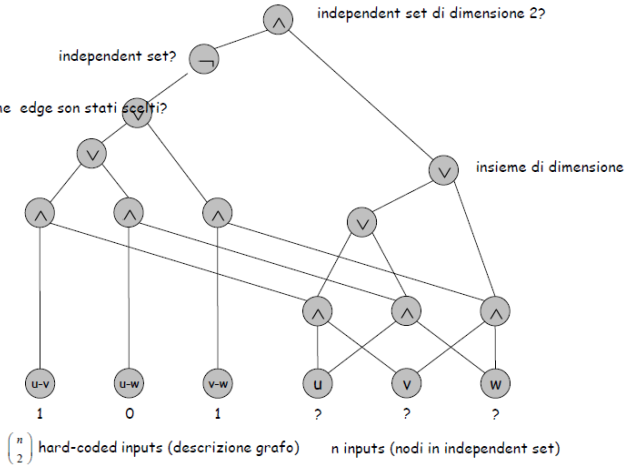
Costruiamo un circuito equivalente K:

Supponiamo che siamo interessati a decidere la risposta a questo problema per un grafo G sui tre nodi u, v, w, in cui v è unito sia a u che a w (grafo rappresentato dalla figura a destra a sinistra). Ciò significa che ci occupiamo di un input di lunghezza n = 3. La Figura a destra mostra un circuito equivalente a un certificatore efficiente per il nostro problema (il lato destro del circuito verifica che siano stati selezionati almeno due nodi e il lato sinistro verifica che non abbiamo selezionato nodi collegati da un arco). Codifichiamo gli archi di G come costanti nelle prime tre foglie e lasciamo le restanti tre foglie (che rappresentano la scelta dei nodi da inserire nell'insieme indipendente) come variabili.

Ora osserviamo che questa istanza di CIRCUIT-SAT è soddisfacibile con l'assegnazione u=1, v=0 e w=1 come input.



G = (V, E), n = 3



Ciò corrisponde alla scelta dei nodi *uw*, che in effetti formano un insieme indipendente a due nodi nel nostro grafo G.

STABILIRE NP-COMPLETEZZA:

Una volta che conosciamo il primo problema NP-completo, possiamo scoprirne altri attraverso la seguente osservazione:

Se X è un problema NP-completo e Y è un problema in NP con la proprietà che $X \leq_P Y$, allora Y è NP-completo.

Dimostrazione:

Sia W un qualsiasi problema in NP, abbiamo $W \leq_P X$, per definizione di NP-completo, e $X \leq_P Y$ per ipotesi, tramite transitività ne consegue che $W \leq_P Y$. Quindi Y è NP-Completo.

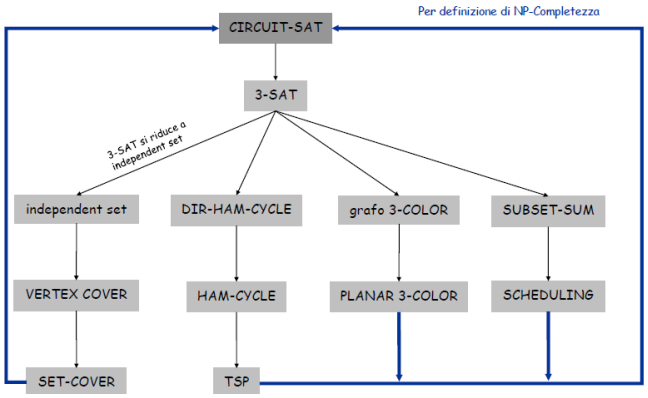
Riassumendo: **Passi per stabilire NP-Completezza** di problema Y:

- passo 1. Mostrare che Y è in NP.
- passo 2. Scegliere un problema NP-Completo X.
- passo 3. Provare che $X \leq_p Y$.

In precedenza, abbiamo riscontrato una serie di riduzioni tra alcuni problemi di base, come Indipendet-Set, Ham-Cycle, etc….

Per stabilire la loro NP-completezza, basterebbe collegare CIRCUIT-SAT a quest'ultimi, ma risulterebbe molto lungo e difficile.

Il modo più semplice per farlo è mettere in relazione CIRCUIT-SAT con 3-SAT, più precisamente riduciamo tutti questi problemi di base a 3-SAT, ed infine riduciamo quest'ultimo a CIRCUIT-SAT, così per la transitività e come se riducessimo tutti questi problemi di base, a CIRCUIT-SAT.



Osservazione: Tutti i problemi sono NP-Completi e ammettono riduzione polinomiale da uno all'altro!

Pratica: La maggior parte dei problemi NP sono noti essere in P oppure essere NP-Completi.

Adesso mostriamo che **3-SAT è NP-Completo**.

Dimostrazione:

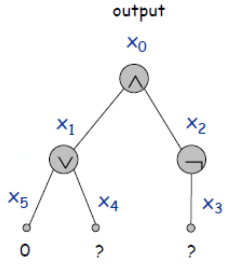
Chiaramente 3-SAT è in NP, poiché possiamo verificare in tempo polinomiale che un'assegnazione di verità soddisfa l'insieme di clausole dato.

Dimostreremo che è NP-completo tramite la riduzione **CIRCUIT-SAT$\leq_P$3-SAT**.

Data un'istanza arbitraria di CIRCUIT-SAT, costruiremo un'istanza equivalente di 3-SAT in cui ogni clausola contiene al massimo tre variabili.

