1
Problemi difficili

1.1 Introduzione

Molti problemi possono essere risolti con algoritmi efficienti, cioè con algoritmi che necessitano di tempo *polinomiale* per la risoluzione del problema. Vari approcci metodologici, come il *dividi e domina (dìvide et ìmpera)*, la tecnica *greedy* e la *programmazione dinamica*, aiutano nella progettazione di algoritmi efficienti.

Sfortunatamente (o fortunatamente per alcuni aspetti) non tutti i problemi possono essere risolti efficientemente. Esistono dei problemi per i quali non siamo in grado di fornire degli algoritmi che garantiscano il raggiungimento di una risposta in tempo polinomiale. Si noti che per essere efficiente, data una istanza di input, l'algoritmo deve garantire una risposta in tempo polinomiale. E questo deve essere vero per *tutte* le possibili istanze del problema.

Tuttavia, per molti problemi per i quali non siamo in grado di fornire algoritmi efficienti, non siamo nemmeno in grado di provare che tali algoritmi non esistono. Questo è uno dei più grandi problemi irrisolti nel campo dell'informatica ed è noto come la *questione* $\mathcal{P}$-$\mathcal{NP}$. In questo contesto, con $\mathcal{P}$ si indica l'insieme dei problemi per i quali si conoscono degli algoritmi efficienti, cioè algoritmi che trovano una soluzione in tempo polinomiale. Ad esempio, trovare il cammino minimo da una sorgente a una destinazione in un grafo è un problema in $\mathcal{P}$.

Nei successivi paragrafi definiremo formalmente la classe di problemi $\mathcal{NP}$. Per adesso, senza nessuna pretesa di formalità, anticipiamo che i problemi $\mathcal{NP}$ sono quei problemi per i quali è facile verificare che ciò che ci viene presentato come una soluzione del problema è effettivamente una soluzione[1]. Si consideri ad esempio il problema della fattorizzazione di un intero: sia dato un numero intero n ottenuto come il prodotto di due interi $n = a \cdot b$, che però non si conoscono; fattorizzare n significa trovare i due interi a e b usati per ottenere n. Questo problema, all'apparenza semplice, può essere molto difficile da risolvere. Se i due interi a e b sono numeri primi molto grandi, ad esempio numeri con circa 2000 cifre, allora è estremamente difficile trovarli. Tuttavia, anche se a e b sono così grandi, se qualcuno ce li dice, è facile verificare che $n = a \cdot b$, basta fare la moltiplicazione, che, anche per numeri molto grandi, si riesce a fare in maniera efficiente. Il problema della fattorizzazione è un problema in $\mathcal{NP}$.

È abbastanza intuitivo il fatto che se un problema appartiene a $\mathcal{P}$, allora appartiene

[1] L'acronimo $\mathcal{P}$ sta, come probabilmente si intuisce, per "polinomiale" (deterministico). Meno intuitivamente, l'acronimo $\mathcal{NP}$ sta per "non-deterministico polinomiale" e fa riferimento a una definizione alternativa, e equivalente, della classe $\mathcal{NP}$, come verrà spiegato nel paragrafo 1.4.

anche a $\mathcal{NP}$: trovare una soluzione è più "difficile" che verificarla! (Nel seguito daremo una prova di questa intuizione.) Quindi si ha che $\mathcal{P} \subseteq \mathcal{NP}$. Non sappiamo però se $\mathcal{P} = \mathcal{NP}$ oppure se $\mathcal{P} \subset \mathcal{NP}$. Quest'ultima ipotesi è quella più accreditata ma nessuno è stato capace di dimostrarla, quindi rimane la possibilità che $\mathcal{P} = \mathcal{NP}$.

La questione $\mathcal{P}$-$\mathcal{NP}$ è di fondamentale importanza e la sua risoluzione avrà grosse ripercussioni. Se da un lato possiamo "augurarci" che $\mathcal{P}$ sia uguale a $\mathcal{NP}$ per poter utilizzare algoritmi efficienti per tanti problemi, dall'altro siamo "contenti" che problemi difficili esistano. Infatti, alcuni di essi sono alla base dei moderni sistemi crittografici che ci permettono di usare tranquillamente la carta di credito per fare acquisti su Internet (e non solo). In particolare il problema della fattorizzazione, usato nel sistema di cifratura noto come RSA (dai nomi dei suoi ideatori, Rivest, Shamir e Adleman), è alla base del commercio elettronico. Anche chi riuscirà a risolvere la questione $\mathcal{P}$-$\mathcal{NP}$ avrà ripercussioni: diventerà famoso e vincerà un milione di dollari messo in palio da Landon Clay. Che ha anche pensato ad altri problemi meritevoli di tale premio: una descrizione non tecnica dei problemi da un milione di dollari può essere trovata nel saggio di Marcus du Sautoy *L'equazione da un milione di dollari* [12].

1.2 *Problemi decisionali*

Prima di procedere nella discussione è necessario definire la nozione di problema decisionale e spiegare come questa si leghi al problema stesso. In questo paragrafo chiariremo cosa intendiamo per problema, per istanza di un problema e che cosa intendiamo per problema decisionale.

Un problema è una specifica domanda per la quale cerchiamo una risposta. È caratterizzato da una serie di parametri che nella definizione del problema vengono lasciati non specificati, sono, cioè, delle variabili. Oltre alla descrizione dei parametri, che di fatto costituiscono il problema, è necessario specificare che cosa è una soluzione al problema.

Un'istanza di un problema è data da un particolare assegnamento di valori alle variabili che descrivono il problema.

Un algoritmo che risolve un problema è un procedimento che partendo da un'istanza del problema (input) produce una soluzione (output) per quella particolare istanza.

Facciamo degli esempi. Consideriamo il problema del cammino minimo in un grafo $G = (V, E)$ con n nodi[2] $V = \{v_1, v_2, \ldots, v_n\}$ ed m archi $E = \{e_1, e_2, \ldots, e_m\}$. A ogni arco $e = (i, j)$ è associato un costo $c_{i,j}$. Un tale grafo potrebbe rappresentare un insieme di n città (i nodi) fra di loro collegate da strade, collegamenti ferroviari, o altro, e ogni collegamento ha un costo. La domanda che ci poniamo è la seguente: date due specifiche città s e d, quale è il cammino minimo da s a d? In questo problema ci sono vari parametri: il numero n di nodi, il numero m di archi, gli archi, i costi degli archi, il nodo s ed il nodo d. Una soluzione è un cammino da s a d. Quando i parametri non sono specificati il problema è astratto. La Figura 1.1 riporta una rappresentazione del problema astratto: non conosciamo il numero di nodi, non conosciamo il numero di archi, non conosciamo quali nodi sono collegati da archi, non conosciamo i costi e non conosciamo i nodi s e d.

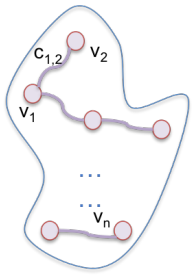

Figura 1.1: Un grafo astratto

[2] Nella descrizione di grafi utilizzeremo "nodi" e "vertici" come sinonimi. Per questo motivo spesso i nodi di un grafo sono indicati con la lettera v.

Un'istanza del problema specifica i parametri. Ad esempio potremmo avere $n = 5$, $V = \{v_1, v_2, v_3, v_4, v_5\}$, $m = 7$ e $E = \{(1,2), (1,3), (1,5), (2,4), (3,4), (3,5), (4,5)\}$, con costi $c_{1,2} = 8, c_{1,3} = 3, c_{1,5} = 12, c_{2,4} = 15, c_{3,4} = 9, c_{3,5} = 5, c_{4,5} = 2$, e $s = v_1$, $d = v_4$. La Figura 1.2 mostra tale istanza del problema, la cui soluzione è il cammino s, v_3, v_5, d, con costo pari a 10. È facile verificare che in questo grafo non c'è un cammino da s a d con costo minore di 10.

Ricordiamo che esistono vari algoritmi efficienti per il problema del cammino minimo in un grafo. L'algoritmo di Dijkstra, che funziona per grafi con costi non negativi, ha complessità $O((|V| + |E|) \log |V|)$ se la coda a priorità usata dall'algoritmo viene implementata con un heap, mentre ha complessità $O(|V|^2)$ se il minimo deve essere cercato in un array non ordinato. L'algoritmo di Bellman-Ford funziona anche con costi negativi e ha complessità $O(|V| \cdot |E|)$. Entrambi gli algoritmi calcolano i cammini minimi da una sorgente data in input a tutti gli altri nodi del grafo. L'algoritmo di Floyd-Warshall, che ha complessità $O(|V|^3)$, calcola i cammini minimi fra tutte le coppie di nodi. Se il grafo non è orientato e i costi sono unitari, una semplice visita BFS che parte dalla sorgente calcola i cammini minimi in tempo $O(|V| + |E|)$.

Consideriamo un altro esempio: il problema del commesso viaggiatore[3], TSP (Travel Salesman Problem). Anche in questo caso abbiamo un grafo con degli archi a cui sono associati dei costi. Questa volta però non ci interessa il cammino minimo fra due città, bensì ci interessa trovare un *giro* che visiti una sola volta tutte le città ritornando nella città di partenza[4] e che abbia costo minimo. Il grafo astratto è uguale a quello del caso precedente: abbiamo un numero non specificato di nodi e un numero non specificato di archi. Una soluzione è un ciclo che contiene (una e una sola volta) tutti i nodi. Anche un'istanza è molto simile a quella del problema dei cammini minimi: specificheremo i nodi, gli archi ed i costi, ma non ci sono sorgente e destinazione. Ad esempio, lo stesso grafo riportato nella Figura 1.2 è un'istanza del problema del commesso viaggiatore, se ignoriamo s e d. In questo caso la soluzione è data da $(v_1, v_3, v_5, v_4, v_2, v_1)$. Si noti che qualunque nodo può essere usato come partenza visto che la soluzione è un giro. In altre parole la soluzione $(v_1, v_3, v_5, v_4, v_2, v_1)$ è equivalente a $(v_3, v_5, v_4, v_2, v_1, v_3)$ e a tutti gli altri giri con la stessa sequenza di nodi.

[3] Il nome deriva dal fatto che un ipotetico commesso viaggiatore debba partire dalla propria città di residenza, visitare tutte le città una sola volta, e ritornare a casa.

[4] Quindi un giro è un ciclo che contiene tutti i nodi. Vedremo più avanti che per grafi non pesati un tale giro viene chiamato ciclo hamiltoniano.

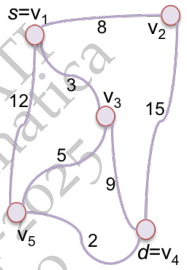

Figura 1.2: Un grafo concreto

Per il problema del commesso viaggiatore non si conoscono algoritmi efficienti; è uno dei problemi *difficili* che vedremo nel prosieguo del capitolo.

I problemi che abbiamo visto come esempi sono problemi di ottimizzazione: si cerca una soluzione che ottimizzi una metrica; negli esempi la metrica è un costo e l'obiettivo è quello di minimizzarlo. Nella maggior parte dei casi pratici abbiamo a che fare con problemi di ottimizzazione. Nei casi in cui non esiste una metrica che misuri la bontà delle soluzioni, siamo interessati a trovare una soluzione qualsiasi; in questi casi si parla di problemi di *ricerca* (di una soluzione).

Per affrontare lo studio dei problemi difficili è però conveniente considerare *problemi decisionali*. Nella versione decisionale di un problema ci chiediamo se (almeno) una soluzione al problema esiste. Pertanto la risposta alla domanda posta dalla versione

decisionale di un problema è un `si` oppure un `no`. Di primo acchito, può sembrare che questa restrizione sia molto forte. In realtà non lo è.

Infatti dato un qualsiasi problema di ottimizzazione possiamo facilmente trasformarlo in un problema decisionale semplicemente introducendo un parametro k e cambiando la domanda in: esiste una soluzione il cui costo è minore uguale (se si cerca un minimo, o maggiore uguale se si cerca un massimo) di k? In questo modo il problema decisionale ci permette di sapere quale è il valore della soluzione ottima. Ad esempio, considerando il problema del commesso viaggiatore, potremmo chiederci se nel grafo G esiste un giro il cui costo è minore o uguale a 100. Considerando l'istanza della Figura 1.2 la risposta sarà `si`. Quindi potremmo chiederci se esiste un giro con costo 50, e così via. Con una ricerca binaria riusciremo a scoprire, usando un numero logaritmico di esecuzioni dell'algoritmo per il problema decisionale, il valore 10 della soluzione ottima.

Inoltre, sebbene non sia stato provato che la versione decisionale è computazionalmente equivalente alla versione di ricerca, per quasi la totalità dei problemi per i quali si conosce un algoritmo efficiente per la versione decisionale del problema, si conosce anche un algoritmo efficiente per trovare una soluzione o la soluzione ottima nel caso di problemi di ottimizzazione. Approfondiremo questo aspetto nella Sezione 1.7.

1.3 *Grandezza del problema*

La difficoltà della risoluzione di un problema è ovviamente funzione della "taglia" dell'istanza da risolvere. Se la taglia è piccola anche un problema "difficile" può essere risolto facilmente. Abbiamo definito l'efficienza rispetto al tempo di esecuzione dell'algoritmo che per essere considerato efficiente deve operare in tempo polinomiale. Polinomiale rispetto a cosa? Rispetto alla taglia o grandezza del problema. Ma come misuriamo la grandezza di un'istanza di un problema? Una possibilità è quella di stabilire una (ragionevole) codifica del problema in termini di stringhe binarie e quindi misurare la lunghezza della stringa che serve per rappresentare le istanze del problema. Ad esempio per specificare un'istanza del problema del commesso viaggiatore dobbiamo codificare il numero di nodi del grafo, gli archi, i costi degli archi. Codificando ognuno di questi elementi con delle sequenze di bit e concatenandoli secondo uno schema che ci permetta di sapere cosa rappresentano i singoli pezzi per poterli poi interpretare correttamente, si ottiene una singola stringa binaria che rappresenta l'istanza del problema.

Ad esempio, consideriamo l'istanza del problema della ricerca del cammino minimo nel grafo della Figura 1.2. Stabiliamo che nella rappresentazione dell'istanza specificheremo prima il numero di nodi, poi l'indice della sorgente seguito da quello della destinazione e di seguito delle triple che specificano archi e relativi costi, cioè una stringa del tipo

$$n, s, d, (v, v, c), (v, v, c), \ldots (v, v, c)\#$$

dove n, s, d e i vari v e c sono numeri e # è un segnalatore di fine stringa. Si noti che una volta specificato il numero di nodi n, ogni singolo nodo può essere implicitamente identificato da un numero fra 1 e n. I simboli di cui abbiamo bisogno sono: le cifre per rappresentare i numeri, la virgola per separare gli elementi, le parentesi tonde per

raggruppare gli archi con incluso il costo — in realtà potremmo anche fare a meno di queste parentesi, ma per facilità di lettura della codifica le utilizziamo — e un carattere di fine stringa per indicare che la rappresentazione è finita. Nel caso specifico quindi avremo la sequenza

$$5, 1, 4, (1, 2, 8), (1, 3, 3), (1, 5, 12), (2, 4, 15), (3, 4, 9), (3, 5, 5), (4, 5, 2)\#$$

In realtà tale sequenza la vogliamo rappresentare in binario, quindi dobbiamo anche stabilire una rappresentazione binaria per i simboli che utilizziamo[5]. Poichè ci servono 14 simboli in totale (10 cifre, le due parentesi tonde, la virgola e il carattere di fine stringa), possiamo utilizzare sequenze di 4 bit: i valori da 0000 a 1001 rappresentano le cifre, 1010 rappresenta "(", 1011 rappresenta ")", 1100 rappresenta "," e 1111 rappresenta il carattere di fine stringa "#". Le strighe 1101 e 1110 non vengono utilizzate, quindi la rappresentazione binaria dell'istanza della Figura 1.2 è:

0101 1100 0001 1100 0100 1100 1010 0001 1100 0010 1100 1000 1011 1100 1010
0001 1100 0101 1100 0001 0010 1011 1100 1010 0010 1100 0100 1100 0001 0101
1011 1100 1010 0011 1100 0100 1100 1001 1011 1100 1010 0011 1100 0101 1100
0101 1011 1100 1010 0100 1100 0101 1100 0010 1011 1111

La lunghezza di tale stringa è 224. Si noti come il costo degli archi $(1, 5)$ e $(2, 4)$, rispettivamente, 12 e 15, siano stati rappresentati con 2 cifre (quindi 8 bit). Questo perchè nella nostra codifica possiamo rappresentare solo le singole cifre; quindi per rappresentare valori più grandi, dobbiamo codificare i numeri rappresentando la sequenza delle singole cifre. Questo aspetto è importante, come spiegheremo fra un po'. Si noti che avremmo potuto anche usare la rappresentazione binaria del valore per specificare il costo degli archi, ma la sostanza del discorso non sarebbe cambiata: avremmo avuto bisogno di più bit per rappresentare valori più grandi.

Assumendo di usare delle codifiche ragionevoli[6], la lunghezza di tale stringa è proporzionale (quindi legata polinomialmente) ai parametri più significativi del problema. Ad esempio, nel caso del problema TSP, il parametro più significativo è il numero di nodi n: tale numero determina il massimo numero di archi e costi possibili (n^2) ed è legato polinomialmente alla lunghezza della stringa binaria che rappresenta il problema. Pertanto per semplificare la trattazione considereremo come misura della grandezza del problema il parametro più significativo.

Affinchè la lunghezza della stringa che rappresenta il problema sia polinomialmente legata al parametro più significativo è però necessario che i valori rappresentabili con un numero di simboli variabili, come nel caso dei costi degli archi nell'esempio precedente, siano anche essi legati polinomialmente al parametro più significativo. Quindi faremo implicitamente tale assunzione. Approfondiremo ulteriormente questo aspetto più tardi, quanndo parleremo di pseudopolinomialità.

Ovviamente la determinazione del parametro più significativo dipende dal problema. Per problemi che riguardano grafi è il numero di nodi. Nel seguito vedremo dei problemi che coinvolgono delle formule booleane; in tal caso il parametro più significativo, rispetto alla grandezza, è il numero di variabili.

[5] Codifica dei simboli:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
(	1010
)	1011
,	1100
	1101
	1110
#	1111

[6] Ad esempio una codifica unaria è da escludere in quanto genererebbe delle stringhe irragionevolmente lunghe. Per approfondimenti si veda il Capitolo 34 di [8].