Quindi consideriamo un circuito arbitrario K ed associamo delle variabili 3-SAT x_v a ciascun nodo v del circuito. Ora dobbiamo codificare i nodi etichettati con operatori booleani, in modo da soddisfare le clausole. Ci saranno tre casi a seconda della tipologia di operazione:

- Se il nodo x_2 è etichettato con ¬, e il suo solo arco di input proviene dal nodo x_3, allora dobbiamo avere $x_2 = \overline{x}_3$. Trasformiamo ciò aggiungendo 2 clausole $(x_2 \vee x_3)$ e $(\overline{x}_2 \vee \overline{x}_3)$.
- Se il nodo x_1 è etichettato con **V** e i suoi due archi di input provengono dai nodi x_4 e x_5, dobbiamo avere $x_1 = x_4 \vee x_5$. Trasformiamo ciò aggiungendo le 3 clausole: $(x_1 \vee \overline{x}_4)$, $(x_1 \vee \overline{x}_5)$ e $(\overline{x}_1 \vee x_4 \vee x_5)$.
- Se il nodo x_0 è etichettato con **Λ** e i suoi due archi di input provengono dai nodi x_1 e x_2, dobbiamo avere $x_0 = x_1 \wedge x_2$. Trasformiamo ciò aggiungendo le 3 clausole: $(\overline{x}_0 \vee x_1)$, $(\overline{x}_0 \vee x_2)$ e $(x_0 \vee \overline{x}_1 \vee \overline{x}_2)$.



Hard-coded input e output:

Dobbiamo garantire che le costanti (foglie a sinistra dell'albero) abbiano i loro valori e che l'output sia uguale a 1.

Pertanto, per il nodo x_5 che è stato etichettato come una costante, aggiungiamo una clausola con la variabile x_5 o $\overline{x}_5$, in modo da forzare x_5 ad assumere il valore designato, in questo caso aggiungiamo la clausola con la singola variabile $\overline{x}_5$.

Per il nodo x_0 che rappresenta l'output, aggiungiamo la clausola con la singola variabile x_0, la quale richiede di valere 1, concludendo la costruzione.

Passo finale:

Finora abbiamo creato un'istanza SAT, ma il nostro obiettivo era quello di creare un'istanza di 3-SAT, quindi dobbiamo convertire questa istanza di SAT in una istanza equivalente in cui ogni clausola ha esattamente tre variabili.

Per far ciò, bisogna inserire altre variabili nelle clausole con lunghezza <3, e assegnare tali variabili con valori in modo che il risultato non cambi.

8.4 CO-NP

Abbiamo visto che l'idea di un certificatore efficiente non suggerisce un algoritmo concreto per risolvere effettivamente il problema.
La definizione di certificazione efficiente, e quindi in NP, è fondamentalmente *asimmetrica*:
Una stringa di input s è un'istanza "si" se e solo se *esiste una* t in modo che C(s, t) = "si", negando questa affermazione, vediamo che una stringa di input s è un'istanza "no" se e solo se *per tutte le* t, abbiamo che C(s, t) = "no".
Per ogni problema X, esiste un problema naturale ***complementare*** $\overline{X}$: per tutte le stringhe di input s, diciamo $s \in \overline{X}$ se e solo se $s \notin X$. Nota che se $X \in$ classe P, allora $\overline{X} \in$ P, poiché da un algoritmo A che risolve X, possiamo produrre un algoritmo $\overline{A}$ che esegue A e poi inverte la sua risposta.
In altre parole, dato un problema di decisione X, il ***complemento*** $\overline{X}$ è lo stesso problema con risposte si e no invertite.
Ma se $X \in$ NP, dovrebbe essere $\overline{X} \in$ NP. Il problema $\overline{X}$, invece, ha una proprietà diversa in NP: per tutti s, abbiamo $s \in \overline{X}$ se e solo se per ogni t di lunghezza al massimo p(|s|), abbiamo C(s, t) = no.
In termini concreti: data una serie insoddisfacente di clausole, come posiamo provare rapidamente che *non* esiste un'assegnazione soddisfacente?
Questa è una definizione fondamentalmente diversa e non può essere aggirata semplicemente "invertendo" l'output del certificatore efficiente C, per produrre $\overline{C}$. Il problema è che l'"esiste t" nella definizione di NP è diventata un "per ogni t", e questo è un problema.
Esiste una classe di problemi parallela a NP progettata per modellare questo specifico problema, chiamato **co-NP**.

Un problema X appartiene a co-NP se e solo se il problema complementare $\overline{X}$ appartiene a NP.

Non sappiamo con certezza che NP e co-NP siano diversi; possiamo solo chiederci: **NP=co-NP**?
Dimostrare NP = co-NP sarebbe un passo ancora più grande di provare P = NP, per il seguente motivo:

Se NP≠co-NP, allora P≠NP

Dimostrazione:
Dimostriamo l'affermazione contrapposta, ovvero: se P = NP allora NP = co-NP.
Il punto è che P è chiusa per il complemento, quindi se P = NP, allora anche NP è chiusa per il complemento. Partendo dal presupposto P=NP, abbiamo:

$$X \in \text{NP} \Rightarrow X \in \text{P} \Rightarrow \overline{X} \in \text{P} \Rightarrow \overline{X} \in \text{NP} \Rightarrow X \in \text{co-NP} \qquad \text{e} \qquad X \in \text{co-NP} \Rightarrow \overline{X} \in \text{NP} \Rightarrow \overline{X} \in \text{P} \Rightarrow X \in \text{P} \Rightarrow X \in \text{NP}.$$

Quindi ne conseguirebbe che NP ⊆ co-NP e co-NP ⊆ NP, da cui NP = co-NP.

CARATTERIZZAZIONE NP ∩ co-NP:
Se il problema X è sia in NP e co-NP, allora se istanza è "si" esiste un certificato succinto, mentre se l'istanza è "no" esiste un "disqualifier" succinto.
Pertanto, si dice che i problemi che appartengono a questa intersezione NP ∩ co-NP abbiano una ***buona caratterizzazione***, poiché esiste sempre un buon certificato per la soluzione.
Esempio:

Dato un grafo bipartito, esso ammette un perfect matching, e questo tipo di grafo è in NP ∩ co-NP:

- se "si", possiamo fornire un perfect matching.
- se "no", possiamo fornire un insieme di nodi S tale che |N(S)| < |S| (*fornendo una metodologia o teorema per provarlo*).

Nota: un grafo ha un perfect matching sse per ogni S risulta |N(S)| ≥ |S| (*N sta per neighborhood, ovvero il numero di nodi vicini*).

Naturalmente, si vorrebbe sapere se esiste un problema con una buona caratterizzazione ma nessun algoritmo del tempo polinomiale. Ma anche questa è una domanda aperta: **P = NP ∩ co-NP**? Al momento possiamo solo osservare che **P ⊆ NP ∩ co-NP.**
L'opinione generale sembra alquanto contrastata su questa domanda. In parte, questo è perché ci sono molti casi in cui è stato riscontrato che un problema ha una buona caratterizzazione non banale; e poi (molti anni dopo) si scoprì anche che aveva un algoritmo polinomiale.
Adesso vediamo 2 problemi appartenenti a NP ∩ co-NP:

Teorema. PRIMES è in NP ∩co-NP.
Dim. Gia sappiamo che PRIMES è in co-NP. Basta provare che PRIMES è in NP.

Teorema. Un intero dispari s è primo sse esiste intero 1 < t < s t.c.

$$t^{s-1} \equiv 1 \pmod{s}$$
$$t^{(s-1)/p} \neq 1 \pmod{s}$$
per tutti i divisori primi p di s-1

Input. s = 437,677
Certificato. t = 17, $2^2 \times 3 \times 36{,}473$

↑

Fattorizzazione di s-1: richiede certificati per asserire che 2, 3 e 36,473 sono primi

Il certificatore.
- Controlliamo s-1 = $2 \times 2 \times 3 \times 36{,}473$.
- Controlliamo $17^{s-1} \equiv 1 \pmod{s}$.
- Controlliamo $17^{(s-1)/2} \equiv 437{,}676 \pmod{s}$.
- Controlliamo $17^{(s-1)/3} \equiv 329{,}415 \pmod{s}$.
- Controlliamo $17^{(s-1)/36{,}473} \equiv 305{,}452 \pmod{s}$.

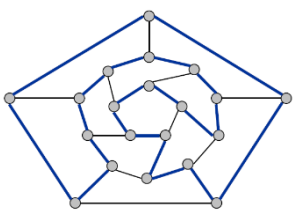
FACTORIZE. Dato intero x, trova fattorizzazione.

FACTOR. Dati due interi x e y, x ha un fattore minore di y?

Teorema. FACTOR è in NP ∩co-NP.
Dim.
- Certificato: un fattore p di x minore di y.
- Disqualifier: fattorizzazione di x (senza fattore minore di y), e certificato che ogni fattore è primo

8.5 PROBLEMI DI SEQUENCING

Con questi problemi troviamo se esiste, una *sequenza* di elementi che risolve il problema dato.
HAMILTONIAN CYCLE PROBLEM:
Dato un grafo G=(V, E), diciamo che un ciclo C in G è un ciclo Hamiltoniano se visita ogni vertice esattamente una volta. In altre parole, costituisce un "giro" di tutti i vertici, senza ripetizioni.
Il problema del ciclo hamiltoniano è:
Dato un grafo *non orientato* G=(V, E), esiste un ciclo semplice Γ che contiene ogni nodo in V?



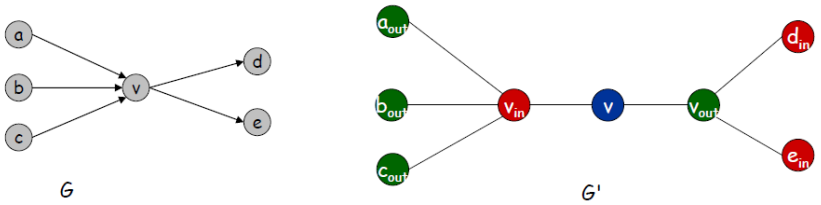
DIR-HAM-CYCLE:

Consideriamo una versione orientata di HAM-CYCLE: Dato un grafo *orientato* G=(V, E), esiste un ciclo orientato semplice Γ contenente ogni nodo in V?

Vogliamo far vedere che **DIR-HAM-CYCLE≤$_P$HAM-CYCLE**.

Dimostrazione:

Dato un grafo orientato G = (V, E), costruiamo un grafo non orientato G'.



Dobbiamo mostrare che **G ha un ciclo Hamiltoniano se e solo se G' lo ha anche esso**.

(⇒) Ciascun nodo u di G, eccetto s e t, viene sostituito da una tripla di nodi u_{in}, u, u_{out} in G', collegati tra loro tramite archi, mentre i nodi s e t verranno sostituiti rispettivamente con s_{out} e t_{in}.
I nodi u e v in G, che sono collegati da un arco, in G´ saranno collegati da u_{out} e v_{in} come in figura sopra, ciò completa la costruzione in G.
Per dimostrare che ciò funziona, facciamo vedere che un cammino Hamiltoniano in G, ad esempio s-u-v-t, in G' sarà s_{out}-u_{in}-u-u_{out}-v_{in}-v-v_{out}-t_{in}.