1.4 Classi P e NP

Per la definizione delle classi $\mathcal{P}$ e $\mathcal{NP}$ si considerano problemi formulati in versione decisionale. La motivazione è legata al fatto che i problemi decisionali possono essere messi in corrispondenza con i linguaggi formali: tutte le stringhe che rappresentano istanze che ammettono una soluzione fanno parte del corrispondente linguaggio mentre tutte le altre stringhe no. Quindi un problema decisionale è un linguaggio formale. Trattare i problemi attraverso linguaggi formali ha dei vanataggi. Non approfondiremo questo aspetto ma, come esempio, nel paragrafo 1.7 c'è un ragionamento che mostra l'utilità dei linguaggi formali nel contesto della questione $\mathcal{P}$-$\mathcal{NP}$. Un ulteriore esempio è la possibilità di definire facilmente il "complemento" di un problema, semplicemente invertendo la risposta.

Definizione 1.4.1 *La classe di problemi indicata con $\mathcal{P}$ è l'insieme dei problemi per i quali conosciamo un algoritmo efficiente, cioè polinomiale.*

Dunque, considerato che la difficoltà della versione decisionale è paragonabile a quella del problema stesso, per un problema in $\mathcal{P}$ *trovare* una soluzione costa tempo polinomiale. Per un problema come TSP, invece, non si conosce un algoritmo polinomiale e non si sa nemmeno che un tale algoritmo non esiste. Cioè non si sa se il problema TSP appartiene a $\mathcal{P}$. In altre parole non sappiamo se trovare una soluzione per TSP sia difficile o meno. Tuttavia se ci viene data una particolare soluzione, è facile *verificare* che il suo costo è minore della soglia stabilita dal problema. La classe $\mathcal{NP}$ è l'insieme dei problemi per i quali la verifica di una potenziale soluzione può essere fatta in tempo polinomiale.

Per rendere più formale la discussione occore specificare meglio cosa significa verificare una soluzione. Come abbiamo già detto l'input al problema viene codificato con una stringa binaria s. Denoteremo la lunghezza della stringa con $|s|$. Un problema decisionale X può essere identificato con l'insieme delle stringhe binarie che corrispondono agli input per i quali la risposta al problema è "`si`". Un algoritmo A per X prende in input una stringa s e risponde "`si`" oppure "`no`". Diremo anche che A risolve X se $A(s) =$`si` se e solo se $s \in X$. Il tempo di esecuzione di A è polinomiale se esiste un polinomio $p()$ tale che per qualsiasi stringa di input s, $A(s)$ termina in al massimo $O(p(|s|)$ passi.

Un algoritmo permette di risolvere un problema. Se volessimo invece solo essere capaci di verificare una potenziale soluzione? Per formalizzare tale nozione, avremo bisogno di un *certificato*. Un verificatore quindi prende in input due parametri, un'istanza del problema s e un certificato c. L'output del verificatore può essere `ok`, se il certificato costituisce una prova del fatto che s ammette soluzioni, oppure `nok` se invece il certificato non "convince".

Definizione 1.4.2 *Un algoritmo V è un "verificatore di esistenza" per un problema X se per ogni stringa $s \in X$, cioè per ogni istanza del problema che ammette soluzioni, esiste una stringa c (certificato), di lunghezza polinomiale in $|s|$, tale che $V(s, c) =$ `ok`, in tempo polinomiale in $|s|$.*

Si noti che il verificatore non ha l'obiettivo di stabilire se la stringa s appartenga a X o meno. Invece, ha come obiettivo quello di "farsi convincere" dal certificato c che la stringa di input s sia effettivamente in X. Quindi l'approccio del verificatore potrebbe essere spiegato, informalmente, come segue: non credo che l'istanza s abbia una soluzione a meno che il certificato c non costituisce una prova di questo fatto. Si noti anche che un verificatore è efficiente per definizione (un verificatore non efficiente non è utile). Un certificato, tipicamente, è una soluzione e il verificatore deve solo "verificare" che la soluzione descritta dal certificato sia effettivamente valida. Quindi è abbastanza semplice fornire un verificatore (di esistenza).

Definizione 1.4.3 *La classe $\mathcal{NP}$ è la classe dei problemi per i quali esiste un verificatore di esistenza di una soluzione.*

La "N" in $\mathcal{NP}$ deriva dal fatto che la verifica di una soluzione può anche essere vista come un approccio non deterministico alla risoluzione del problema. Cioè, una definizione alternativa della classe $\mathcal{NP}$ è quella che definisce i problemi $\mathcal{NP}$ come quei problemi che possono essere risolti con algoritmi polinomiali ma *non-deterministici*. Quindi $\mathcal{NP}$ sta per tempo *polinomiale non-deterministico*. Le due definizioni sono equivalenti.

Lemma 1.4.4 $\mathcal{P} \subseteq \mathcal{NP}$.

DIMOSTRAZIONE. Informalmente il lemma è vero in quanto verificare una soluzione è più facile che trovarla. Formalmente dobbiamo provare che per un qualsiasi problema $X \in \mathcal{P}$ esiste un verificatore in tempo polinomiale. Poichè però $X \in \mathcal{P}$ è facile trovare un tale verificatore: è l'algoritmo efficiente che risolve X. Tale algoritmo può semplicemente ignorare il certificato che viene presentato al verificatore e risolvere il problema per poter decidere se esiste una soluzione. Si ricordi che dobbiamo solo stabilire se una soluzione esiste oppure no, e il certificato serve solo come prova dell'esistenza di una soluzione. Quindi trovarne un'altra, ignorando quella proposta, è perfettamente lecito. □

Quando un problema non è in $\mathcal{P}$ per mostrare che è in $\mathcal{NP}$ dovremo fornire un verificatore. Il verificatore si limiterà a controllare il certificato, quindi in genere è molto semplice. Ad esempio per il problema TSP basterà verificare che la potenziale soluzione, cioè un giro del grafo, sia effettivamente un giro e abbia un costo minore o uguale alla soglia stabilita dal problema.

Ci sono tanti problemi in $\mathcal{NP}$ per i quali non si conoscono algoritmi efficienti e nemmeno si sa dire che tali algoritmi non esistono. Se tutti i problemi in $\mathcal{NP}$ ammettessero un algoritmo efficiente si avrebbe $\mathcal{P} = \mathcal{NP}$; se invece ci fosse qualche problema che non può essere risolto in modo efficiente, allora si avrebbe $\mathcal{P} \subset \mathcal{NP}$. Stabilire se $\mathcal{P} = \mathcal{NP}$ oppure se $\mathcal{P} \subset \mathcal{NP}$ è un problema aperto.

Nel seguito studieremo alcuni problemi per i quali non si sa se esiste o meno una soluzione efficiente. Trovare una soluzione efficiente per uno di questi problemi, o dimostrare che non esiste una tale soluzione, risolverebbe la questione $\mathcal{P}$-$\mathcal{NP}$. In particolare i problemi di cui tratteremo sono i seguenti:

- CircuitSat: Dato un circuito booleano, stabilire se esiste un assegnamento dell'input che produce il valore vero.
- Sat: Data una formula booleana, espressa come la congiunzione di clausole stabilire se esiste un assegnamento delle variabili che rende vera la formula.
- 3Sat: Un caso speciale di Sat in cui tutte le clausole hanno esattamente 3 termini.
- IndependentSet: Dato un grafo e un intero k, stabilire se esiste un insieme di nodi indipendenti (cioè senza archi che li collegano) di taglia almeno k.
- VertexCover: Dato un grafo e un intero k, stabilire se esiste un insieme di nodi che "copre" il grafo, cioè tale che tutti gli archi sono incidenti in almeno un nodo dell'insieme.
- SetCover: Una generalizzazione di VertexCover in cui vogliamo ricoprire un insieme arbitrario di oggetti utilizzando un insieme di insiemi.
- SetPacking: Una generalizzazione di IndependentSet in cui vogliamo trovare sottoinsiemi che non si intersecano fra loro.
- GraphColoring: Dato un grafo e un intero k, stabilire se i nodi del grafo possono essere colorati con al più k colori facendo in modo che due nodi collegati da un arco non abbiano lo stesso colore.
- HamCycle: Dato un grafo, stabilire se esiste un ciclo hamiltoniano.
- TSP: Dato un grafo e un valore C, stabilire se è possibile visitare tutto il grafo con un "giro" di costo al massimo C.

Per ognuno dei problemi sopra elencati è facile fornire un verificatore, quindi tutti questi problemi appartengono a $\mathcal{NP}$.

1.5 *Riduzioni*

Lo studio della questione $\mathcal{P}$-$\mathcal{NP}$ riguarda la classificazione dei problemi in problemi "facili", cioè che possono essere risolti in tempo polinomiale, e problemi "difficili", per i quali non si conosce un algoritmo polinomiale. Un punto cruciale è quello di poter dire che un problema A è "più facile" di un problema B, o equivalentemente, che un problema B è "più difficile" di un problema A. Per formalizzare affermazioni di questo tipo, sono molto utili le *riduzioni*. Informalmente, un problema A può essere ridotto a un problema B se possiamo usare B per risolvere A. In questo senso B è "più difficile" di A. Affinchè la riduzione sia utile è necessario che la difficoltà del processo di utilizzo di B per risolvere A non sia troppo grande: se la difficoltà della riduzione supera la difficoltà della risoluzione di B, potrebbe non convenire utilizzare la riduzione. Nel contesto della questione $\mathcal{P}$-$\mathcal{NP}$, è necessario che la riduzione sia efficiente, cioè che il lavoro necessario per utilizzare B per risolvere A sia polinomiale. In questo modo la risoluzione, in tempo polinomiale[7], di A attraverso B, usando quindi la riduzione,

[7] Si noti che ciò che interessa è la risolvibilità del problema in tempo polinomiale.

dipende solo dall'efficienza della soluzione per B.

Per definire formalmente la nozione di riducibilità, supporremo di poter disporre di un oracolo per la risoluzione dei problemi. L'oracolo, schematizzato nella Figura 1.3, è capace di risolvere qualsiasi problema in tempo costante.

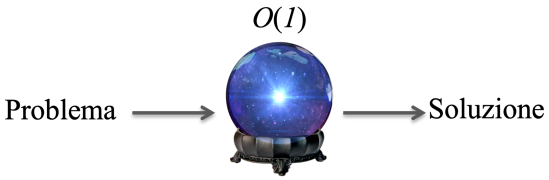

Figura 1.3: Oracolo: risolve qualsiasi problema in tempo $O(1)$

Definizione 1.5.1 *Un problema A può essere ridotto a B, se è possibile risolvere una qualsiasi istanza di A usando tempo polinomiale e sfruttando un oracolo che risolve B. Scriveremo* $A \leq_P B$.

Osserviamo che l'oracolo può essere chiamato più di una volta; spesso basterà una sola invocazione dell'oracolo, ma in generale è possibile sfruttarlo un numero polinomiale di volte. Quando si invoca l'oracolo è necessario fornire in qualche modo l'input del problema da risolvere e, successivamente, è necessario leggere l'output dell'oracolo. Nel costo della riduzione dobbiamo calcolare il tempo necessario sia a scrivere l'input per l'oracolo sia a leggere l'output dell'oracolo.

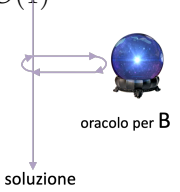

Figura 1.4: Risoluzione di A tramite una riduzione a B

Dalla Definizione 1.5.1 si deduce immediatamente il seguente risultato:

Lemma 1.5.2 *Assumiamo che* $A \leq_P B$. *Se B può essere risolto in tempo polinomiale allora anche A può essere risolto in tempo polinomiale.*

Dimostrazione. Poichè $A \leq_P B$, possiamo dare un algoritmo che risolve A sfruttando la riduzione: questo algoritmo fa esattamente quello che viene fatto nella riduzione e al posto dell'oracolo, che non esiste, usa un algoritmo efficiente che risolve B, che invece esiste, in quanto B può essere risolto in tempo polinomiale. Il costo totale di questo algoritmo sarà comunque polinomiale in quanto la riduzione richiede al massimo un numero polinomiale di passi, e anche se ognuno di questi passi fosse una invocazione dell'oracolo, quindi un utilizzo dell'algoritmo che risolve B, costerebbe comunque un tempo polinomiale: moltiplicando un polinomio per un altro polinomio si ottiene comunque un polinomio. □

Il Lemma 1.5.2 è molto utile per trovare soluzioni efficienti a nuovi problemi riducendoli a problemi per i quali già si conosce una soluzione efficiente. Ad esempio il problema del matching bipartito può essere ridotto a un problema di massimo flusso (vedi il Capitolo 7 di KT2014 [20]).

La Definizione 1.5.1 implica anche un risultato simmetrico a quello del Lemma 1.5.2:

Lemma 1.5.3 *Assumiamo che* $A \leq_P B$. *Se A non può essere risolto in tempo polinomiale allora anche B non può essere risolto in tempo polinomiale.*

In realtà i Lemmi 1.5.2 e 1.5.3 sono equivalenti, e rappresentano solo due modi diversi di vedere la stessa cosa. Il Lemma 1.5.3 fornisce un modo più diretto per provare che

alcuni problemi sono "difficili". Infatti se A è un problema difficile e si prova che $A \leq_P B$ allora si è provato che anche B è un problema difficile.

Poichè di fatto non sappiamo se i problemi difficili che studiamo possano o meno essere risolti in modo efficiente, la riduzione è uno strumento per classificarli in base alla difficoltà relativa fra di essi.

Non è difficile vedere che la relazione $\leq_P$ è *transitiva*.

Lemma 1.5.4 *Se $A \leq_P B$ e $B \leq_P C$ allora $A \leq_P C$.*

DIMOSTRAZIONE. Per ridurre A a C è sufficiente "passare" per B. Cioè data l'istanza di A da risolvere, poichè $A \leq_P B$ si ha che avendo un oracolo per risolvere B si può risolvere A. Anche non avendo un oracolo per B (ma solo uno per C), si ha che poichè $B \leq_P C$ è possibile risolvere B usando l'oracolo per C. Quindi si può comunque risolvere A usando l'oracolo per C, il tutto sempre usando tempo polinomiale nei vari passaggi che non coinvolgono l'oracolo. □

1.5.1 INDEPENDENTSET *e* VERTEXCOVER

Prima di tutto definiamo formalmente i problemi INDEPENDENTSET e VERTEXCOVER e vediamo degli esempi. Successivamente vedremo che essi sono equivalenti dal punto di vista computazionale, cioè l'uno può essere ridotto all'altro.

Dato un grafo $G = (V, E)$, un insieme di nodi $S \subseteq V$ è *indipendente* se in E non ci sono archi fra i nodi di S.

Problema 1.5.5 INDEPENDENTSET*: Dato un grafo G e un intero k, G contiene un insieme indipendente di taglia almeno k?*

È facile trovare insiemi indipendenti di piccole dimensioni. Infatti ogni singolo nodo è un insieme indipendente di taglia 1. Per sapere se esiste un insieme indipendente di taglia 2 basta controllare tutte le coppie di nodi e vedere se almeno una non è collegata da un arco o equivalentemente se il grafo è completo, cioè tutte le coppie di nodi sono collegate da un arco (è sufficiente controllare il numero di archi: se è minore di $n(n-1)/2$ allora esiste un insieme indipendente di taglia 2, se è uguale a $n(n-1)/2$ allora il grafo è completo e quindi non esiste un insieme indipendente di taglia 2).

Come esempio consideriamo il grafo riportato in Figura 1.5.

I nodi $\{2, 4\}$ formano un insieme indipendente di taglia 2. I nodi $\{1, 3, 5\}$ formano un insieme indipendente di taglia 3. Non è difficile verificare che in questo grafo non esistono insiemi indipendenti di taglia più grande: basta controllare che tutti gli insiemi di taglia 4 hanno almeno un arco che collega una coppia di nodi.

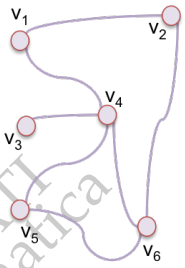

Figura 1.5: Un grafo con un insieme indipendente di taglia 3

Al crescere della taglia però diventa sempre più difficile trovare insiemi indipendenti o verificare che non esistono. Infatti il numero di sottoinsimi di taglia k è $\binom{n}{k}$, quindi per controllarli tutti abbiamo bisogno di tempo esponenziale.

Abbiamo posto il problema come un problema decisionale: dato un grafo *decidere* (stabilire) se in quel grafo esiste un insieme indipendente di taglia almeno k. Lo stesso problema potremmo vederlo come un problema di ottimizzazione: dato un grafo, trovare l'insieme indipendente più grande.

Passiamo adesso al problema VertexCover. Dato un grafo $G = (V, E)$ un insieme $S \subseteq V$ è un insieme di vertici che ricopre (gli archi di) G se ogni arco $e \in E$ ha almeno uno dei suoi due vertici in S. Si noti che ad essere "coperti" sono gli archi del grafo e che ogni vertice del grafo "copre" tutti gli archi incidenti su di esso. È facile trovare degli insiemi di vertici ricoprenti: ad esempio $S = V$ è un insieme ricoprente. È più difficile trovare insiemi ricoprenti più piccoli. Considerando di nuovo il grafo della Figura 1.5 si ha che l'insieme di nodi $\{1, 2, 3, 5\}$ è un insieme ricoprente di taglia 4 e che l'insieme di nodi $\{2, 4, 5\}$ è un insieme ricoprente di taglia 3. Il problema di ottimizzazione richiederebbe l'individuazione dalla taglia minima di un insieme ricoprente. Si può verificare che, in questo grafo, non esistono insiemi ricoprenti di taglia 2. Quindi l'insieme $\{2, 4, 6\}$ è un insieme ricoprente di taglia minima. Nella versione decisionale del problema ci chiediamo:

Problema 1.5.6 VertexCover*: Dato un grafo G e un intero k, G contiene un insieme ricoprente di taglia al massimo k?*

Non si conoscono algoritmi efficienti per risolvere IndependentSet e VertexCover. Possiamo però provare che questi due problemi sono equivalenti nel senso che uno può essere ridotto all'altro, cioè si ha sia che IndependentSet$\leq_P$VertexCover ma anche che VertexCover$\leq_P$IndependentSet.

Lemma 1.5.7 *Sia $G = (V, E)$ un grafo. Un inseme $S \subset V$ è un insieme indipendente se e solo se $V \setminus S$ è un insieme ricoprente.*

Dimostrazione. Sia S un insieme indipendente. Consideriamo un arco qualsiasi $e = (u, v)$. Poichè S è indipendente u e v non possono entrambi appartenere ad S. Quindi almeno uno dei due deve appartenere a $V \setminus S$. Poichè questo è vero per tutti gli archi si ha che $V \setminus S$ è un insieme ricoprente.

Assumiamo adesso che $V \setminus S$ sia un insieme ricoprente. Consideriamo due nodi qualsiasi u, v dell'insieme S. Non può esistere l'arco $e = (u, v)$; infatti se esistesse tale arco esso non sarebbe "coperto" da nessun nodo di $V \setminus S$ contraddicendo il fatto che $V \setminus S$ è un insieme ricoprente. Poichè ciò è vero per qualunque coppia di nodi di S, concludiamo che S è un insieme indipendente. □

Usando il lemma precedente possiamo provare che i problemi IndependentSet e VertexCover sono di fatto equivalenti.

Lemma 1.5.8 IndependentSet$\leq_P$VertexCover.

Dimostrazione. Per provare la riduzione dobbiamo far vedere come risolvere IndependentSet sfruttando un oracolo che risolve VertexCover, ed usando al massimo tempo polinomiale.

Consideriamo un'istanza del problema IndependentSet. Tale istanza è costituita da un grafo G e un intero k: dobbiamo decidere se G ha un insieme indipendente di taglia almeno k.

Abbiamo a disposizione un oracolo per VertexCover. Pertanto, per usarlo, dobbiamo costruire un'istanza di VertexCover. L'istanza deve essere costruita in modo tale che

la soluzione per VertexCover fornita dall'oracolo ci permetta di trovare una soluzione per l'istanza di IndependentSet. Un'istanza per VertexCover è costituita anch'essa da un grafo G' e da un intero k': l'oracolo dirà se G' ha un insieme ricoprente di taglia almeno k'.

Per creare l'istanza da dare in input all'oracolo usiamo $G' = G$ e $k' = n - k$, dove n è il numero di nodi. Questo passaggio richiede tempo polinomiale (di fatto costante in quanto non dobbiamo fare quasi nulla, se non calcolare k') e per "scrivere" l'input per l'oracolo è sufficiente tempo lineare in quanto occorre semplicemente copiare G e scrivere k'.

Preparato l'input, l'oracolo ci dirà se $G' = G$ contiene un insieme ricoprente con al massimo $k' = n - k$ nodi. Sia $r \in \{\texttt{si}, \texttt{no}\}$ la risposta dell'oracolo. Per il Lemma 1.5.7 si ha che r è anche la risposta alla domanda "Esiste un insieme indipendente di almeno $n - k' = k$ nodi?". Quindi r è anche la risposta all'istanza del problema IndependentSet. □

Lemma 1.5.9 VertexCover$\leq_P$IndependentSet.

Dimostrazione. Questa dimostrazione è simile a quella del lemma precedente. La si svolga come esercizio. □

Riassumendo, anche se per nessuno dei due problemi sappiamo se esiste o meno una soluzione efficiente, sappiamo che se riusciamo a risolvere in modo efficiente uno dei due possiamo risolvere efficientemente anche l'atro. E anche che se dimostriamo che uno dei due problemi non può essere risolto efficientemente, avremo dimostrato che anche l'altro non può essere risolto efficientemente.

1.5.2 SetCover

Consideriamo adesso un altro problema: SetCover. Il problema del ricoprimento degli archi può essere visto come un caso speciale di un problema più generale, SetCover, in cui vogliamo ricoprire un insieme arbitrario di oggetti utilizzando un insieme di insiemi. Più formalmente:

Problema 1.5.10 SetCover*: dato un insieme U di n elementi, una collezione $S_1, \ldots, S_m$ di sottoinsiemi di U e un intero k, esiste una collezione di al massimo k fra gli m sottoinsiemi $S_1, \ldots, S_m$ tali che l'unione di tali sottoinsiemi è uguale a U?*

La Figura 1.6 mostra un esempio con $n = 9$, $U = \{a, b, c, d, e, f, g, h, i\}$, $m = 7$ e $S_1 = \{a, b, c\}$, $S_2 = \{d, e\}$, $S_3 = \{g, h, i\}$, $S_4 = \{d, f, g, i\}$, $S_5 = \{d, f\}$, $S_6 = \{a, d, f\}$, e $S_7 = \{e, c, h\}$. Un insieme ricoprente è $\{S_1, S_4, S_7\}$.

Un esempio concreto in cui potremmo modellare la realtà con un'istanza del problema SetCover è il seguente: in un importante sito turistico un'agenzia organizza escursioni giornaliere alle quali partecipano molti visitatori provenienti da tutto il mondo. L'agenzia ha a disposizione delle guide ognuna delle quali sa parlare un certo numero di lingue. Per ogni giro turistico l'agenzia vuole selezionare il minimo numero di guide da impiegare in modo tale che per ogni partecipante ci sia almeno una

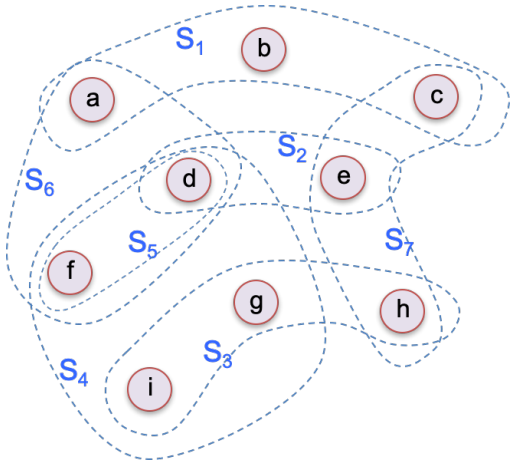

Figura 1.6: Grafo di esempio per SetCover

guida che parli la sua lingua. In questo caso l'insieme U è l'insieme di tutte le lingue parlate dal gruppo di partecipanti all'escursione mentre ogni guida corrisponde a un sottoinsieme $S_i \subseteq U$, dato dalle lingue parlate dalla guida.

VertexCover è un caso speciale di SetCover: nel caso di VertexCover, l'insieme U è dato dall'insieme degli archi, mentri gli insiemi S_i, uno per nodo, sono dati dagli archi incidenti nel nodo stesso (un nodo ricopre tutti gli archi ad esso incidenti). Pertanto non sorprende il fatto che:

Lemma 1.5.11 VertexCover$\leq_P$SetCover.

Dimostrazione. Per provare la riduzione dobbiamo far vedere come risolvere VertexCover sfruttando un oracolo che risolve SetCover, ed usando al massimo tempo polinomiale.

Consideriamo un'istanza del problema VertexCover. Tale istanza è costituita da un grafo G e un intero k: dobbiamo decidere se G ha un insieme ricoprente di al massimo k nodi.

Abbiamo a disposizione un oracolo per SetCover. Pertanto, per usarlo, costruiamo un'istanza per SetCover; una tale istanza è costituita da un insieme U, da m sottoinsiemi di U, $S_1, \ldots, S_m$ e da un intero k'.

La specifica istanza che costruiamo è la seguente. L'insieme U è costituito da tutti gli archi di G, cioè $U = E$. Inoltre per ogni vertice $i \in V$, consideriamo l'insieme $S_i \subseteq U$, costituito da tutti gli archi di E incidenti su i. Infine, $k' = k$. La costruzione di questa istanza può essere chiaramente fatta in tempo polinomiale.

L'istanza di SetCover che abbiamo appena costruita ammette un insieme ricoprente di al massimo $k' = k$ degli insiemi $S_1, \ldots, S_n$ se e solo se il grafo G ha un insieme ricoprente di al massimo $k = k'$ nodi. Infatti, sia $S_{i_1}, \ldots, S_{i_\ell}$ un insieme ricoprente per U con $\ell \leq k'$, allora si ha che ogni elemento di U è coperto da uno degli insiemi S_{i_j}. Poichè $U = E$ e per come abbiamo definito gli insiemi S_{i_j} si ha che $\{i_1, \ldots, i_\ell\}$ è un insieme ricoprente di G di taglia $\ell \leq k' = k$. Analogamente, sia $\{i_1, \ldots, i_\ell\}$ un insieme ricoprente per G, con $\ell \leq k$, allora $S_{i_1}, \ldots, S_{i_\ell}$ è un insieme ricoprente per U, con $\ell \leq k' = k$.

Pertanto per risolvere il problema VertexCover su G possiamo semplicemente costruire l'istanza di SetCover come descritto prima e usare l'oracolo per risolvere questa istanza. La risposta data al problema SetCover è anche la risposta al problema VertexCover. □

Notiamo che sia nella prova appena fatta sia nelle prove fatte per le precedenti riduzioni, sebbene la definizione di riduzione permetta l'utilizzo dell'oracolo fino a un numero polinomiale di volte, l'oracolo viene invocato esattamente una volta. In tutti i casi, partendo da un'istanza del problema da risolvere, abbiamo creato un'istanza del problema per il quale abbiamo l'oracolo e abbiamo usato l'oracolo una sola volta per risolvere la nuova istanza, la cui soluzione è anche una soluzione del problema originario. Questo modo di procedere è abbastanza comune e sarà così per tutte le riduzioni che vedremo. Si rammenti però che in una riduzione è lecito usare l'oracolo fino a un numero polinomiale di volte.

1.5.3 SetPacking

Come VertexCover può essere generalizzato in SetCover, così IndependentSet può essere generalizzato nel seguente problema:

Problema 1.5.12 SetPacking*: Dato un insieme U di elementi, una collezione $S_1, \ldots, S_m$ di sottoinsiemi di U, e un intero k, esiste una collezione di almeno k di questi sottoinsiemi tali che nessuno di loro si interseca con un altro?*

Non dovrebbe sorprendere (omettiamo la dimostrazione) che:

Lemma 1.5.13 IndependentSet$\leq_P$SetPacking.

1.5.4 Sat *e* 3Sat

Sat e 3Sat sono 2 problemi simili formulati in termini di funzioni booleane. Essi modellano un ampio insieme di problemi in cui sono in gioco variabili decisionali a cui deve essere assegnato un valore nel rispetto di determinati vincoli.

Sat è definita come una formula booleana su un insieme di n variabili booleane $x_1, \ldots, x_n$, espressa come l'and di or. Più precisamente, la formula è espressa come

$$\phi = C_1 \cdot C_2 \cdot \ldots \cdot C_k$$

dove ogni *clausola* $C_i = t_{i_1} + t_{i_2} + \ldots + t_{i_\ell}$, ed ogni *letterale* t_{i_s} è una delle variabili x_j oppure la sua negazione $\bar{x}_j$.

Il problema della soddisfacibilità di formule booleane è definito come segue.

Problema 1.5.14 Sat*: Data una formula booleana ϕ, espressa come l'*and *di* or *su un insieme di variabili $\{x_1, \ldots, x_n\}$, esiste un'assegnazione di valori delle variabili che rende vera la formula?*

Consideriamo come esempio la seguente formula

$$\phi_1 = (x_1 + x_4) \cdot \overline{x_3} \cdot (x_1 + \overline{x_2} + x_3 + \overline{x_4}) \cdot (x_2 + x_3) \cdot (\overline{x_1} + x_2 + x_3).$$

La formula ϕ è composta da 5 clausole $\phi_1 = C_1 \cdot C_2 \cdot C_3 \cdot C_4 \cdot C_5$, con:

$$\begin{aligned} C_1 &= x_1 + x_4 \\ C_2 &= \overline{x_3} \\ C_3 &= x_1 + \overline{x_2} + x_3 + \overline{x_4} \\ C_4 &= x_2 + x_3 \\ C_5 &= \overline{x_1} + x_2 + x_3 \end{aligned}$$

L'assegnamento $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ e $x_4 = 1$, ha come conseguenza $\phi = 1$. La formula $\phi_2 = \overline{x_1} \cdot (x_1 + x_2) \cdot \overline{x_2}$, invece, non ammette assegnamenti che la rendono vera.

Esiste un caso speciale del problema SAT che di fatto è equivalente alla formulazione generale. Tale caso speciale si ha quando tutte le clausole sono composte da esattamente 3 letterali. Ad esempio la formula

$$\phi_3 = (x_1 + x_2 + x_4) \cdot (x_2 + \overline{x_3} + x_4) \cdot (\overline{x_1} + \overline{x_2} + x_3) \cdot (\overline{x_1} + \overline{x_3} + \overline{x_4})$$

è composta da 4 clausole, ognuna avente esattamente 3 letterali:

$$\begin{aligned} C_1 &= x_1 + x_2 + x_4 \\ C_2 &= x_2 + \overline{x_3} + x_4 \\ C_3 &= \overline{x_1} + \overline{x_2} + x_3 \\ C_4 &= \overline{x_1} + \overline{x_3} + \overline{x_4} \end{aligned}$$

Problema 1.5.15 3SAT*: Data una formula booleana $C_1 \cdot C_2 \cdot \ldots \cdot C_k$ su un insieme di variabili $\{x_1, \ldots, x_n\}$, con ognuna delle clausole C_i avente esattamente 3 letterali, esiste un'assegnazione di valori delle variabili che rende vera la formula?*

Nel caso della formula ϕ_3 la risposta a tale domanda è affermativa: un assegnamento che rende vera la formula è $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$.

La definizione del problema 3SAT è la stessa del problema SAT fatta eccezione per il vincolo dei 3 letterali per clausola. Quindi $3\text{SAT} \leq_P \text{SAT}$, in quanto 3SAT è un caso particolare di SAT. Meno intuitivo è il fatto che che una qualsiasi istanza di SAT può essere trasformata in una equivalente istanza di 3SAT: l'Esercizio 6 chiede di provare questo fatto. Dunque si ha che $\text{SAT} \leq_P 3\text{SAT}$, pertanto i due problemi sono equivalenti.

Proviamo adesso che 3SAT può essere ridotto a INDEPENDENTSET.

Lemma 1.5.16 $3\text{SAT} \leq_P$ INDEPENDENTSET.

DIMOSTRAZIONE. Abbiamo a disposizione un oracolo per risolvere INDEPENDENTSET e vogliamo risolvere 3SAT. Sia $\phi = C_1 \cdot C_2 \cdot \ldots \cdot C_k$, l'istanza di 3SAT in cui ogni clausola C_i è costituita da 3 letterali delle variabili $x_1, \ldots, x_n$.

Per trovare un assegnamento alle variabili che renda vera la formula dobbiamo individuare in ognuna delle clausole un particolare letterale al quale deve essere dato

il valore vero (ne serve almeno uno per clausola). La cosa potrebbe sembrare facile in quanto per rendere vera una clausola basta dare il valore vero anche a un solo dei tre letterali. Il problema è che letterali in clausole diverse potrebbero essere in conflitto fra di loro. Ad esempio una clausola potrebbe contenere x_i e un'altra clausola potrebbe contenere $\bar{x}_i$. In questo caso non possiamo scegliere entrambi questi letterali per rendere vere le due clausole in quanto non c'è modo di renderli entrambi veri. Tuttavia se evitando tutti i letterali che vanno in conflitto riusciamo comunque ad individuarne uno da poter rendere vero per ogni clausola, allora abbiamo trovato un assegnamento che rende vera la formula.

Quanto detto poc'anzi è utile per trasformare l'istanza di 3SAT in una equivalente istanza di INDEPENDENTSET, anche se la costruzione del grafo non è immediata. Lo costruiamo nel seguente modo. Il grafo contiene $3k$ nodi, 3 per ognuna delle k clausole, ed ogni nodo in un gruppo di 3 corrisponde a uno dei letterali della clausola.

Più formalmente, per $j = 1, \ldots, k$, costruiamo 3 vertici, $v_{j,1}, v_{j,2}$ e $v_{j,3}$. I tre vertici corrispondenti alla stessa clausola saranno uniti da 3 archi per formare un triangolo, come mostrato nella Figura 1.7. L'intera formula conterrà quindi k di questi triangoli, uno per ogni clausola. Usando come esempio la formula ϕ_3 vista in precedenza si avrebbe il grafo mostrato nella Figura 1.8.

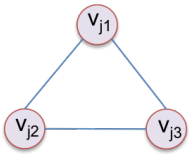

Figura 1.7: Il triangolo di nodi che rappresenta la clausola C_j

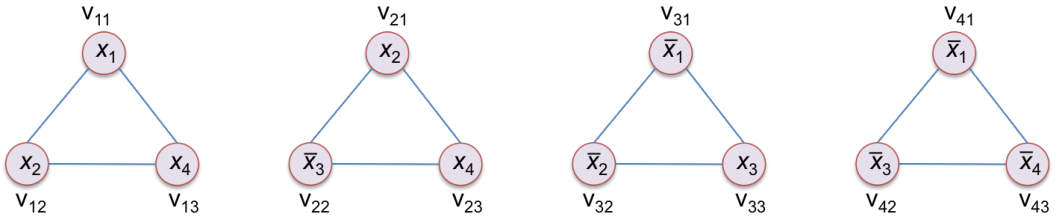

Figura 1.8: Grafo dei triangoli per la formula ϕ_3

Gli archi nei triangoli servono a selezionare un solo nodo per ogni triangolo nell'insieme indipendente: questo corrisponderà a rendere vero quel letterale (e questo ci basta per rendere vera la clausola alla quale il letterale appartiene) per ogni clausola. Tuttavia non possiamo rendere veri i letterali in modo arbitrario in quanto così facendo potremmo dover rendere vero sia un letterale che la sua negazione e ciò è, come abbiamo già detto, ovviamente impossibile. Pertanto dovremo aggiungere al grafo che stiamo costruendo degli archi per codificare i conflitti fra i letterali. Dovremo aggiungere un arco fra due nodi in triangoli diversi che rappresentano due letterali uno la negazione dell'altro. Come esempio, il grafo della Figura 1.8 diventerà il grafo della Figura 1.9, che chiameremo $G_{3\text{SAT}}$. La costruzione richiede tempo polinomiale in quanto il numero di nodi e di archi da specificare è proporzionale al numero di clausole della formula di partenza.