(⇐) Affermiamo che qualsiasi cammino Hamiltoniano in G' da s_{out} a t_{in} deve andare da una tripla di nodi ad un'altra tripla, eccetto per l'inizio e la fine, perché qualsiasi cammino di questo tipo ha un cammino Hamiltoniano corrispondente in G.
Per confermare ciò, se si inizia il cammino da s_{out} e lo si segue, non è possibile andare al di fuori delle varie triple, fino a t_{in}.

Mostriamo ora che **3-SAT≤$_P$DIR-HAM-CYCLE**, e proviamo che quest'ultimo è NP-Completo.

Innanzitutto, mostriamo che DIR-HAM-CYCLE è in NP. Dato un grafo diretto G=(V, E) e un certificato, il quale mostra che esiste una soluzione, ovvero l'elenco ordinato dei vertici su un ciclo hamiltoniano. Potremmo *verificare*, in tempo polinomiale, che l'elenco dei vertici contenga ogni vertice esattamente una volta e che ogni coppia consecutiva nell'ordinamento sia unita da un arco, ciò stabilirebbe che definisce un ciclo Hamiltoniano.

Dimostrazione:

Consideriamo un'istanza arbitraria di 3-SAT, con le variabili x_1, …, x_n e clausole C_1, …, C_k, essenzialmente, possiamo impostare i valori delle variabili come vogliamo e ci vengono date tre possibilità per soddisfare ogni clausola.

Per ciascuna formula φ, facciamo vedere come costruire un grafo diretto G con due nodi, s e t, in cui esiste un cammino Hamiltoniano tra s e t se e solo se φ è soddisfacibile.

Iniziamo la costruzione con una formula 3cnf φ contenente k clausole: $\varphi=(a_1 \vee b_1 \vee c_1)\wedge(a_2 \vee b_2 \vee c_2)\wedge \dots \wedge(a_k \vee b_k \vee c_k)$, dove ciascun a, b, e c è un letterale x_i o $\overline{x}_i$ e siano x_1, …, x_l le l variabili di φ.

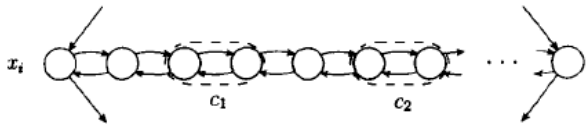
Rappresentiamo ciascuna variabile x_i, con una struttura di forma romboidale che contiene una riga orizzontale di nodi.

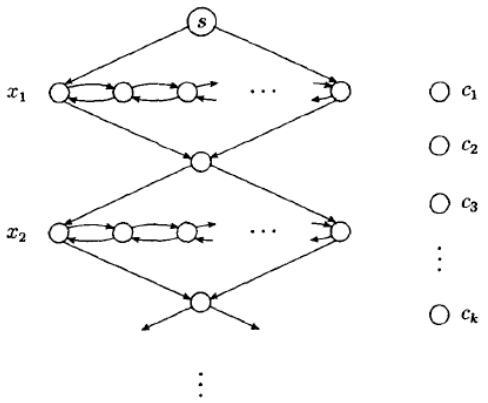
Rappresentiamo ciascuna clausola di φ con un singolo nodo ⊙ c_j.

La figura a destra riporta la struttura globale di G, essa mostra tutti gli elementi di G e le loro relazioni, eccetto le relazioni delle variabili con le loro clausole.

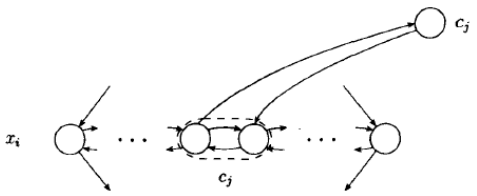
Ciascuna struttura romboidale contiene una riga orizzontale di nodi collegati tramite archi orientati in entrambe le direzioni, questa riga contiene 3k+1 nodi in aggiunta ai due nodi alle estremità del rombo.

Questi nodi sono raggruppati in coppie adiacenti, una per ciascuna clausola, con nodi separati aggiuntivi tra le coppie, come segue:

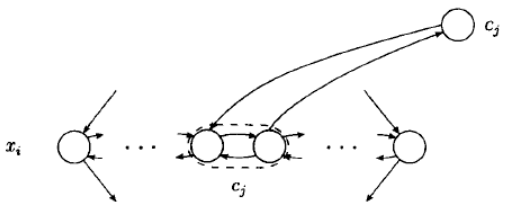




Se la variabile x_i è presente nella clausola c_j, aggiungiamo due archi dalla coppia j-esima nell'i-esimo rombo al j-esimo nodo della clausola, come segue:

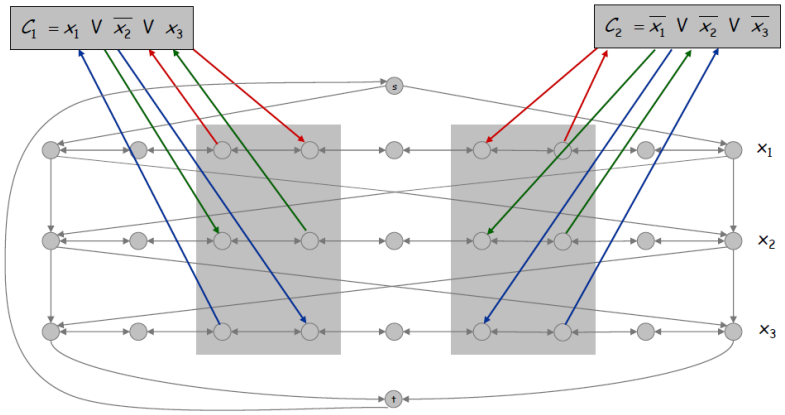


Se $\overline{x}_i$ è presente nella clausola c_j, aggiungiamo due archi dalla coppia j-esima nell'i-esimo rombo al j-esimo nodo della clausola, come segue:



Dopo aver aggiunto tutti gli archi alle rispettive occorrenze, la costruzione di G è completa. Per far vedere che funziona tale costruzione, dimostriamo che: Data un'istanza di 3-SAT, costruiamo un'istanza di **DIR-HAM-CYCLE ammette un circuito Hamiltoniano se e solo se φ è soddisfacibile**.

Mostriamo prima un esempio complessivo per includere tutto ciò che è stato detto:



(⇐) Supponiamo che φ sia soddisfacibile, per mostrare un cammino Hamiltoniano da s a t, ignoriamo per il momento i nodi clausola. Il cammino inizia da s, attraversa ciascun rombo in successione, e termina in t.
Per raggiungere i nodi orizzontali in un rombo, il cammino può procedere in due direzioni, ovvero da sinistra a destra e viceversa, l'assegnamento che soddisfa φ determina quale scegliere dei due versi.
Se x_i =1 allora il cammino procede da sinistra a destra, altrimenti se x_i =0 va da destra a sinistra.
Possiamo dire quindi che per ogni clausola C_j, esiste una riga i-esima che attraversa nella direzione "corretta" (modalità descritta alla riga precedente) per includere il nodo C_j nel circuito.

(⇒) Se G ha un ciclo Hamiltoniano da s a t, facciamo vedere un assegnamento che soddisfa φ. Se il cammino Hamiltoniano passa attraverso i rombi in ordine da quello più in alto a quello più in basso e muovendosi da sinistra a destra o viceversa, possiamo determinare quali valori avranno le variabili booleane, ovvero se il cammino procede da sinistra a destra allora x_i =1, altrimenti se va da destra a sinistra x_i =0.

Nel caso dell'esempio sopra possiamo dire che le due clausole, per essere soddisfatte, x_2 dovrà essere uguale a 0, mentre le altre qualsiasi valore.

TRAVELING SALESMAN PROBLEM (Problema del commesso viaggiatore):

Probabilmente il sequencing problem più famoso è il "***Problema del commesso viaggiatore***".
Consideriamo un venditore che deve visitare n città etichettate $v_1, v_2, \ldots, v_n$. Il venditore inizia nella città v_1, la sua casa, e vuole trovare un ***tour***, ovvero un ordine in cui visitare tutte le altre città e tornare a casa. Il suo obiettivo è trovare un tour che gli faccia percorrere la minor distanza totale possibile.
Più formalmente, per ogni coppia ordinata di città (v_i, v_j), specificheremo un numero non negativo $d(v_i, v_j)$ come distanza da v_i a v_j.
La ragione di ciò è rendere la nostra formulazione il più generale possibile. In effetti, il commesso viaggiatore si presenta naturalmente in molte applicazioni in cui i punti non sono città e il viaggiatore non è un commesso. Pertanto, dato l'insieme delle distanze, bisogna ordinare le città in un tour $v_1, \ldots, v_n$, in modo da ridurre al minimo la distanza totale.
Da quanto detto ci chiediamo: **Dato un insieme di n città e una funzione di distanza d(u, v), esiste un tour di lunghezza ≤D?**
Per dimostrare quanto detto usiamo HAM-CYCLE, e mostriamo che **Traveling Salesman è NP-Completo**.
Dimostrazione:
È facile vedere che il commesso viaggiatore è in NP: il certificato è una permutazione delle città e un certificatore verifica che la durata del tour corrispondente sia al massimo il limite indicato D.
Mostriamo ora che **Hamiltonian Cycle≤$_P$Traveling Salesman**.
Dato un grafo diretto G = (V, E), definiamo la seguente istanza di Traveling Salesman: Abbiamo una città v per ogni nodo u del grafo G.
Definiamo d(u, v) = 1 se c'è un arco (u, v) in G, altrimenti sarà uguale a 2, più formalmente:

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$$

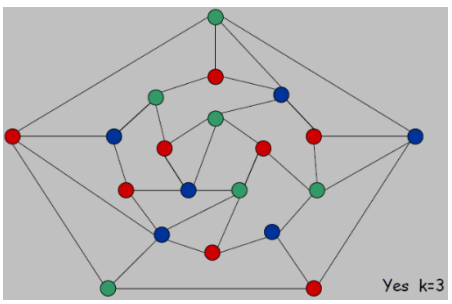
Ora affermiamo che **G ha un ciclo Hamiltoniano se e solo se c'è un tour di lunghezza al massimo n nella nostra istanza di Traveling Salesman**.

(⇒) Se G ha un ciclo Hamiltoniano, allora questo ordinamento delle città corrispondenti definisce un giro di lunghezza n.

(⇐) Al contrario, supponiamo che ci sia un tour di lunghezza al massimo n, ad esempio A-B-C-D-(e poi torno ad A).
La durata di questo tour è la somma degli n termini, ognuno dei quali è almeno 1; quindi deve essere il caso che tutti i termini sugli archi siano uguali a 1. Quindi ogni coppia di nodi in G, che corrispondono a città consecutive nel tour, deve essere collegata da un arco; ne consegue che l'ordinamento di questi nodi corrispondenti deve formare un ciclo Hamiltoniano.