Il grafo così costruito ha la seguente proprietà: esiste un insieme indipendente di taglia k se e solo se la formula di partenza è soddisfacibile. Infatti se la formula è soddisfacibile allora esiste un assegnamento che rende vero almeno un letterale per ogni clausola. I nodi del grafo corrispondenti a questi letterali formano un insieme

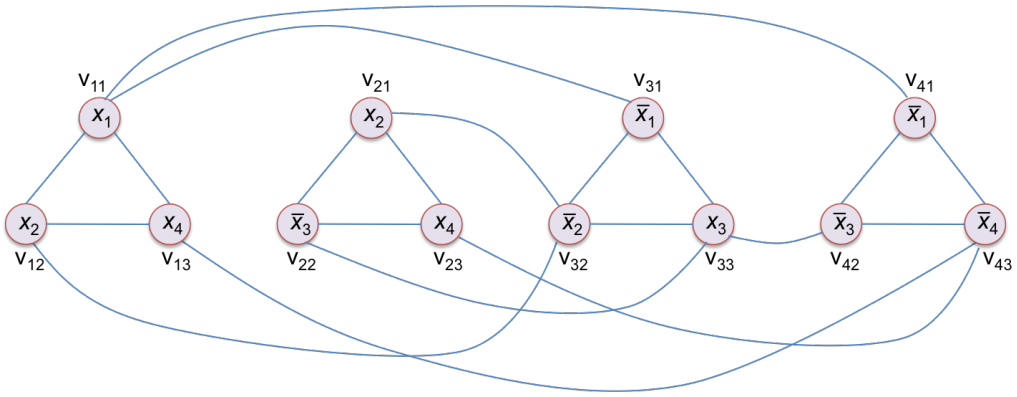

Figura 1.9: Grafo $G_{3\text{SAT}}$ per la formula ϕ_3

indipendente: infatti non può esistere un arco fra due di questi nodi in quanto ognuno di essi fa parte un gruppo di 3 diverso e ogni coppia non può essere in conflitto in quanto tutti hanno valore vero. Viceversa se esiste un insieme indipendente di k vertici (l'insieme indipendente non può avere più di k nodi in quanto ce ne sarebbero due appartenenti allo stesso triangolo e che quindi sarebbero collegati da un arco), è possibile soddisfare la formula assegnando vero ai letterali corrispondenti a quei k vertici: ognuno di essi è in una clausola diversa in quanto i 3 nodi di una clausola sono legati da un arco e nessuna coppia è in conflitto in quanto le coppie in conflitto sono anch'esse legate da un arco.

Riconsiderando l'esempio della Figura 1.9 si ha che l'insieme di nodi $\{v_{11}, v_{21}, v_{33}, v_{43}\}$ è un insieme indipendente di taglia 4. Tale insieme corrisponde all'assegnamento $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$ che come abbiamo visto in precedenza rende vera la formula ϕ_3. Un qualsiasi altro insieme indipendente di $G_{3\text{SAT}}$ corrisponde a un assegnamento che rende vera ϕ_3 e un qualsiasi assegnamento che rende vera ϕ_3 corrisponde a un insieme indipendente nel grafo $G_{3\text{SAT}}$.

A questo punto usiamo l'oracolo per risolvere il problema IndependentSet e vedere se esiste un insieme indipendente di taglia k: se esiste la formula è soddisfacibile, se non esiste la formula non è soddisfacibile. □

1.5.5 CircuitSat

Prima di tutto dobbiamo specificare cosa intendiamo per circuito. Consideriamo gli operatori booleani standard: $\wedge$ (AND), $\vee$ (OR), e $\neg$ (NOT). Un circuito è rappresentato da un grafo direzionato che descrive un circuito fisico fatto di porte AND, OR e NOT, come mostrato ad esempio nella Figura 1.10.

I nodi con un arco entrante etichettato con *input* rappresentano l'input al circuito; i nodi etichettati con una costante (0 oppure 1), rappresentano dei valori costanti usati dal circuito; infine i nodi etichettati con un operatore booleano, $\wedge$, $\vee$ e $\neg$, rappresentano le corrispondenti operazioni booleane. Gli archi entranti in un nodo che rappresenta

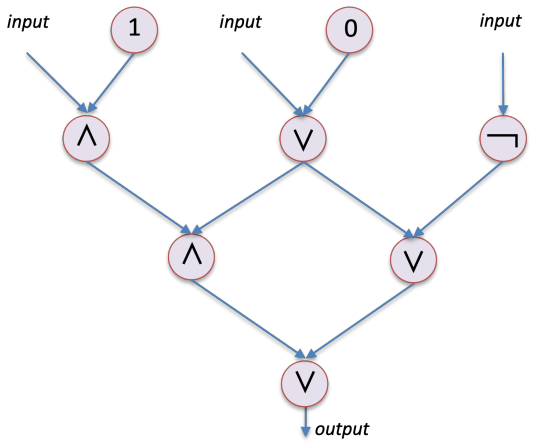

Figura 1.10: Un circuito

una operazione portano l'input per l'operazione, mentre l'arco uscente veicola l'output dell'operazione. L'ouput dell'operazione senza archi uscenti è l'output del circuito.

Un circuito calcola una funzione booleana dei suoi input: la funzione è quella descritta dal circuito stesso: ognuno degli operatori booleani prende l'input e calcola l'output fino ad arrivare al nodo senza archi uscenti la cui computazione sui valori passati sugli archi entranti (o sull'arco entrante) fornisce il valore di output del circuito.

Ad esempio, il circuito della Figura 1.10 sull'input $1, 0, 0$ assegnato ai 3 nodi di input, produce come output delle 3 porte booleane più in alto, rispettivamente, $1, 0, 1$, quindi come output delle successive 2 porte booleane 0 e 1, ed infine come output della porta $\vee$ più in basso, e quindi come output del circuito, il valore 1.

Avendo definito un circuito, possiamo ora definire il problema della soddisfacibilità di un circuito. Dato un circuito, il problema della soddisfacibilità del circuito consiste nello stabilire se esiste un assegnamento di input che causa un valore di output pari a 1. Se ciò è possibile diremo che il circuito è soddisfacibile. Nell'esempio precedente il circuito è soddisfacibile, infatti abbiamo visto che l'assegnamento $1, 0, 0$ all'input produce come output il valore 1.

Lemma 1.5.17 CircuitSat$\leq_P$3Sat.

Dimostrazione. Consideriamo una istanza arbitraria di CircuitSat. Vogliamo costruire una equivalente istanza di 3Sat, problema per il quale abbiamo a disposizione un oracolo. Per costruire tale formula utilizzeremo una variabile per ogni input (variabile o costante) e per ogni porta del circuito. Per far sì che la formula sia equivalente al circuito, tutte le variabili che sono di fatto vincolate perchè rappresentano il valore di una costante o di una porta del circuito saranno rappresentate da clausole che in qualche modo forzano il valore corretto.

Per differenziare le variabili "libere" da quelle vincolate (che rappresentano valori costanti o il valore di output di una porta del circuito) utilizzeremo per le prime la lettera x, quindi $x_1, x_2, \ldots$ saranno le variabili di input del circuito, mentre per le altre la lettera y, quindi $y_1, y_2, \ldots$ saranno le variabili che rappresentano le costanti o i valori

di uscita delle porte del circuito.

La formula ϕ che vogliamo costruire deve vincolare i valori delle variabili y in modo tale che esse rappresentino la computazione del circuito.

Consideriamo ad esempio una porta $\neg$ con input a (che può essere una x o una y) e output y (vedi Figura 1.11). Vogliamo inserire una clausola per codificare il fatto che y deve essere la negazione di a. Tale clausola deve valere 1 se e solo se $y = \overline{a}$. In altre parole cerchiamo una clausola C tale che la seguente tavola della verità sia soddisfatta:

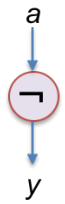

Figura 1.11: Porta NOT

a	y	C
0	0	0
0	1	1
1	0	1
1	1	0

In questo modo il valore di y sarà il negato di a se e solo se $C = 1$, quindi C codifica la porta $\neg$. La formula $C = (a \vee y) \wedge (\overline{a} \vee \overline{y})$ soddisfa la proprietà richiesta. Dunque in realtà stiamo inserendo due clausole, ognuna di due letterali, per codificare una porta $\neg$.

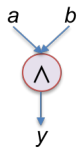

Figura 1.12: Porta AND

Analogamente possiamo fare per le porte $\wedge$ e $\vee$. Consideriamo una porta AND con input a e b (che possono essere sia delle x che delle y) e output y (vedi Figura 1.12). In questo caso la clausola C deve codificare il fatto che $y = a \wedge b$. Considerando tutti i possibili valori di a, b, e y, inseriremo un 1 nella tavola di verità di C per quelle combinazioni che soddisfano il vincolo:

a	b	y	C
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

La formula $C = (\overline{y} \vee a) \wedge (\overline{y} \vee b) \wedge (y \vee \overline{a} \vee \overline{b})$ soddisfa la tavola della verità richiesta. Quindi inseriamo 3 clausole, due con 2 letterali e una con 3 letterali.

Infine consideriamo una porta OR con input a e b e output y (vedi Figura 1.13). In questo caso la clausola C deve codificare il fatto che $y = a \vee b$. La tavola della verità che specifica C è la seguente:

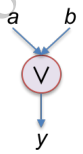

Figura 1.13: Porta OR

a	b	y	C
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

La formula $C = (y \vee \overline{a}) \wedge (y \vee \overline{b}) \wedge (\overline{y} \vee a \vee b)$ soddisfa la tavola della verità richiesta. Come per una porta AND anche per una porta OR inseriamo 3 clausole, due con 2 letterali e una con 3 letterali.

Dobbiamo inserire anche delle clausole che codificano gli input di valori costanti. Ma questo è facile da fare in quanto per il valore 1 inseriamo la corrispondente variabile y in forma vera, mentre per il valore 0 inseriamo la corrispondente variabile y in forma negata, cioè $\overline{y}$.

L'ultima clausola da inserire è quella che codifica l'output ed ovviamente sarà la variabile y di uscita della porta finale del circuito (in forma vera).

Codificando gli input e tutte le porte del circuito avremo creato una formula $\phi = C_1 \cdot C_2 \cdot \ldots$ equivalente al circuito, nel senso che il valore calcolato dal circuito per un determinato assegnamento delle variabili di input $x_1, x_2, \ldots$ è lo stesso valore che ha ϕ sullo stesso assegnamento delle variabili $x_1, x_2, \ldots$. La formula ϕ contiene un certo numero di clausole (il numero di clausole dipende da quante porte ci sono nel circuito e dalla tipologia delle porte), ognuna delle quali ha $1, 2$ o 3 letterali.

L'ultimo passo da fare è quello di mostrare che la formula costruita, in cui le clausole hanno 1, 2 o 3 letterali, può essere trasformata in un'altra equivalente in cui tutte le clausole hanno esattamente 3 letterali.

Per fare ciò possiamo inserire delle variabili fittizie z_1 e z_2 forzandole a valere 0. Con tali variabili a disposizione è facile aumentare il numero di letterali di una clausola semplicemente aggiungendo z_1 o z_1 e z_2. Per forzare il valore di z_1 e z_2 ad essere 0 possiamo inserire le clausole $\overline{z_1}$ e $\overline{z_1}$.

Questo però crea il problema che queste due clausole hanno un solo letterale e non 3. Possiamo risolvere questo problema usando ulteriori due variabili fittizie, z_3 e z_4 e inserendo le seguenti 4 clausole al posto di $\overline{z_1}$:

$$(\overline{z_1} \vee z_3 \vee z_4), (\overline{z_1} \vee z_3 \vee \overline{z_4}), (\overline{z_1} \vee \overline{z_3} \vee z_4), (\overline{z_1} \vee \overline{z_3} \vee \overline{z_4}),$$

e le seguenti 4 clausole al posto di $\overline{z_2}$:

$$(\overline{z_2} \vee z_3 \vee z_4), (\overline{z_2} \vee z_3 \vee \overline{z_4}), (\overline{z_2} \vee \overline{z_3} \vee z_4), (\overline{z_2} \vee \overline{z_3} \vee \overline{z_4}).$$

La presenza di queste 8 clausole impone che per soddisfare la formula dobbiamo necessariamente avere $z_1 = z_2 = 0$, indipendentemente dai valori di z_3 e z_4.

Questo ci permette di trasformare le clausole con 1 o 2 letterali in equivalenti clausole con esattamente 3 letterali: se una clausola ha un solo letterale t, la trasformiamo in

$t \vee z_1 \vee z_2$; se ne ha due allora la trasformiamo in $t_1 \vee t_2 \vee z_1$. □

Facciamo un esempio. Consideriamo il circuito della Figura 1.10 ed associamo ad ogni porta una variabile così come mostrato nella Figura 1.14.

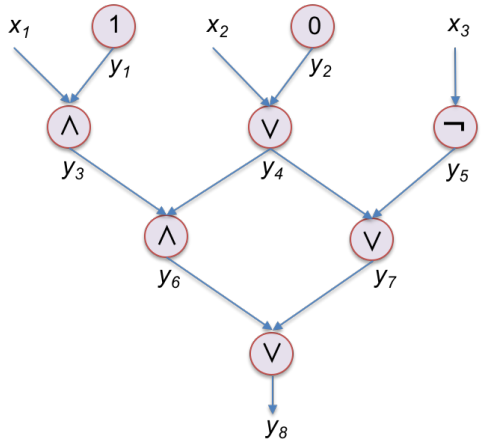

Figura 1.14: Il circuito della Figura 1.10 con le variabili (x_i e y_i) associate alle porte.

Abbiamo usato la lettera x per le variabili che rappresentano l'input (x_1, x_2 e x_3) e la lettera y per le variabili vincolate al valore di uscita di una porta (da y_1 a y_8). La variabile y_1 rappresenta il valore costante 1 pertanto inseriremo la clausola $C_1 = y_1$. Per la variabile y_2 che rappresenta il valore costante 0, invece inseriremo la clausola $C_2 = \overline{y_2}$. La variabile y_3 rappresenta il valore di uscita di una porta AND i cui input sono x_1 e y_1. Pertanto, per la variabile y_3, inseriremo le clausole $C_3 = \overline{y_3} + x_1$, $C_4 = \overline{y_3} + y_1$, $C_5 = y_3 + \overline{x_1} + \overline{y_1}$. La variabile y_4, invece, rappresenta il valore di uscita di una porta OR i cui input sono x_2 e y_2. Pertanto, per la variabile y_4, inseriremo le clausole $C_6 = y_4 + \overline{x_2}$, $C_7 = y_4 + \overline{y_2}$, $C_8 = \overline{y_4} + x_2 + y_2$. Per la variabile y_5 che rappresenta l'output di una porta NOT il cui input è x_3, introdurremo le clausole $C_9 = y_5 + \overline{x_3}$, $C_{10} = \overline{x_3} + y_5$. Procedendo in modo simile, per la variabile y_6 introdurremo le clausole $C_{11} = \overline{y_6} + x_3$, $C_{12} = \overline{y_6} + y_4$, $C_{13} = y_6 + \overline{x_3} + \overline{y_4}$, per la variabile y_7 le clausole $C_{14} = y_7 + \overline{y_4}$, $C_{15} = y_7 + \overline{y_5}$, $C_{16} = \overline{y_7} + y_4 + y_5$, e per la variabile y_8 le clausole $C_{17} = y_8 + \overline{y_6}$, $C_{18} = y_8 + \overline{y_7}$, $C_{19} = \overline{y_8} + y_6 + y_7$. Infine, introduciamo una clausola per il valore di output del circito: $C_{20} = y_8$.

Queste 20 clausole permettono di rappresentare il circuito; tuttavia non tutte le clausole hanno 3 letterali. Pertanto introdurremo le otto clausole aggiuntive con le variabili z_1, z_2, z_3 e z_4, e modificheremo le clausole con uno o due letterali aggiungendo z_1 e z_2, ottenendo la formula 3SAT finale che contiene 28 clausole (mostrate a margine).

$$
\begin{aligned}
C_1 &= y_1 + z_1 + z_2 \\
C_2 &= \overline{y_2} + z_1 + z_2 \\
C_3 &= \overline{y_3} + x_1 + z_1 \\
C_4 &= \overline{y_3} + y_1 + z_1 \\
C_5 &= y_3 + \overline{x_1} + \overline{y_1} \\
C_6 &= y_4 + \overline{x_2} + z_1 \\
C_7 &= y_4 + \overline{y_2} + z_1 \\
C_8 &= \overline{y_4} + x_2 + y_2 \\
C_9 &= y_5 + \overline{x_3} + z_1 \\
C_{10} &= y_5 + \overline{x_3} + z_1 \\
C_{11} &= \overline{y_6} + x_3 + z_1 \\
C_{12} &= \overline{y_6} + y_4 + z_1 \\
C_{13} &= y_6 + \overline{x_3} + \overline{y_4} \\
C_{14} &= y_7 + \overline{y_4} + z_1 \\
C_{15} &= y_7 + \overline{y_5} + z_1 \\
C_{16} &= \overline{y_7} + y_4 + y_5 \\
C_{17} &= y_8 + \overline{y_6} + z_1 \\
C_{18} &= y_8 + \overline{y_7} + z_1 \\
C_{19} &= \overline{y_8} + y_6 + y_7 \\
C_{20} &= y_8 + z_1 + z_2 \\
C_{21} &= \overline{z_1} + z_3 + z_4 \\
C_{22} &= \overline{z_1} + z_3 + \overline{z_4} \\
C_{23} &= \overline{z_1} + \overline{z_3} + z_4 \\
C_{24} &= \overline{z_1} + \overline{z_3} + \overline{z_4} \\
C_{25} &= \overline{z_2} + z_3 + z_4 \\
C_{26} &= \overline{z_2} + z_3 + \overline{z_4} \\
C_{27} &= \overline{z_2} + \overline{z_3} + z_4 \\
C_{28} &= \overline{z_2} + \overline{z_3} + \overline{z_4}
\end{aligned}
$$

1.5.6 HAMCYCLE

Questo problema è definito per un grafo direzionato e non pesato. Dato un grafo $G = (V, E)$ un ciclo C in G è un ciclo *hamiltoniano* se visita ogni vertice esattamente una volta.

Problema 1.5.18 HAMCYCLE*: Dato un grafo direzionato G, G contiene un ciclo hamiltoniano?*

Lemma 1.5.19 3SAT$\leq_P$ HAMCYCLE.

DIMOSTRAZIONE. Consideriamo una qualsiasi istanza di 3SAT. Siano $x_1, \ldots, x_n$ le variabili e $C_1, \ldots, C_k$ le clausole. Il nostro obiettivo è quello di sfruttare un oracolo che risolve HAMCYCLE per risolvere l'istanza di 3SAT. Per fare questo dovremo codificare in qualche modo l'istanza di 3SAT usando un grafo direzionato non pesato.

Poichè sono coinvolte n variabili, ci sono 2^n possibili diverse combinazioni dei valori da assegnare come input alle variabili. L'idea è quella di costruire un grafo che contiene 2^n diversi cicli hamiltoniani, ognuno dei quali corrisponde a un possibile assegnamento dei valori di input delle variabili nell'istanza di 3SAT. Dopo aver costruito questo grafo inseriremo dei nodi che modellano i vincoli imposti dalle clausole della formula.

Costruiamo n cammini $P_1, \ldots, P_n$, dove P_i è formato dai nodi $v_{i,1}, v_{i,2}, \ldots, v_{i,b}$ per un valore di $b = 3k + 3$. Inseriamo sia gli archi $(v_{i,j}, v_{i,j+1})$ che gli archi nell'altra direzione $(v_{i,j+1}, v_{i,j})$. Quindi ogni cammino P_i può essere percorso sia da "sinistra a destra" da $v_{i,1}$ a $v_{i,b}$ che da "destra a sinistra" da $v_{i,b}$ a $v_{i,1}$.

I cammini sono inoltre uniti dai seguenti archi. Per ogni $i = 1, 2, \ldots, n-1$, inseriamo un arco da $v_{i,1}$ a $v_{i+1,1}$ e un altro arco sempre da $v_{i,1}$ ma questa volta a $v_{i+1,b}$. Cioè ci sono due archi che dal primo nodo del cammino P_i portano rispettivamente al primo e all'ultimo nodo del cammino P_{i+1}. In modo simmetrico inseriamo due archi dall'ultimo nodo del cammino P_i, al primo e all'ultimo nodo del cammino P_{i+1}, cioè gli archi da $(v_{i,b}, v_{i+1,1})$ e $(v_{i,b}, v_{i+1,b})$. Inoltre inseriamo altri due nodi, s e t e gli archi $(s, v_{1,1}), (s, v_{1,r}), (v_{n,1}, t), (v_{n,b}, t)$ ed infine l'arco (t, s). Il grafo risultante è mostrato nella Figura 1.15.

Prima di procedere con la prova, cerchiamo di capire quali sono i cicli hamiltoniani nel grafo che abbiamo costruito. Poichè c'è un solo arco che esce da t, sappiamo che un qualsiasi ciclo hamiltoniano dovrà usare tale arco, quindi in qualunque ciclo hamiltoniano dobbiamo passare necessariamente da t ad s. Quando siamo in s possiamo procedere o passando nel primo o nell'ultimo nodo di P_1. Se passiamo nel primo attraverseremo P_1 da sinistra a destra, se passiamo nell'ultimo attraverseremo P_1 da destra a sinistra. Si noti che sebbene dal primo o dall'ultimo nodo di P_1 si abbia anche la possibilità di passare ai nodi di P_2 senza attraversare P_1, se lo facessimo non avremmo modo di visitare gli altri nodi di P_1 in quanto per ritornarci dovremmo necessariamente ripassare da t a s. Stesso discorso per gli altri cammini, quindi per visitare tutti i nodi dobbiamo visitare i cammini P_i, in ordine, cioè prima P_1 poi P_2 e così via fino a P_n. Inoltre ogni cammino potrà essere visitato o da sinistra a destra o da destra a sinistra passando per tutti i nodi in quanto non c'è modo di "saltare" visto che ogni nodo diverso dal primo e dall'ultimo in un cammino è collegato solo al nodo precedente e al successivo.

Pertanto le possibilità di creare cicli diversi fra di loro sono esattamente n, una per ogni P_i, ed in ogni caso abbiamo 2 scelte, attraversare P_i da destra a sinistra oppure farlo da sinistra a destra. Pertanto ci sono esattamente 2^n diversi cicli hamiltoniani nel grafo che abbiamo costruito. Questo ci permette di associare ognuno di questi cicli a una delle 2^n diverse possibili combinazioni per l'input di 3SAT: se P_i viene attraversato da sinistra a destra allora $x_i = 1$, se invece attraversiamo P_i da destra a sinistra allora

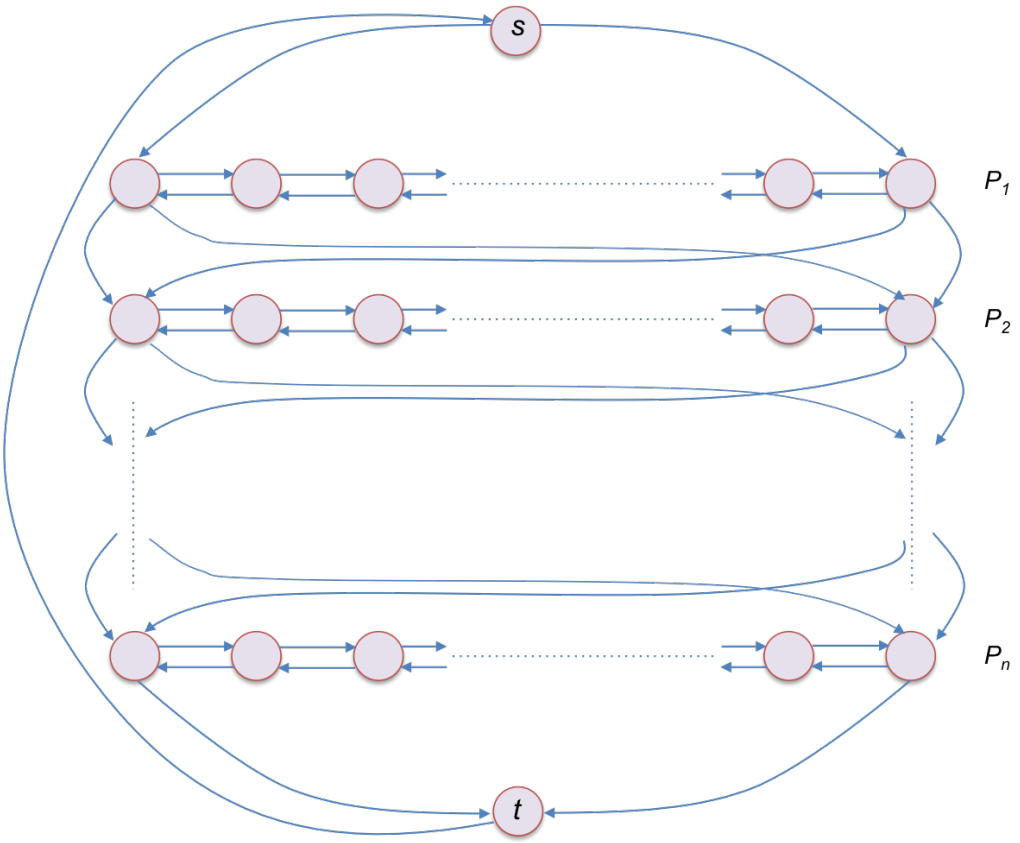

Figura 1.15: Riduzione di 3SAT a HAMCYCLE

$x_i = 0$.

A questo punto inseriamo dei nodi per modellare la clausole del problema 3SAT. Ogni clausola rappresenta un "vincolo" che di fatto esclude alcune possibili scelte per l'input in quanto queste scelte renderebbero la formula falsa. L'effetto sarà quello di "eliminare" i corrispondenti cicli hamiltoniani dal nostro grafo. Se alla fine un ciclo sopravvive, quel ciclo corrisponde a un assegnamento che rende vera la formula.

Consideriamo ad esempio la clausola

$$C_j = \overline{x_1} + x_3 + x_4.$$

Con la nostra associazione ai cicli hamiltoniani, questa clausola richiede di attraversare P_1 da destra a sinistra ($x_1 = 0$ rende vera la clausola), oppure di attraversare P_3 da sinistra a destra ($x_2 = 1$ rende vera la clausola), oppure di attraversare P_3 da sinistra a destra ($x_3 = 1$ rende vera la clausola). Per "forzare" la direzione voluta inseriamo un nodo c_j extra nei cammini P_1, P_3 e P_4. Più precisamente, inseriamo k nodi extra, $c_1, c_2, \ldots, c_k$ uno per ogni clausola C_j, $j = 1, 2, \ldots, k$, e, per evitare conflitti, useremo i nodi $3j$ e $3j + 1$ per i letterali della clausola j. Quindi se il letterale è x_i, per inserire

c_j nel cammino P_i useremo degli archi fra i nodi $(v_{i,3j}, c_j)$ e $(c_j, v_{i,3j+1})$ per forzare l'attraversamento di P_i da sinistra a destra, mentre se il letterale è $\overline{x_i}$ inseriamo gli archi $(v_{i,3j+1}, c_j)$ e $(c_j, v_{i,3j})$ per forzare l'attraversamento di P_i da destra a sinistra. Si noti che questo schema lascia sempre un nodo "di transito" per ogni coppia di nodi $3j$ e $3j+1$, i nodi $3j+2$, sia a sinistra che a destra. Questo nodo è fondamentale per la correttezza della costruzione, come vedremo fra poco.

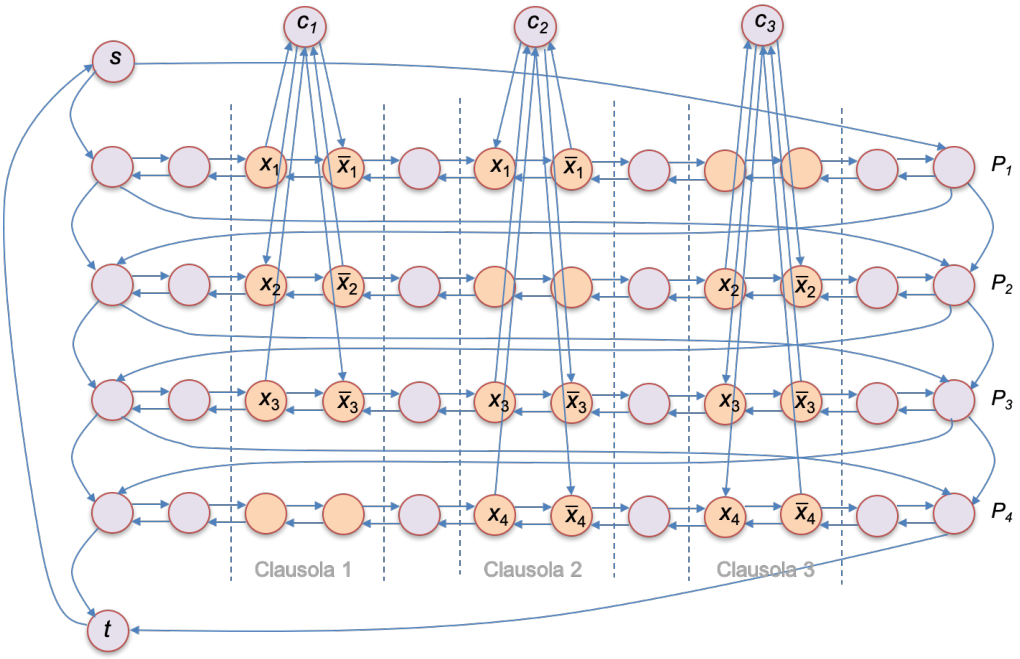

Figura 1.16: Il grafo finale per la formula ϕ_4

Come esempio, la Figura 1.16 mostra il grafo finale per la formula

$$\phi_4 = (x_1 + \overline{x_2} + x_3) \cdot (\overline{x_1} + x_3 + x_4) \cdot (x_2 + \overline{x_3} + \overline{x_4}).$$

Abbiamo completato la costruzione del grafo e non ci resta che provare che effettivamente l'istanza iniziale di 3Sat è soddisfacibile se e solo se il grafo che abbiamo costruito ha un ciclo hamiltoniano.

Quindi, assumiamo che esista un assegnamento che rende vera l'istanza di 3Sat. Consideriamo il seguente percorso nel nostro grafo: se x_i ha ricevuto il valore 1 attraversiamo P_i da sinistra a destra, altrimenti lo attraversiamo da destra a sinistra. Per ogni clausola C_j riusciremo ad attraversare il nodo c_j facendo le necessarie "deviazioni" dal cammino P_i (tramite i nodi $v_{i,3j}$ e $v_{i,3j+1}$) e seguendo la direzione scelta. Infatti le deviazioni sono state costruite proprio per attraversare i nodi c_j nella direzione indicata dal valore delle variabili. La Figura 1.17 mostra un possibile percorso per l'assegnamento $x_1 = 1, x_2 = 1, x_3 = 1$ e $x_4 = 0$ che rende la formula ϕ_4 vera. Si noti che i nodi c_j possono essere visitati dalla coppia di nodi che corrispondono a un qualsiasi letterale che rende vera la clausola. Nell'esempio della Figura 1.17, il nodo c_1, che è

stato visitato da x_3 "deviando" il cammino P_3, poteva essere visitato anche sfruttando il nodo x_1 nel cammino P_1. Ovviamente basta (e si deve) visitarlo una volta sola.

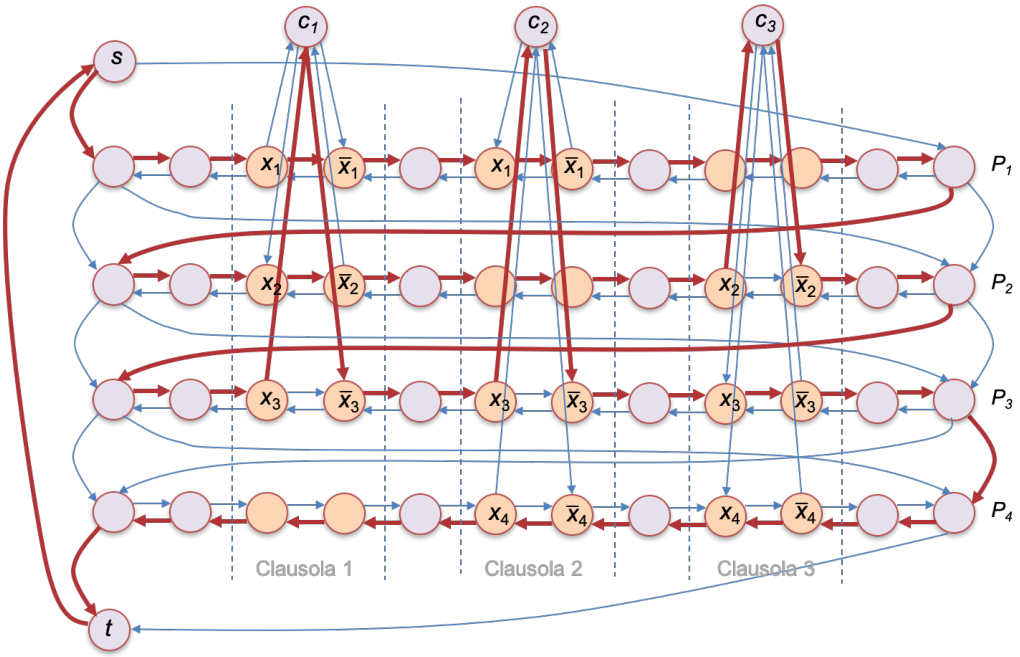

Figura 1.17: Un ciclo hamiltoniano per l'assegnamento $x_1 = 1, x_2 = 1, x_3 = 1$ e $x_4 = 0$.

Viceversa, assumiamo che esista un ciclo hamiltoniano C nel grafo che abbiamo costruito. Osserviamo che se il ciclo C passa per c_j arrivando da $v_{i,3j}$ deve necessariamente lasciare c_j andando a $v_{i,3j+1}$, altrimenti non potrebbe più visitare il nodo di transito $v_{i,3j+2}$. In modo simmetrico, se si visita c_j provenendo da $v_{i,3j+1}$ dovrà lasciarlo andando a $v_{i,3j}$, altrimenti il nodo di transito $v_{i,3j-1}$ non potrà più essere visitato. In altre parole l'unico modo per visitare i nodi c_j è quello di seguire le deviazioni costruite sui cammini (e per questo il nodo di transito svolge un ruolo importante). Duqnue il nodo di transito svolge un ruolo fondamentale: quando si lascia un cammino per visitare un nodo c_j bisogna necessariamente ritornare su quel cammino altrimenti non si avrebbe più la possibilità di vistare il successivo nodo di transito.

Pertanto C attraversa tutti i cammini da destra a sinistra o viceversa, seguendo le devizioni per visitare anche i nodi c_j. Ignorando le deviazioni, quindi, possiamo assegnare dei valori alle variabili x_i in funzione di come il ciclo attraversa i cammini P_i: se li attraversa da sinistra a destra assegniamo il valore 1 altrimenti il valore 0. Poichè il ciclo attraversa ovviamente anche i nodi c_j, che, ricordiamo, rappresentano il fatto che la clausola è vera, un tale assegnamento rende vere tutte le clausole e quindi anche l'istanza del problema 3Sat. □

Si può considerare il problema anche in un grafo non direzionato: di fatto è un caso particolare visto che un grafo non direzionato può essere rappresentato facilmente da un grafo direzionato semplicemente sostituendo ogni arco non direzionato fra due nodi u

e v con gli archi (u, v) e (v, u). Quindi chiaramente UndHamCycle$\leq_P$HamCycle. Tuttavia UndHamCycle non è più facile di HamCycle. Si può dimostrare, ad esempio, che VertexCover$\leq_P$UndHamCycle e siccome, come vedremo in seguito, VertexCover$\leq_P$3Sat, per la proprietà transitiva si ha che HamCycle$\leq_P$UndHamCycle.

1.5.7 TSP

Un commesso viaggiatore deve visitare n città, $c_1, \ldots, c_n$. Il commesso risiede nella città c_1. Il problema consiste nel trovare una sequenza delle città che permetta al commesso viaggiatore di visitare ogni città esattamente una volta, di ritornare a casa, il tutto viaggiando il meno possibile. Questo problema è simile a quello dei cicli hamiltoniani; in questo caso però ad ogni arco è associato un peso (distanza).

Il problema può essere modellato con un grafo $G = (V, E)$, in cui $V = \{c_1, \ldots, c_n\}$ e $E = \{(c_i, c_j)\}$. Ad ogni arco (c_i, c_j) associeremo la distanza $d(c_i, c_j)$ che serve per raggiungere c_j partendo da c_i. Tale distanza potrebbe anche non rappresentare la distanza fisica. Ovviamente fornisce la metrica per misurare ciò che il commesso viaggiatore vuole minimizzare durante il "giro". Pertanto non è nemmeno necessario che sia simmetrica. Possiamo pensare alla "distanza" come a un costo in cui il commesso viaggiatore incorre andando da una città all'altra. L'obiettivo è quello di trovare una sequenza $c_{i_1}, \ldots, c_{i_n}$ con $i_1 = 1$ (cioè si parte dalla città di residenza del commesso viaggiatore) tale che $\sum_j d(c_{i_j}, c_{i_{j+1}}) + d(i_n, i_1)$ sia minima.