8.6 PROBLEMI DI PARTITIONING

Adesso consideriamo dei problemi fondamentali di partizionamento, in cui cerchiamo dei modi per dividere una raccolta di oggetti in sottoinsiemi.
Un problema di questo tipo è quello di trovare un grafo bipartito, ovvero un grafo dove l'insieme dei suoi vertici si può partizionare in due sottoinsiemi tali che ogni vertice di una di queste due parti è collegato solo a vertici dell'altro insieme, questo è risolvibile in tempo polinomiale.
Ma le cose diventano più complicate quando passiamo da *coppie* ordinate a *triple* ordinate.
Per semplificare la comprensione del problema di trovare triple ordinate in un dato grafo, aiutiamoci con i colori.
La colorazione del grafo si riferisce allo stesso processo su un grafo G non orientato, coi nodi che svolgono il ruolo delle regioni da colorare e gli archi che rappresentano le coppie vicine. Cerchiamo di assegnare un colore a ciascun nodo di G in modo che se (u, v) è un arco, u e v saranno colorati in modo diverso.
Più formalmente, una k colorazione di G è una funzione f: V → {1, 2, …, k} in modo che per ogni arco (u, v), abbiamo f(u) ≠ f(v) (i colori qui sono chiamati 1, 2, …, k, e la funzione f rappresenta la scelta di un colore per ogni nodo). Questo introduce un problema noto, ovvero **k-COLOR**:



Dato un grafo non orientato G esiste un modo di colorare i nodi usando k colori in modo che nodi adiacenti NON hanno lo stesso colore?

Se consideriamo k=2, questo è il problema di determinare se un grafo G è bipartito, ma se passiamo a k=3, le cose diventano molto più difficili.
In effetti, il caso del grafo tricolore è già un problema molto difficile e possiamo dimostrare che **3-COLOR è NP-Completo**.

Dimostrazione:

È facile capire perché il problema si trova in NP. Dati G e k, un certificato che afferma che la risposta è "sì" è una colorazione k: si può verificare in tempo polinomiale che al massimo vengono usati i k colori e che nessuna coppia di nodi uniti da un arco riceve lo stesso colore.

3-COLOR è un problema difficile da mettere in relazione con altri problemi NP-completi che abbiamo visto.

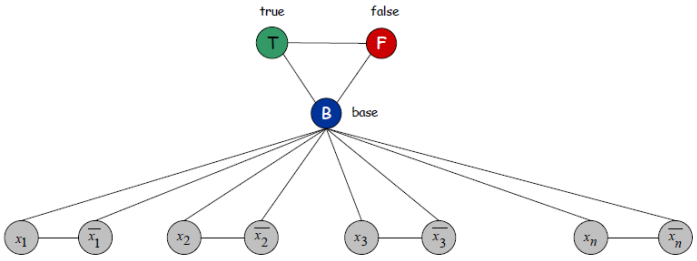
Quindi, ancora una volta, torneremo su 3-SAT, mostrando che **3-SAT≤$_P$3-COLOR**.

Data un'istanza arbitraria di 3-SAT, con variabili $x_1, \ldots, x_n$ e clausole $C_1, \ldots, C_k$, lo risolveremo usando un oracolo per 3-COLOR.

Definiamo tre "nodi speciali" T, F e B, che chiamiamo Vero, Falso e Base. Adesso costruiamo il grafo:

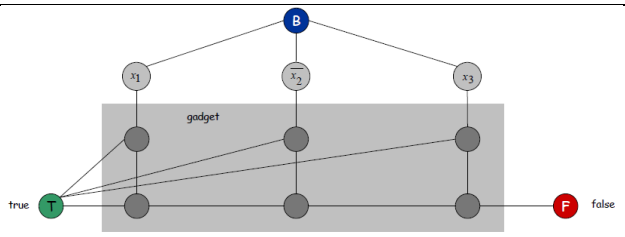
1. Creiamo un nodo per ogni letterale.
2. Creiamo 3 nuovi nodi T, F, B, assicurandoci che questi 3 abbiano colori diversi, connettiamoli in un triangolo, e colleghiamo ogni letterale negato e non, al nodo B.
3. Connettiamo ogni letterale alla sua negazione, quest'ultimo dovrà avere un colore diverso rispetto al nodo B e al nodo letterale corrispondente non negato.

Ogni nodo letterale ha valore T o F.



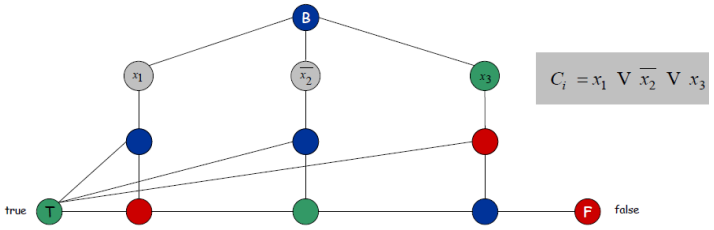
Tramite la riduzione via "Gadget", aggiungiamo un gadget di 6 nodi e 13 archi, come in figura a destra →