Il problema ha molte applicazioni pratiche. La prima, suggerita anche dalla formulazione stessa, è quella della pianificazione di consegne da parte di un corriere che partendo dal punto di smistamento della merce deve effettuare varie tappe per poi ritornare in sede. Anche altri problemi simili nel settore dei trasporti (taxi, camion, flotte navali) possono essere formnulati con TSP per minimizzare il tempo di viaggio e/o i costi del carburante. Oltre a questi casi che sembrano abbastanza naturali, TSP trova applicazioni anche in problemi apparentemente molto diversi, come la produzione industriale di utensili per la quale una macchina deve lavorare su parti in diverse posizioni e si deve quindi spostare da una all'altra. Più in generale nelle applicazioni che coinvolgono macchine robot è spesso necessario pianificare le varie attività (città) riducendo i tempi/costi di spostamento (metrica). Problemi simili sorgono nei più svariati campi, dalla biologia computazionale alle telcomunicazioni, dalla meteoreologia alla medicina.

La versione decisionale del problema è la seguente:

Problema 1.5.20 TSP*: Dato un insieme di distanze fra n città, e un limite D sulla distanza, esiste un giro di lunghezza al massimo D?*

Lemma 1.5.21 HamCycle$\leq_P$ TSP.

Dimostrazione. Abbiamo a disposizione un oracolo per TSP. Sia $G = (V, E)$ un grafo direzionato con n nodi, per il quale vogliamo sapere se esiste un ciclo hamiltoniano. Definiamo la seguente istanza del problema TSP. Abbiamo una città c_i per ogni nodo v_i del grafo G. Inoltre definiamo le distanze in questo modo: $d(c_i, c_j)$ vale 1 se esiste l'arco (v_i, v_j), 2 altrimenti.

Il grafo G ha un ciclo hamiltoniano se e solo se esiste un giro di lunghezza al massimo n per il commesso viaggiatore. Infatti assumiamo che esista un ciclo hamiltoniano in G, questo significa che esiste una sequenza di n nodi $v_1, \ldots, v_n$ per i quali esistono gli archi $(v_1, v_2), (v_2, v_3,) \ldots, (v_{n-1}, v_n)$ ed infine (v_n, v_1). Per come è stato definito l'istanza di TSP, la corrispondente sequenza di città $c_1, c_2, \ldots, c_n$ è un giro di lunghezza n che permette al commesso viaggiatore di partire da c_1 e ritornarci dopo aver visitato una e una sola volta tutte le altre città. Viceversa, assumiamo che esista un giro di lunghezza n per il commesso viaggiatore e che tale giro sia composto dalle città $c_1, c_2, \ldots, c_n$. Per come è stato definita l'istanza di TSP, questo significa che nel grafo G esistono gli archi $(v_1, v_2), (v_2, v_3,) \ldots, (v_{n-1}, v_n)$ ed infine (v_n, v_1). Infatti la distanza fra tutte le coppie di questo giro deve necessariamente essere 1 in quanto la lunghezza totale è n e se ce ne fosse anche solo una coppia a distanza 2 la lunghezza sarebbe maggiore. Pertanto $v_1, \ldots, v_n$ è un ciclo hamiltoniano per G. □

Notiamo che nella riduzione abbiamo sfruttato il fatto che le "distanze" fra le città possono essere asimmetriche: questo ci ha permesso di creare un'istanza di TSP che, grazie all'asimmetria delle distanze, ha codificato la presenza o l'assenza di archi nel grafo del problema originario.

Si può considerare il problema del commesso viaggiatore nel caso in cui le distanze siano simmetriche. Questa restrizione sulle distanze non rende il problema più facile: anche in questa versione il problema rimane difficile, e lo si può vedere riducendo UndHamCycle ad esso (la riduzione in questo caso non è semplice).

1.5.8 GraphColoring

Il problema della colorazione di grafi prende spunto dal problema della colorazione di una mappa: in una mappa si assegnano dei colori ai vari stati in modo tale che stati adiacenti, cioè che condividono un confine, abbiano un colore diverso. Storicamente il problema nasce con la seguente domanda: è possibile colorare una qualsiasi mappa con al massimo 4 colori?

Generalizzando il problema a un grafo non direzionato G, vogliamo assegnare un colore ad ogni nodo del grafo in modo tale che se esiste l'arco (u, v) allora i colori di u e di v sono diversi. Una mappa è un caso particolare in cui c'è un nodo per ogni stato e gli archi rappresentano le adiacenze. Si noti che una mappa è un caso ristretto in quanto le adiacenze devono rispettare dei vincoli fisici, mentre in generale gli archi possono collegare una coppia di nodi qualsiasi.

Formalmente definiamo una k-colorazione del grafo come una funzione $f : V \to \{1, 2, \ldots, k\}$ tale che per ogni arco (u, v) si ha $f(u) \neq f(v)$. In pratica stiamo usando gli interi $1, 2, \ldots, k$, come nomi per i k colori che si possono usare per colorare il grafo. Se esiste una k-colorazione per G allora diremo che G è k-colorabile.

È abbastanza evidente che l'esistenza di una k-colorazione dipende dal grafo. Un grafo senza archi potrebbe essere colorato con un solo colore. All'altro estremo, un grafo completo ha bisogno di un colore diverso per ogni nodo. La questione che ci poniamo è la seguente.

Problema 1.5.22 GRAPHCOLORING: *Dato un grafo G e un intero k, il grafo G è k-colorabile?*

Per specificare il parametro k, useremo anche la notazione k-GRAPHCOLORING.

Sebbene il problema nasca storicamente per la colorazione delle mappe, GRAPHCOLORING ha molteplici applicazioni come, ad esempio, nell'allocazione di risorse in presenza di particolari vincoli, nella progettazione di compilatori, nelle comunicazioni wireless. Si veda il paragrafo 8.7 di [20] per una descrizione di tali applicazioni.

Il caso $k = 2$ è un caso particolare la cui soluzione viene lasciata come esercizio (vedi Esercizio 15). Per $k = 2$ si può dare un algoritmo efficiente che decide se il grafo è 2-colorabile. Consideriamo dunque $k \geq 3$. Osserviamo che, abbastanza intuitivamente, il problema per $k > 3$ è più difficile del caso $k = 3$. In altre parole possiamo provare che:

Lemma 1.5.23 3-GRAPHCOLORING$\leq_P k$-GRAPHCOLORING, $k \geq 4$.

DIMOSTRAZIONE. Consideriamo una istanza di 3-GRAPHCOLORING e trasformiamola in una istanza di k-GRAPHCOLORING, $k \geq 4$, semplicemente inserendo $k - 3$ nodi fittizi, ognuno collegato a tutti gli altri nodi. Questi 3 nodi fittizi richiedono $k - 3$ colori diversi fra di loro e diversi dai colori di tutti gli altri nodi. In pratica per i nodi rimanenti dobbiamo risolvere il problema con i restanti 3 colori. Quindi il grafo ottenuto è k-colorabile se e solo se il grafo di partenza è 3-colorabile. Pertanto sapendo risolvere GRAPHCOLORING possiamo risolvere anche 3-GRAPHCOLORING. □

Consideriamo adesso il caso $k = 3$. Nel passaggio da $k = 2$ a $k = 3$, abbastanza sorprendentemente, il problema diventa molto più difficile. Addirittura si ha che:

Lemma 1.5.24 3SAT$\leq_P$3-GRAPHCOLORING.

DIMOSTRAZIONE. Partiamo da un'istanza di 3SAT, quindi una formula booleana fatta dalla congiunzione (AND) di k clausole $C_1, \ldots, C_k$, ognuna con 3 letterali delle variabili $x_1, \ldots, x_n$. Vogliamo risolvere il problema sfruttando un oracolo per 3-GRAPHCOLORING.

Useremo un nodo per ogni variabile x_i in forma vera e un nodo per ogni varibiale in forma negata $\overline{x_i}$ 2 nodi speciali, T, F che rappresentano rispettivamente i valori *true* e *false*, ed infine un nodo speciale B, che chiameremo nodo di base.

L'idea è quella di codificare l'impossibilità di rendere entrambi veri due letterali rendendo impossibile l'assegnamento dello stesso colore ai nodi che rappresentano i due letterali. Il nodo di base servirà per legare insieme i vari vincoli.

Pertanto iniziamo con il creare un arco fra ogni coppia di nodi x_i e $\overline{x_i}$, e uniamo entrambi con un arco al nodo di base. Facciamo lo stesso per i nodi T e F. Quindi ogni tripla $(x_i, \overline{x_i}, B)$ forma un triangolo, come pure la tripla (T, F, B). La Figura 1.18 mostra il grafo per 3 variabili.

Notiamo che per la presenza di questi archi è necessario che in ognuna di queste triple i nodi ricevano colori diversi. Pertanto data una 3-colorazione del grafo che stiamo costruendo potremo far riferimento al "colore di T", al "colore di F" e al "colore di B". Inoltre poichè B è connesso a tutti i nodi x_i e $\overline{x_i}$, ogni nodo x_i o $\overline{x_i}$ riceverà o il colore di T o il colore di F.

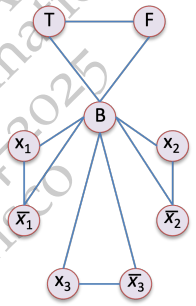

Figura 1.18: Costruzione del grafo

Questo grafo di base fornisce una corrispondenza 1 a 1 fra gli assegnamenti di valori booleani alle variabili $x_1, \ldots, x_n$ ed i colori dei nodi che rappresentano le variabili. Ora dobbiamo estendere questo grafo di base per codificare i vincoli dovuti alle clausole in modo tale che solo gli assegnamenti che rendono vera la formula corrispondano a delle 3-colorazioni del grafo.

Quindi per ogni clausola C_i introdurremo dei nodi fittizi e dei nuovi archi per escludere le colorazioni che corrispondono ad assegnamenti che rendono falsa la formula. Procediamo con un esempio e consideriamo la clausola $C_i = x_1 \vee \overline{x_2} \vee x_3$. Con la trasformazione usata per la costruzione del grafo G tale clausola si traduce in: almeno uno dei nodi $x_1, \overline{x_2}$ e x_3 deve ricevere il colore di T. Per "forzare" questa proprietà dobbiamo trovare un sottografo che inserito in G faccia in modo da eliminare le 3-colorazioni che non soddisfano questa proprietà. Trovare tale sottografo richiede un po' di sforzo: uno che funziona è mostrato nella Figura 1.19

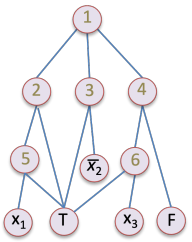

Figura 1.19: Costruzione del grafo, secondo widget

I 6 nuovi nodi si uniscono al grafo di base usando 5 nodi fra quelli già esistenti: i nodi T, F ed i nodi della clausola, cioè $x_1, \overline{x_2}$ e x_3. Per provare che il sottografo impone la proprietà richiesta, assumiamo per assurdo che tutti e tre i nodi della clausola ricevano il colore di F. Poichè x_1 e x_3 hanno il colore di F i nodi etichettati con 5 e 6, che sono collegati anche a T, dovranno necessariamente ricevere il colore di B. Consideriamo ora i 3 nodi etichettati 2, 3 e 4. Il nodo 2 è collegato a un nodo che riceve il colore di B ed al nodo T, quindi dovrà essere colorato con il colore di F. Il nodo 3 è collegato a T ed a $\overline{v_2}$ e quindi deve ricevere il colore di B. Infine il nodo 4 è collegato a un nodo che riceve il colore di B ed al nodo F quindi deve ricevere il colore di T. Riassumendo, i nodi 2, 3 e 4 devono ricevere, ripsettivamente, i colori di F, di B e di T. Tutti questi nodi sono collegati al nodo 1 e per questo nodo non ci sono più colori disponibili. Quindi, nel grafo appena costruito, non è possibile che $x_1, \overline{x_2}$ e x_3 ricevano tutti il colore di F.

Si noti la tecnica utilizzata per la costruzione del sottografo: si parte dal nodo 1 e lo si collega ai nodi 2, 3, 4 con l'intento di "forzare" su questi 3 nodi tutti e tre i colori in modo da rendere impossibile la colorazione. Poi si collegano i 3 nodi o direttamente ai letterali oppure ad altri nodi fittizi creati per forzare i colori necessari a creare la contraddizione. Inserendo opportunamente nodi ed archi è possibile forzare qualsiasi colorazione.

Non ci rimane da osservare che basta che uno solo fra $x_1, \overline{x_2}$ e x_3 riceva il colore di T per rendere possibile la 3-colorazione del grafo (questa parte della dimostrazione è lasciata come esercizio: partire da tutte le possibile colorazioni in cui almeno uno dei letterali ha il colore di T e trovare una colorazione del sottografo).

Per completare la costruzione del grafo finale, procediamo all'inserimento di un sottografo di 6 nodi simile a quello mostrato nella Figura 1.19 per ogni clausola; si noti che non c'è nessuna relazione fra il fatto che il letterale sia negato ed il fatto che questo è l'unico letterale che non richiede un nodo fittizio al terzo livello. Al posto di $x_1, \overline{x_2}$ e x_3 possiamo mettere qualunque combinazione, come ad esempio x_1, x_2, x_3; la struttura del sottografo da inserire è sempre lo stessa, cambiano solo i letterali in funzione della clausola.

Chiamiamo G' il grafo finale. La figura 1.20 mostra tale grafo per la formula $(x_1 + \overline{x_2} + x_3) + (\overline{x_1} + \overline{x_2} + x_3) + (x_1 + \overline{x_2} + \overline{x_3})$.

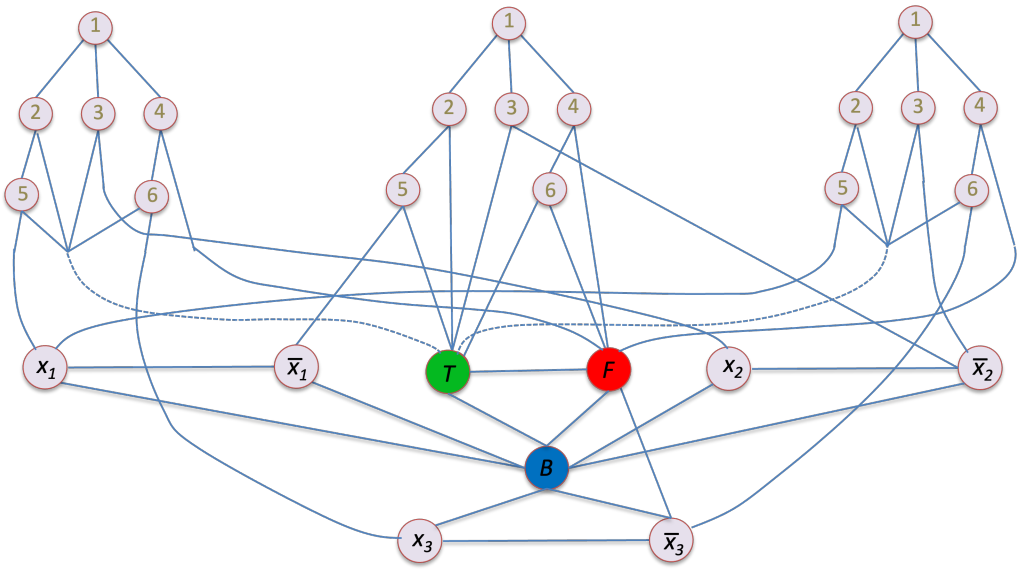

Figura 1.20: Grafo G' per la formula $(x_1 + \overline{x_2} + x_3) + (\overline{x_1} + \overline{x_2} + x_3) + (x_1 + \overline{x_2} + \overline{x_3})$

L'istanza iniziale di 3SAT è soddisfacibile se e solo se G' ammette una 3-colorazione. Per provarlo, assumiamo prima che esista una assegnazione di valori alle variabili $x_1, \ldots, x_n$ che renda vera la formula. Possiamo costruire una 3-colorazione di G semplicemente colorando prima i 3 nodi speciali T, F e B e poi assegnando a v_i il colore di T se $x_i = 1$ ed il colore di F se $x_1 = 0$. Per $\overline{v_i}$ il colore a quel punto sarà l'unico rimasto (quello di F se $x_i = 0$ e quello di T se $x_1 = 1$). Infine la colorazione si estende ai sottografi aggiunti come descritto nella costruzione del grafo: sappiamo che è possibile in quanto per ogni clausola almeno uno dei letterali ha ricevuto il valore vero.

Viceversa, assumiamo che G' ammetta una 3-colorazione. In tale colorazione ogni nodo v_i riceve o il colore di T o il colore di F; nel primo caso poniamo $x_i = 1$ e nel secondo caso poniamo $x_i = 0$. Con questo assegnamento, in ognuna della clausole ci sarà almeno un letterale vero. Infatti se così non fosse, ci sarebbe una clausola in cui tutti i letterali hanno il valore F. Questo tuttavia renderebbe impossibile la 3-colorazione del grafo nel sottografo costruito per quella clausola. Quindi possiamo concludere che ogni clausola è vera e quindi la formula è vera. □

Concludiamo la discussione sulla colorazione dei grafi con una nota: una mappa geografica è sempre colorabile con 4 colori! La prova di questo risultato è stata ottenuta solo nel 1976 da Appel e Haken e fa ricorso a una dimostrazione che utilizza una semplice induzione sul numero di stati e la prova del passo induttivo richiede l'analisi di circa 2000 casi. La prova è stata finalizzata solo grazie all'aiuto di un computer. Il problema di trovare una prova breve, ragionevolmente controllabile senza l'aiuto di un computer, è ancora aperto.

1.6 Problemi NP-completi

Tutti i problemi che abbiamo visto in precedenza appartengono a $\mathcal{NP}$ (per ognuno di essi è facile fornire un verificatore di potenziali soluzioni). Per nessuno di essi è stato trovato un algoritmo efficiente. Trovare un algoritmo efficiente per un singolo problema permetterebbe di classificare quel problema come "facile", ma lascerebbe aperta la questione $\mathcal{P}$-$\mathcal{NP}$. Per poter risolvere la questione $\mathcal{P}$-$\mathcal{NP}$ ci si è posto la seguente domanda: quali sono i problemi di $\mathcal{NP}$ più difficili? La strumento della riduzione che abbiamo introdotto in precedenza è ideale per individuare i problemi più difficili di $\mathcal{NP}$, che sono quei problemi $X \in \mathcal{NP}$ tali che un qualsiasi altro problema $Y \in \mathcal{NP}$ può essere ridotto a X:

$$\mathcal{NP}\text{-completi} = \{X \in \mathcal{NP} | \forall Y \in \mathcal{NP}, Y \leq_P X\}.$$

Tali problemi vengono detti $\mathcal{NP}$-completi. La motivazione per l'individuazione di problemi $\mathcal{NP}$-completi è abbastanza ovvia:

Lemma 1.6.1 *Sia A un problema $\mathcal{NP}$-completo. Allora A è risolvibile in modo efficiente se e solo se $\mathcal{P} = \mathcal{NP}$.*

DIMOSTRAZIONE. Se A può essere risolto in modo efficiente allora possiamo usarlo per risolvere in modo efficiente un qualunque altro problema in $\mathcal{NP}$. Infatti sia X un qualunque altro problema in $\mathcal{NP}$, poichè A è $\mathcal{NP}$-completo, si ha che $X \leq_P A$ e quindi anche X può essere risolto in tempo polinomiale. Per cui $\mathcal{P} = \mathcal{NP}$.

Viceversa, se $\mathcal{P} = \mathcal{NP}$ allora tutti i problemi in $\mathcal{NP}$ possono essere risolti in tempo polinomiale e quindi anche A può essere risolto in tempo polinomiale. □

Avendo definito la classe $\mathcal{NP}$-completi il prossimo passo è quello di individuare dei problemi appartenenti a tale classe. Sebbene la definizione sia chiara e semplice, non è altrettanto chiaro e immediato il fatto che effettivamente dei problemi $\mathcal{NP}$-completi esistano. Un problema $\mathcal{NP}$-completo deve avere una proprietà estremamente forte: deve poter essere usato per risolvere *qualunque* altro problema in $\mathcal{NP}$. Quindi per dimostrare che un problema X è $\mathcal{NP}$-completo dobbiamo trovare un modo per poter "trasformare" un qualsiasi altro problema in un'istanza di X. In altre parole dobbiamo creare una riduzione come quelle viste precedentemente ma con una piccola grande differenza: mentre negli esempi di riduzione visti finora dovevamo ridurre uno specifico problema ad X, adesso dobbiamo ridurre ad X un *qualsiasi* altro problema in $\mathcal{NP}$, cioè dobbiamo ridurre ad X tutti gli altri problemi in $\mathcal{NP}$.

Teorema 1.6.2 CIRCUITSAT *è $\mathcal{NP}$-completo.*

La prova di questo risultato va al di là degli obiettivi di questo corso. Tuttavia l'idea di base della dimostrazione è quella che un circuito può di fatto "implementare" qualsiasi algoritmo. Astraendo molto dai complessi dettagli formali, la cosa non dovrebbe creare incredulità in quanto un qualsiasi algoritmo viene implementato usando le operazioni logiche di base, AND, OR e NOT, che sono i componenti di base di un circuito.

Una pietra miliare dello studio dei problemi $\mathcal{NP}$-completi è il lavoro di Cook "The Complexity of Theorem-Proving Procedures", pubblicato nel 1971 che prova che SAT è $\mathcal{NP}$-completo. La scoperta dell'esistenza di problemi $\mathcal{NP}$-completi è attribuita sia a Cook, per il lavoro già citato, che a Levin che ha provato indipendentemente da Cook l'esistenza di problemi $\mathcal{NP}$-completi, incluso SAT, in un lavoro del 1973 "Universal Search Problems" (pubblicato in russo, "Universal'nye zadachi perebora"). Spesso si parla di teorema di Cook-Levin per indicare la $\mathcal{NP}$-completezza di SAT. Il problema CIRCUITSAT è chiaramente simile a SAT. Una prova della $\mathcal{NP}$-completezza di CIRCUITSAT può essere trovata nel libro CLRS2009 [8] (nel capitolo che tratta la $\mathcal{NP}$-completezza).

Lemma 1.6.3 *Sia A un problema $\mathcal{NP}$-completo. Sia B un problema in $\mathcal{NP}$. Se $A \leq_P B$, allora anche B è $\mathcal{NP}$-completo.*

DIMOSTRAZIONE. Poichè A è $\mathcal{NP}$-completo, un qualsiasi altro problema X può essere ridotto ad A, cioè $X \leq_P A$. Poichè $A \leq_P B$, per la proprietà transitiva della riduzione si ha che $X \leq_P B$. Quindi un qualsiasi problema $X \in \mathcal{NP}$ può essere ridotto a B, pertanto B è $\mathcal{NP}$-completo. □

Il precedente lemma e il fatto che CIRCUITSAT sia $\mathcal{NP}$-completo, implicano la $\mathcal{NP}$-completezza di tutti i problemi che abbiamo studiato nella sezione 1.5. La seguente figura riassume le riduzioni viste in precedenza.

$$\text{CIRCUITSAT} \leq \text{3SAT} \leq \begin{cases} \text{SAT} \\ \text{GRAPHCOLORING} \\ \text{INDEPENDENTSET} \leq \begin{cases} \text{VERTEXCOVER} \leq \text{SETCOVER} \\ \text{SETPACKING} \end{cases} \\ \text{HAMCYCLE} \leq \text{TSP} \end{cases}$$

Dal fatto che CIRCUITSAT è $\mathcal{NP}$-completo tale catena di riduzioni prova che tutti i problemi visti sono $\mathcal{NP}$-completi (ricordiamo che per ognuno di essi è facile fornire un verificatore efficiente, quindi tutti appartengono a $\mathcal{NP}$).

1.7 *Problemi decisionali e di ottimizzazione*

Nella sezione 1.2 abbiamo discusso brevemente della relazione fra la versione decisionale di un problema e quella di ottimizzazione. Ricordiamo che si parla di problema *decisionale* quando la risposta che cerchiamo è un sì o un no alla domanda "esiste una soluzione?" e di problema di *ricerca* quando cerchiamo una soluzione al problema; nel caso dei problemi di ottimizzazione cerchiamo una soluzione ottima. In precedenza abbiamo detto che il limitarsi a considerare problemi decisionali non è una grossa restrizione. Possiamo ora approfondire un po' di più questa asserzione. Abbiamo già osservato che se sappiamo risolvere in maniera efficiente la versione decisionale di un problema, sappiamo anche dire, sempre in modo efficiente, quale è il valore di una

soluzione ottima al corrispondente problema di ottimizzazione: basta fare una ricerca binaria sul valore della soluzione ottima e con un numero logaritmico di ripetizioni dell'algorimo scopriamo il valore della soluzione ottima.

Non è stato provato che la versione decisionale è sempre equivalente alla corrispondente versione di ricerca/ottimizzazione, ma per la maggior parte dei problemi per i quali si conosce un algoritmo efficiente si è capaci sia di risolvere in modo efficiente la versione decisionale che il corrisponde problema di ricerca/ottimizzazione.

Un caso in cui questo non è vero è il test di primalità. Nella versione decisionale occorre stabilire se un numero n è primo oppure no, cioè se è il prodotto di fattori. La versione di ricerca di questo problema è quella della ricerca dei fattori di n; in questo caso non c'è nulla da ottimizzare! Si è a lungo creduto che il test di primalità non appartenesse alla classe $\mathcal{P}$. Nel 2002 però è stato dimostrato, da Agrawal, Kayal e Saxena [1], che è possibile effettuare il test di primalità con un algoritmo deterministico polinomiale. Per il corrispondente problema di ricerca, cioè quello di trovare i fattori di n, invece, come abbiamo detto anche nell'introduzione di questo capitolo, non si conosce un algoritmo efficiente. Ovviamente se si dovesse dimostrare che $\mathcal{P} = \mathcal{NP}$ anche il problema di trovare i fattori di un numero verrebbe risolto in tempo polinomiale.

Dunque per molti problemi sappiamo risolvere in maniera efficiente sia la versione decisionale che quella di ricerca. Ci sono dei casi in cui sappiamo risolvere in maniera efficiente la versione decisionale ma non sappiamo se la versione di ricerca può essere risolta in maniera efficiente. La cosa è legata alla questione $\mathcal{P}$-$\mathcal{NP}$.

Infine osserviamo che per i problemi $\mathcal{NP}$-completi la versione decisionale è equivalente alla versione di ricerca. Infatti, la versione decisionale è automaticamente risolta dalla versione di ricerca, mentre la ricerca di una soluzione si può spesso fare usando come subroutine la versione decisionale sfruttando tecniche come ricerca binaria o tentativi incrementali.

Come esempio consideriamo il problema SAT. Sia A un algoritmo che risolve la versione decisionale in tempo $O(f(n))$, dove n è la grandezza del problema. L'algoritmo A ci dice se esiste un assegnamento delle variabili che soddisfi la formula di input $\phi = (x_1, x_2, \ldots, x_n)$. Possiamo fissare un valore per x_1 ($x_1 = 0$ oppure $x_1 = 1$) e ottenere una nuova istanza di SAT e usando l'oracolo possiamo sapere se esiste un assegnamento che rende vera la formula originale nei due casi considerati.

La cosa si può iterare fissando valori successivi fino a fissarli tutti. Scriveremo $A()$ per indicare l'utilizzo di A nella formula originaria e $A(0)$ o $A(1)$ o per indicare l'utilizzo di A su ϕ con x_1 fissato al valore specificato. Ovviamente potremo usare A fissando anche altri valori. Ad esempio $A(1, 0, 1)$ indica l'utilizzo di A con $x_1 = 1$, $x_2 = 0$ e $x_3 = 1$ in ϕ.

Possiamo sfruttare A per costruire un algoritmo B che trova una soluzione in tempo $O(n \cdot f(n))$. L'algoritmo B "trova" una soluzione fissando i valori delle singole variabili e "chiedendo" ad A se le scelte portano a un assegnamento che rende vera la formula. Ad esempio possiamo fissare $x_1 = 1$ e usare $A(1)$ per capire se esiste un assegnamento che renda vera ϕ con $x_1 = 1$. Se non esiste possiamo provare con $x_1 = 0$. Se per nessuno dei due casi esiste, allora la formula non è soddisfacibile. Possiamo ripetere il test per ogni variabile, riuscendo così a trovare una soluzione (se esiste) eseguendo per un numero polinomiale (lineare) di volte l'algoritmo decisionale.

Algorithm 1: Algoritmo B

for $i = 1, 2, \ldots, n$ **do**
  $s_i =$`null`
if $A(1)$ **then**
  $s_1 = 1$
else
  if $A(0)$ **then** $s_1 = 0$ **else return** `null`
if $A(s_1, 1)$ **then**
  $s_2 = 1$
else
  if $A(s_1, 0)$ **then** $s_2 = 0$ **else return** `null`
...
if $A(s_1, \ldots, s_{n-1}, 1)$ **then**
  $s_n = 1$
else
  if $A(s_1, \ldots, s_{n-1}, 0)$ **then** $s_n = 0$ **else return** `null`
return$(s_1, \ldots, s_n)$

L'esempio di SAT è particolarmente semplice in quanto fissando il valore di una variabile otteniamo una nuova istanza dello stesso problema SAT. Più in generale, partendo da un problema $\mathcal{NP}$-completo X, dovremo sfruttare il fatto che il "sottoproblema" che otteniamo fissando una parte della soluzione cercata può essere ridotto a X in quanto X è $\mathcal{NP}$-completo. Non daremo una prova formale ma solo l'intuizione. La prova si basa sulla rappresentazione tramite linguaggi. In particolare dovremo definire il linguaggio che corrisponde alle soluzioni del problema. Per capire questo passaggio definiamo un po' più formlamente la relazione fra versione decisionale e versione di ricerca di un dato problema X. La versione decisionale può essere vista come

$$\exists w \text{ tale che } \rho(x, w) = 1?$$

mentre la versione di ricerca come

$$\text{trova } w \text{ tale che } \rho(x, w) = 1,$$

per una fissata istanza x, dove ρ è la relazione che identifica le soluzioni per tale istanza di input. Dunque ρ è un insieme di stringhe, ognuna delle quali rappresenta una soluzione della specifica istanza. In altre parole ad ogni istanza x del problema X è associato un linguagggio L_x. Nella versione decisionale dobbiamo solo dire se L_x è vuoto o meno, mentre nella versione di ricerca dobbiamo trovare un elememto di L_x.

Per dimostrare che sono computazionalmente equivalenti dobbiamo far vedere che un problema di si riduce all'altro. Se si sa risovere la versione di ricerca è si sa risolvere anche la versione decisionale: saper trovare una soluzione, o dire che non esiste, risolve anche la versione decisionale. Il viceversa non è tanto ovvio. Quindi assumiamo di saper risolvere la versione decisionale, cioè di avere un oracolo per la versione decisionale del problema X. Dobbiamo sfruttarla per trovare un elemento $w \in L_x$.

Per trovarlo possiamo definire una serie di altri linguaggi $L_0 = \{0y | w = 0y \in L\}$, $L_1 = \{1y | w = 1y \in L\}$, $L_{00} = \{00y | w = 00y \in L\}$, $L_{01} = \{01y | w = 01y \in L\}$,

$L_{10} = 10y|w = 10y \in L$, $L_{11} = 11y|w = 11y \in L$, etc. L'idea è quella di trovare un elemento di L, "scoprendo" un bit alla volta, più o meno come per SAT abbiamo trovato il valore di una variabile alla volta. Per ogni bit dovrò "decidere" il problema L_{str}, dove *str* rappresenta la stringa di bit già trovati, che può essere ridotto a X in quanto X è $\mathcal{NP}$-completo.

1.8 Pseudopolinomialità

Consideriamo il problema della somma di un sottoinsieme, SUBSETSUM, che è formulato nel seguente modo: dati i numeri naturali $w_1, \ldots, w_n$ e un altro numero naturale W, vogliamo selezionare il sottoinsieme S tale che $\sum_{i \in S} w_i \leq W$ e, tenendo presente questo vincolo, tale che $\sum_{i \in S} w_i$ sia quanto più grande possibile. Questo problema, nella sua formulazione più generale è conosciuto come *problema dello zaino*. Nel problema dello zaino ogni w_i è il *peso* di un oggetto i che ha un valore v_i mentre W è il peso massimo che lo zaino può sopportare. L'obiettivo è mettere nello zaino un sottoinsieme di valore massimo con il vincolo di non superare il peso massimo. Nel problema SUBSETSUM si ha che $v_i = w_i$.

Ritornando al problema della somma di un sottoinsieme serve considerare una versione decisionale. Per poter fare ciò cambiamo leggermente il vincolo sul numero W e richiediamo che la somma dei numeri del sottoinsieme debba essere esattamente W. Questo perché vogliamo formulare il problema con una domanda del tipo "esiste" un sottoinsieme tale che ...; se il vincolo fosse $\leq$ la risposta sarebbe sempre "sì" (il sottoinsieme vuoto soddisferebbe sempre il vincolo).

Problema 1.8.1 SUBSETSUM*: Dati i numeri naturali $w_1, \ldots, w_n$ e un altro numero W, esiste un sottonsieme di $\{w_1, \ldots, w_n\}$ la cui somma è esattamente W?*

Per trovare il valore di W più grande possibile possiamo usare una ricerca binaria. Il problema SUBSETSUM può essere risolto con un algoritmo di programmazione dinamica che trova una soluzione in tempo $O(nW)$. Ricordiamo velocemente tale algoritmo.

Indichiamo con $S(i, j)$ la soluzione al sottoproblema $\{w_1, \ldots, w_i\}$ e somma pari a j, per $i = 1, 2, \ldots, n$ e $j = 0, 1, 2, \ldots, W$; $S(i, j) = 1$ quando la soluzione è `si` e $S(i, j) = 0$ quando la soluzione è `no`. Il valore $S(n, W)$ è la soluzione all'istanza di input. Con la programmazione dinamica dobbiamo riempire la matrice bidimensionale $S(i, j)$.

Per $j = 0$ chiaramente il problema è banalmente risolvibile in quanto l'insieme vuoto è una soluzione. Quindi si ha che $S(i, 0) = 1$ per tutti i valori di $i = 1, 2, \ldots, n$. Per la prima riga $S(1, j) = 1$ solo per $j = 0$ e $j = w_1$, mentre per tutte gli altri elementi il valore è 0. Per riempire il resto della matrice possiamo sfruttare la seguente osservazione. Data una soluzione al problema per $i - 1$ possiamo trovare una soluzione per i. Infatti, avendo a disposizione anche w_i, oltre a $w_1, \ldots, w_{i-1}$, possiamo o usare o non usare w_i. Se non lo usiamo, allora l'unica possibilità è quella di sfruttare $S(i - 1, j)$, mentre se lo usiamo dobbiamo sfruttare $S(i - 1, j - w_i)$ insieme a w_i. pertanto la relazione di ricorrenza che permette di calcolare i valori della matrice S, per le riche successive alla

prima, è

$$S(i,j) = \begin{cases} S(i-1,j), \text{ se } w_i < j \\ S(i-1,j) \texttt{ or } S(i-1,j-w_i) \text{ se } w_i \geq j. \end{cases}$$

Consideriamo un esempio: $n = 5$, $w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 7, w_5 = 10$ e $W = 15$. La matrice S è la seguente:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	1	0	1+	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1-	1+	0	1+	0	0	0	0	0	0	0	0	0	0
4	1	0	1-	1-	1+	1-	1+	1+	0	1+	0	0	0	0	0	0
7	1	0	1-	1-	1-	1-	1-	1*	0	1*	1+	1+	1+	1+	1+	0
10	1	0	1-	1-	1-	1-	1-	1-	0	1-	1*	1-	1*	1*	1*	1+

dove 1- indica che il corrispondente w_i non è stato usato nella soluzione, mentre 1+ indica che w_i è stato usato; 1* indica che sono possibili entrambe le soluzioni, con e senza w_i.

È facile verificare che l'algoritmo può essere implementato in $O(nW)$ in quanto bisogna solo riempire la matrice S. Perchè quindi lo stiamo prendendo in considerazione ora che parliamo di problemi difficili da risolvere? Il tempo di esecuzione dell'algoritmo di programmazione dinamica, $O(nW)$, non è polinomiale nel senso più stretto del termine. La difficoltà è dovuta alla presenza di W nel tempo di esecuzione. Se W è piccolo allora non è un problema, ma se è grande lo diventa. Ad esempio, assumendo di avere a che fare con $n = 100$ numeri rappresentabili con 100 bit ognuno, la lunghezza dell'input è di 10.100 bit (i 100 numeri più W), ma il valore di W può essere dell'ordine di 2^{100}, e quindi l'algoritmo impiegherebbe un tempo lunghissimo. Questo perchè il valore di W è di fatto *esponenziale* nella grandezza dell'input (il numero di bit necessari a rappresentarlo). In queste situazioni si parla di tempo *pseudopolinomiale*.

Quindi la questione che vogliamo affrontare è: possiamo risolvere SubsetSum in tempo (puramente) polinomiale? In altre parole, esiste un algoritmo con tempo polinomiale in n e $\log W$, cioè polinomiale in n? La risposta è no[8]! Non vedremo la prova, che ovviamente si basa su una riduzione di un problema $\mathcal{NP}$-completo a SubsetSum, ma si ha che:

[8] A meno che $\mathcal{P}$ non sia uguale a $\mathcal{NP}$.