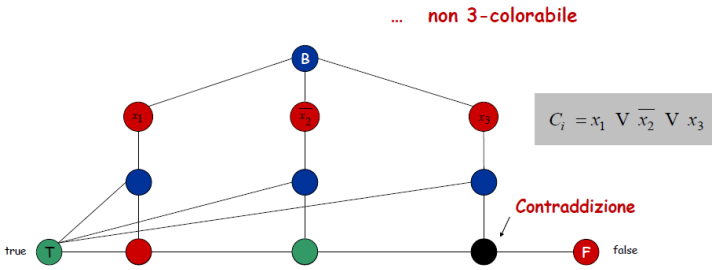
Il gadget assicura che almeno un letterale in ogni clausola è T



Consideriamo una clausola $C=(x_1 \vee \overline{x}_2 \vee x_3)$ e supponiamo che tutte le variabili siano tutte False, avremmo la figura a destra →

Quindi troviamo un'istanza "no", ma se almeno uno dei 3 fosse uguale a True avremmo risolto, e se proviamo a colorare troveremmo la soluzione, ovvero:





Possiamo completare la costruzione: iniziamo con il grafo G definito per primo, e per ogni clausola nell'istanza 3-SAT, alleghiamo un sottografo a sei nodi, come in seconda figura. Dimostriamo ora che **l'istanza 3-SAT è soddisfacente se e solo se G ha una 3-COLOR**.

(⇒) In primo luogo, supponiamo che ci sia un'assegnazione soddisfacente per l'istanza 3-SAT. Definiamo una colorazione di G colorando prima **Base**, **Vero** e **Falso** arbitrariamente con i tre colori, quindi, per ogni i, assegnando a v_i il colore Vero se $x_i = 1$ e il colore Falso se $x_i = 0$. Infine, è ora possibile estendere questa 3-COLOR in ciascun sottografo della clausola a sei nodi, risultando una 3-COLOR in tutto G.

(⇐) Al contrario, supponiamo che G abbia un 3-COLOR. In questa colorazione, a ciascun nodo v_i viene assegnato il colore True o False; impostiamo la variabile x_i di conseguenza. Ora affermiamo che in ciascuna clausola dell'istanza 3-SAT, almeno uno dei termini nella clausola ha il valore di verità 1. In caso contrario, tutti e tre i nodi corrispondenti hanno il colore Falso nella 3-COLOR di G e, come abbiamo visto sopra, non vi è alcuna 3-COLOR del sottografo della clausola corrispondente, ed abbiamo quindi una ***contraddizione***.

8.7 PROBLEMI NURERICI

Consideriamo ora alcuni problemi computazionalmente difficili che coinvolgono operazioni aritmetiche sui numeri.

SUBSET-SUM:

Il nostro problema di base in questo caso sarà la somma dei sottoinsiemi:

Dati dei numeri naturali $w_1, \ldots, w_n$ e intero W, esiste un sottoinsieme la cui somma è esattamente W?

Esempio:

{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344}, W = 3754 → si: 1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754

Poiché i numeri interi vengono forniti in rappresentazione binaria, o in base 10, la quantità W è realmente esponenziale nella dimensione dell'input;

Quindi una domanda di base è: il SUBSET SUM può essere risolta da un algoritmo polinomiale?

Il seguente risultato suggerisce che non è probabile che ciò avvenga, e possiamo dimostrare che **SUBSET SUM è NP-Completo**.

Dimostrazione:

Mostriamo innanzitutto che SUBSET-SUM è in NP. Dati i numeri naturali $w_1, \ldots, w_n$ e un obiettivo W, un certificato attestante che esiste una soluzione sarebbe il sottoinsieme $w_1, \ldots, w_k$ che si suppone che la somma è W. Nel tempo polinomiale, possiamo calcolare la somma di questi numeri e verificare che sia uguale a W. Una volta dimostrato che SUBSET-SUM è in NP, mostriamo che **3-SAT≤$_P$SUBSET-SUM**.

Sia un'istanza ϕ di 3-SAT con variabili $x_1, \ldots, x_n$ e clausole $c_1, \ldots, c_k$, dimostriamo che **SUBSET-SUM ha una soluzione se e solo se ϕ è soddisfacibile.**

(⇐) Data un'istanza ϕ con n variabili e k clausole, formiamo 2n+2k interi, ognuno di n+k cifre, dobbiamo mostrare che ϕ è soddisfacibile se presi dei sottoinsiemi la somma di questi risulta W.

Supponiamo che abbiamo queste clausole:

$$C_1 = \bar{x} \vee y \vee z$$
$$C_2 = x \vee \bar{y} \vee z$$
$$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$$

Possiamo osservare che abbiamo n=k=3 e quindi 2(3)+2(3)=12 interi (quindi 12 righe della tabella).
Adesso costruiamo la tabella a destra, ha una colonna per ogni variabile e clausole date, ed una riga per ogni letterale ed infine abbiamo altre 6 (in realtà sono 2k) righe non indicizzate.
Al suo interno metteremo il valore 1 se abbiamo ad esempio la colonna x in corrispondenza con le righe x e ¬x, in più un altro valore 1 in corrispondenza alle clausole di appartenenza delle variabili, ad esempio x appare nella clausola C_2.
Per quanto riguarda la parte non indicizzata della tabella, le colonne corrispondenti alle variabili saranno 0, mentre quelle corrispondenti alle clausole avranno dei numeri fissi, le prime due righe della seconda parte della tabella, solo la colonna corrispondente alla clausola C_1 avrà valori, ovvero 1 con 2 sottostante, e così via. Per finire, l'ultima riga, ovvero W, ci saranno degli 1 in corrispondenza delle variabili e 4 in corrispondenza delle clausole.
Da questa tabella ricaviamo dei numeri interpretando le righe (numeri posti al lato destro della tb).

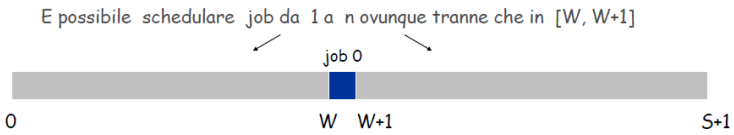
	x	y	z	C_1	C_2	C_3	
x	1	0	0	0	1	0	100,010
¬ x	1	0	0	1	0	1	100,101
y	0	1	0	1	0	0	10,100
¬ y	0	1	0	0	1	1	10,011
z	0	0	1	1	1	0	1,110
¬ z	0	0	1	0	0	1	1,001
	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
W	1	1	1	4	4	4	111,444

Supponiamo che la formula è soddisfacibile prendendo y=z= true e x=false (e quindi ¬x=true) e prendiamo i numeri corrispondenti in tabella.
Adesso consideriamo i numeri posti a destra della tabella corrispondenti al nostro assegnamento di verità (righe in verde) e sommiamoli provando ad ottenere l'ultimo numero ovvero 111,444, se non riusciamo ad ottenere quest'ultimo soltanto sommando i numeri presi dalle variabili dell'assegnamento di verità, possiamo prendere i numeri della seconda parte della tabella, per arrivare al numero obiettivo.

(⇒) Viceversa, se ho dei sottoinsiemi di interi, e quindi una soluzione a SUBSET-SUM, considero le righe corrispondenti e vediamo a che letterale corrisponde. Il numero 4 nell'ultima rica mi assicura che deve esserci almeno un 1 tra le righe selezionate, ma questo mi dice che c'è almeno un letterale posto a true e quindi la clausola corrispondente è soddisfatta.

SCHEDULE-RELEASE-TIMES:

La difficoltà di SUBSET SUM può essere utilizzata per stabilire la difficoltà di **SCHEDULE-RELEASE-TIMES**.
Supponiamo di avere una serie di n jobs che devono essere eseguiti su una singola macchina.
Ogni lavoro i ha un *release time* r_i quando è disponibile per la prima elaborazione, una *deadline* d_i entro la quale deve essere completato, e un *processing time* t_i. Supponiamo che tutti questi parametri siano numeri naturali.
Per essere completato, il lavoro i deve essere assegnato ad uno slot contiguo di unità di tempo t_i da qualche parte nell'intervallo $[r_i, d_i]$. La domanda è: **possiamo pianificare tutti i lavori in modo tale che ciascuno venga completato entro la sua scadenza?**
Dimostriamo che **SCHEDULE-RELEASE-TIMES è NP-Completo.**
Dimostrazione:
Data un'istanza del problema, un certificato risolvibile sarà una specifica *release time* per ciascun lavoro. Potremmo quindi verificare che ogni lavoro venga eseguito per un intervallo di tempo distinto, tra la sua *release time* e la sua *deadline*. Quindi il problema è in NP.
Mostriamo ora che **SUBSET-SUM≤$_P$SCHEDULE-RELEASE-TIMES**.
Pertanto, si consideri un'istanza di SUBSET-SUM con numeri $w_1, \ldots, w_n$ e un numero W.
Innanzitutto, creiamo n jobs con *processing time* $t_i=w_i$, *release time* $r_i=0$ (tutti i jobs arrivano allo stesso tempo all'inizio quindi sono tutti immediatamente disponibili), e nessuna deadline, ma poniamo $d_j = \mathbf{1+}\sum_j^n w_j$.
Da quanto detto potremmo mettere tutti i jobs come vogliamo, non incontrando conflitti durante la schedulazione.
A questo punto, creiamo un job 0 con $t_0=1$, release time $r_0=W$ e deadline $d_0=W+1$, ma questo è problematico in quanto ha $r_0=W$ e $d_0=W+1$, quindi il job 0 lo si deve schedulare quando arriva al tempo W ed eseguirlo immediatamente perché deve finire in tempo W+1. Ma adesso è come se il nostro intervallo fosse diviso in due, la parte sinistra da 0 a W, quindi ho W unità di tempo, e a destra c'è il resto, dovendo terminare entro $\mathbf{1+}\sum_j^n w_j$ (che indichiamo come S+1 per semplicità). Seguendo questa impostazione, abbiamo che le nostre unità di tempo adesso sono S, ovvero pari al tempo che abbiamo bisogno per andare a eseguire tutti i job, ma lo possiamo fare solo se riusciamo a distribuire i job metà prima del job 0 e l'altra metà dopo.



Mostriamo che **SUBSET-SUM ha una soluzione se e solo se troviamo uno SCHEDULE-RELEASE-TIMES fattibile**.

(⇐) Quindi dobbiamo dividere gli interi $w_1, \ldots, w_n$ in qualche modo in un sottoinsieme, in modo tale che quest'ultimo sia eseguibile prima che arrivi il job 0 ed il resto sarà eseguito dopo che job 0 è finito, ma questa è proprio ciò che si deve fare per risolvere SUBSET-SUM su un insieme di interi.

(⇒) Quindi se l'istanza di SUBSET-SUM è un'istanza "si" allora posso dividere $w_1, \ldots, w_n$ in due parti, di cui una somma W, che corrisponde a tutti i jobs eseguiti prima del job 0, mentre i restanti sono eseguiti dopo il job 0, d'altra parte se abbiamo che i jobs possono essere eseguiti e quindi schedulati prima di job 0 ed il loro processing time totale è W, vuol dire che esiste un sottoinsieme dei w_i la cui somma è W.