Lemma 1.8.2 SubsetSum *è $\mathcal{NP}$-completo.*

Vale la pena rimarcare ancora una volta che il problema SubsetSum è "difficile" in quanto W può assumere valori esponenziali in n. Nei casi in cui W è polinomiale in n, il problema diventa risolvibile in tempo polinomiale in quanto $O(nW)$ diventa polinomiale in n.

1.9 Co-NP e l'asimmetria di NP

Dato un problema X, definiamo il problema complementare $\overline{X}$ nel seguente modo: data una stringa di input s, $s \in \overline{X}$ se e solo se $s \notin X$. La classe Co-$\mathcal{NP}$ è definita come segue.

Definizione 1.9.1 *Co-$\mathcal{NP}$ è l'insieme dei problemi X tali che $\overline{X} \in \mathcal{NP}$.*

Riflettendo sulla definizione di verificatore che abbiamo usato per la classe $\mathcal{NP}$, volendola applicare a $\overline{X}$, il verificatore *di esistenza* di una soluzione diventerebbe il verificatore della *non* esistenza di una soluzione. Per capire meglio questa affermazione consideriamo uno specifico problema, ad esempio $X = \textsc{HamCycle}$. Abbiamo che

$$X = \{1010011..., 0100110..., \ldots\}$$

dove ognuno delle stringhe è una codifica di un grafo di input che ammette un ciclo hamiltoniano, e, per definizione si ha che

$$\overline{X} = \{\text{tutte le altre stringhe}\}.$$

Una qualsiasi stringa di $\overline{X}$ è la codifica di un grafo che non ammette un ciclo hamiltoniano (o una stringa che non ha significato, che comunque è un qualcosa che non ammette un ciclo hamiltoniano). Sia s una stringa binaria che rappresenta un grafo di input. Come si fa a stabilire se $s \in X$ oppure se $s \in \overline{X}$? Se $s \in X$ basta fornire un ciclo hamiltoniano per s, cioè una soluzione. In questo senso il certificato che serve a convincere il verificatore è una soluzione che il verificatore deve solo controllare. Se invece $s \in \overline{X}$ (e quindi non ci sono soluzioni) fornire un "non-ciclo" non convince sul fatto che non esistano cicli hamiltoniani per s. (Se la stringa s non è la codifica di un grafo allora banalmente la stringa appartiene a $s \in \overline{X}$ senza bisogno di nessuna certificazione.)

Dunque, con la classe Co-$\mathcal{NP}$ stiamo individuando quei problemi per i quali è facile dire che non esiste una soluzione. Osserviamo subito che per provare che esiste (almeno) una soluzione a un problema è sufficiente fornirne una, mentre provare che un problema non ammette soluzioni è molto più difficile in quanto una singola *non*-soluzione non potrà servire come "prova". Un modo per provare che non esiste nessuna soluzione è analizzarle tutte, ma questo ovviamente può richiedere tempo esponenziale e quindi non è efficiente, mentre il verificatore deve essere efficiente. Si noti come questo crei un'asimmetria fra queste classi.

Se un problema X appartiene a $\mathcal{P}$, allora anche $\overline{X} \in \mathcal{P}$. Infatti avendo a disposizione un algoritmo polinomiale che risolve X, lo stesso algoritmo permette di risolvere $\overline{X}$. Ricordando che stiamo trattando problemi decisionali, la cui soluzione è un `sì` oppure un `no`, per risolvere $\overline{X}$ basterà invertire la risposta ottenuta risolvendo X. Quindi anche $\overline{X} \in \mathcal{P}$.

Lo stesso ragionamento non vale per $X \in \mathcal{NP}$. Infatti in questo caso abbiamo a disposizione un algoritmo che verifica una soluzione di X in tempo polinomiale. Questo non ci dice niente su $\overline{X}$, almeno non in tempo polinomiale. Quindi la seguente domanda sorge spontanea:

Problema 1.9.2 *$\mathcal{NP}$= Co-$\mathcal{NP}$?*

Lo studio di Co-$\mathcal{NP}$ è strettamente legato alla questione $\mathcal{P}$-$\mathcal{NP}$, infatti si ha che:

Lemma 1.9.3 *Se $\mathcal{NP} \neq$ Co-$\mathcal{NP}$, allora $\mathcal{P} \neq \mathcal{NP}$.*

DIMOSTRAZIONE. Assumiamo che $\mathcal{NP} \neq$ Co-$\mathcal{NP}$. Procediamo per assurdo assumendo che $\mathcal{P} = \mathcal{NP}$. Allora si avrebbe che

$$X \in \mathcal{NP} \implies X \in \mathcal{P} \implies \overline{X} \in \mathcal{P} \implies \overline{X} \in \mathcal{NP} \implies X \in \text{Co-}\mathcal{NP}$$

ma anche che

$$X \in \text{Co-}\mathcal{NP} \implies \overline{X} \in \mathcal{NP} \implies \overline{X} \in \mathcal{P} \implies X \in \mathcal{P} \implies X \in \mathcal{NP}.$$

Quindi si avrebbe che $\mathcal{NP} \subseteq$Co-$\mathcal{NP}$ e Co-$\mathcal{NP} \subseteq \mathcal{NP}$ e quindi che $\mathcal{NP} =$ Co-$\mathcal{NP}$. Questo è un assurdo perchè contraddice l'ipotesi di partenza. □

Poichè la definizione di Co-$\mathcal{NP}$ è asimmetrica (è facile certificare l'esistenza di una soluzione, basta verificarla, ma non è altrettanto facile certificare che non esiste una soluzione) è interessante chiedersi se esistano problemi che effettivamente appartengono a $\mathcal{NP} \cap$Co-$\mathcal{NP}$.

Determinare se una rete ammette un flusso di valore almeno v è uno di questi. Il problema appartiene a $\mathcal{NP}$ in quanto è facile fornire un certificato che provi che esiste una soluzione: il certificato è un flusso con valore almeno v. In questo caso è possibile anche fornire un certificato, verificabile in tempo polinomiale, che mostra che non esiste un tale flusso: il certificato è un taglio di capacità più piccola di v; per il teorema del massimo flusso e minimo taglio, si ha che non può esistere un flusso di valore più grande. Quindi detto X il problema, si ha che $X \in \mathcal{NP}$, ma anche $X \in$Co-$\mathcal{NP}$ in quanto $\overline{X} \in \mathcal{NP}$.

In realtà, qualunque problema $X \in \mathcal{P}$, ha la proprietà che $X \in \mathcal{NP}$ e $X \in$Co-$\mathcal{NP}$. Quindi si ha che

$$\mathcal{P} \subseteq \mathcal{NP} \cap \text{ Co-}\mathcal{NP}.$$

Un'altra questione aperta è

Problema 1.9.4 *$\mathcal{P} = \mathcal{NP} \cap$ Co-$\mathcal{NP}$?*

1.10 PSPACE

La classi $\mathcal{P}$ e $\mathcal{NP}$ classificano i problemi in funzione del tempo necessario a risolverli. Una classificazione simile può essere fatta considerando lo *spazio* (cioè la memoria) necessaria a risolvere un problema. La classe PSPACE è l'analogo di $\mathcal{P}$, ma considerando lo spazio al posto del tempo: PSPACE è l'insieme dei problemi che possono essere risolti usando spazio polinomiale. Si noti come nella definizione non venga menzionato il tempo, quindi per risolvere un problema in PSPACE si può usare quanto tempo si vuole!

Il seguente fatto è immediato.

Lemma 1.10.1 *$\mathcal{P} \subseteq$ PSPACE.*

DIMOSTRAZIONE. Sia X un problema in $\mathcal{P}$. Allora esiste un algoritmo che in tempo polinomiale risolve X. Lo stesso algoritmo risolve X usando spazio polinomiale: infatti

poichè X usa tempo polinomiale può utilizzare al massimo un numero polinomiale di celle di memoria, quindi usa spazio polinomiale. Pertanto $X \in$ PSPACE. □

PSPACE però è più ampio di $\mathcal{P}$. Il punto cruciale è che avendo a disposizione tempo illimitato è possibile riusare la memoria. Ad esempio, possiamo risolvere il problema 3SAT usando spazio polinomiale

Lemma 1.10.2 3SAT$\in$ PSPACE.

DIMOSTRAZIONE. Possiamo usare un approccio a forza bruta, facendo attenzione a riusare la memoria utilizzata in modo tale da non usare mai più di un numero polinomiale di celle di memoria.

In particolare, se la formula è definita su n variabili, useremo n celle di memoria (in realtà bastano esattamente n bit) per specificare una soluzione.

Ogni possibile soluzione può essere rappresentata come una sequenza binaria di n bit, dove l'i-esimo bit rappresenta il valore della variabile x_i: 1 rappresenta vero e 0 rappresenta falso. Qundi usando un contatore con n-bit riusciamo ad enumerare tutti i 2^n possibili assegnamenti delle variabili.

Per ognuno degli assegnamenti controlleremo il valore della formula, e questo può essere fatto con un numero costante di celle di memoria addizionali (ad esempio, potremmo valutare la prima clausola e memorizzare il valore in una variabile, poi procedere valutando le altre clausole se e solo se tutte le precedenti hanno valore vero).

Quindi riusciremo a controllare il valore della formula su tutti i possibile assegnamenti usando tempo esponenziale ma spazio polinomiale. □

Il lemma precedente ha una importante conseguenza:

Lemma 1.10.3 $\mathcal{NP}\subseteq$ PSPACE.

DIMOSTRAZIONE. Sia X un qualsiasi problema in $\mathcal{NP}$. Poichè 3SAT è un problema $\mathcal{NP}$-completo, si ha che $X \leq_P$ 3SAT, e quindi esiste un algoritmo che risolve X usando un numero polinomiale di passi e un oracolo che risolve 3SAT (che può essere chiamato un numero polinomiale di volte). Ma dal lemma precedente sappiamo che possiamo risolvere 3SAT usando spazio polinomiale. Quindi se sostituiamo all'oracolo un algoritmo che usa spazio polinomiale per risolvere 3SAT otteniamo un algoritmo che risolve X in spazio polinomiale. Pertanto $X \in$ PSPACE. □

Si noti che come per $\mathcal{P}$ si ha che $X \in \mathcal{P} \equiv \overline{X} \in \mathcal{P}$, anche per PSPACE si ha che $X \in$ PSPACE $\equiv \overline{X} \in$ PSPACE.

Lemma 1.10.4 *Co-*$\mathcal{NP}\subseteq$ PSPACE.

DIMOSTRAZIONE. Sia $X \in$ Co-$\mathcal{NP}$. Allora $\overline{X} \in \mathcal{NP}$. Dal lemma precedente si ha quindi che $\overline{X} \in$ PSPACE e quindi $X \in$ PSPACE. □

La Figura 1.21, riassume queste inclusioni fra le classi. Come per la questione $\mathcal{P}$-$\mathcal{NP}$, anche PSPACE è un mistero irrisolto: sebbene si creda che PSPACE sia molto più ampio di $\mathcal{P}$, e che quindi esistano problemi appartenenti a PSPACE ma non a $\mathcal{P}$ o anche problemi non appartenenti $\mathcal{NP}$ o Co-$\mathcal{NP}$, non si sa se $\mathcal{P} \neq$ PSPACE.

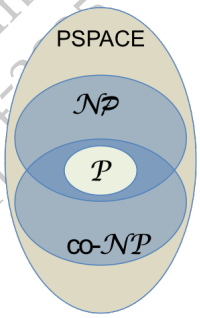

Figura 1.21: Classe PSPACE

1.10.1 *QSAT*

Come esempio di problema in PSPACE vediamo ora un problema legato a 3SAT, il problema QSAT in cui oltre a una formula da soddisfare abbiamo dei quantificatori per ogni variabile. Sia $\Phi(x_1, \ldots, x_n)$ una formula booleana nella forma

$$C_1 \cdot C_2 \ldots \cdot C_k$$

dove ogni clausola C_i è la disgiunzione (l'OR) di 3 letterali. In altre parole, la formula è un'istanza di 3SAT. Assumiamo per semplicità che n sia dispari, il problema QSAT chiede se

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n) = 1?$$

Cioè, vogliamo sapere se possiamo scegliere un valore per x_1 in modo tale che per tutti i valori di x_2, esiste una scelta per x_3, e così via che rende vera la formula Φ.

Il problema 3SAT, usando i quantificatori verrebbe codificato come:

$$\exists x_1 \exists x_2 \ldots \exists x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n) = 1?$$

Un esempio concreto di istanza di QSAT è il seguente. Supponiamo che

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

e chiediamo se

$$\exists x_1 \forall x_2 \exists x_3 \Phi(x_1, x_2, x_3) = 1.$$

La risposta è sì. Infatti possiamo scegliere $x_1 = 1$. A questo punto se $x_2 = 1$, scegliamo $x_3 = 0$ rendendo vera Φ, mentre se $x_2 = 0$ scegliamo $x_3 = 1$ rendendo vera Φ.

1.10.2 *Un algoritmo per QSAT*

Il problema QSAT può essere risolto in spazio polinomiale. Possiamo usare un approccio molto simile a quello usato per provare che 3SAT può essere risolto in spazio polinomiale. La difficoltà aggiuntiva deriva dalla presenza dei quantificatori. Per 3SAT è sufficiente trovare un assegnamento che rende vera la formula (tutti i quantificatori sono $\exists$). Per tenere conto dei quantificatori possiamo utilizzare un approccio ricorsivo. Se il primo quantificatore della formula è $\exists x_i$, consideriamo entrambi i possibili valori in sequenza. Quindi poniamo $x_i = 0$ e ricorsivamente vediamo se la rimanente parte della formula vale 1. Poi facciamo lo stesso per $x_i = 1$. Basta che in uno solo dei casi la formula valga 1 per ottenere una soluzione. Nel caso invece in cui dobbiamo gestire come primo quantificatore $\forall x_i$, allora procediamo come prima a valutare ricorsivamente entrambe le possibilità per x_i, ma questa volta la formula è vera solo se entrambi i sottoproblemi sono veri. Una descrizione informale dell'algoritmo è la seguente.

Algorithm 2: Algoritmo per QSAT

```
x_i = 0
Risolvi il problema ricorsivamente sulle altre variabili.
a =risultato della chiamata ricorsiva
Libera la memoria usata per la computazione intermedia
x_i = 1
Risolvi il problema ricorsivamente sulle altre variabili.
b =risultato della chiamata ricorsiva
Libera la memoria usata per la computazione intermedia
if Il primo quantificatore è ∃x_i then
    if a = 1 oppure b = 1 then
        return 1;
    else
        return 0;
if Il primo quantificatore è ∀x_i then
    if a = b = 1 then
        return 1;
    else
        return 0;
```

Analizziamo l'utilizzo di spazio di memoria. Indichiamo con $S(n)$ lo spazio necessario a risolvere un'istanza con n variabili. L'algoritmo ricorsivo effettua 2 chiamate su problemi di taglia $n-1$. Se stessimo valutando il tempo dovremmo avere una relazione di ricorrenza del tipo $S(n) = 2S(n-1) + f(n)$. Ma stiamo valutando lo spazio e la seconda chiamata ricorsiva (ri)utilizza lo stesso spazio usato dalla prima quindi la relazione di ricorrenza sarà

$$S(n) \leq S(n-1) + p(n)$$

dove $p(n)$ è un polinomio in n che tiene conto delle variabili locali necessarie a far funzionare l'algoritmo (di fatto serve solo memorizzare i risultati delle 2 chiamate ricorsive). Quindi, risolvendo la relazione di ricorrenza, abbiamo

$$S(n) \leq p(n) + p(n-1) + p(n-2) + \ldots + p(1) \leq np(n).$$

Pertanto l'algoritmo proposto risolve QSAT usando spazio polinomiale.

1.10.3 PSPACE-*completi*

In modo analago a quanto fatto per $\mathcal{NP}$, possiamo definire i problemi PSPACE-completi. Non lo dimostreremo, ma:

Lemma 1.10.5 QSAT *è* PSPACE-*completo.*

1.11 *Note bibliografiche*

Gli argomenti trattati in questo capitolo sono principalmenti tratti da CLRS2009 [8] e KT2014 [20] che possono essere consultati per approfondimenti. La NP-completezza del problema SAT è stata provata indipendentemente da Cook [7] e Levin [25]. Una prova della $\mathcal{NP}$-completezza di CIRCUITSATpuò essere trovata in CLRS2009 [8] (Lemma 34.6) mentre KT2014 [20] fornisce delle motivazioni intuitive a supporto del risultato. Sebbene datato, GJ1979 [14] rimane un'ottima guida alla NP-completezza e un buon catalogo di problemi NP-completi (ovviamente aggiornato al 1979). L'appartenenza di QSAT a PSPACE-completi è stata provata nel 1972 da Stockmeyer e Meyer. QSAT è stato il primo problema ad essere stato provato PSPACE-completo. Quindi è l'analogo di SAT per $\mathcal{NP}$. Per molti altri problemi si è provato l'appartenenza a PSPACE-completi riducendo QSAT ad essi.

1.12 *Esercizi*

1. Provare che VERTEXCOVER$\leq_P$INDEPENDENTSET (dimostrazione del Lemma 1.5.9).
2. Provare che INDEPENDENTSET$\leq_P$SETPACKING (dimostrazione del Lemma 1.5.13).
3. Dato un grafo $G = (V, E)$, un *clique* (cricca) è un sottoinsieme $C \subseteq V$ tale che E contiene un arco per ogni coppia di nodi di C, cioè C è un sottografo completo. Il problema CLIQUE consiste nell'individuare una clique di taglia massima.
   (a) Fornire una rappresentazione binaria che permetta di descrivere le istanze del problema CLIQUE.
   (b) Indicare il parametro più significativo in relazione alla grandezza del problema e argomentare sulla sua relazione rispetto alla lunghezza della stringa ottenuta con la rappresentazione precedente.
   (c) Fornire la versione decisionale del problema CLIQUE.
   (d) Adattare la rappresentazione binaria per la versione decisionale.
4. Fornire un verificatore per ognuno dei problemi studiati in questo capitolo (CIRCUITSAT, SAT, 3SAT, INDEPENDENTSET, VERTEXCOVER, SETCOVER, SETPACKING, GRAPHCOLORING, GRAPHCOLORING, HAMCYCLE, TSP). Analizzare il tempo necessario alla verifica della potenziale soluzione argomentando la polinomialità rispetto alla taglia dell'input.
5. Si provi che INDEPENDENTSET$\leq_P$SETPACKING (Lemma 1.5.13).
6. Si provi che SAT$\leq_P$3SAT
7. Si consideri il seguente problema: dati due grafi G_1 e G_2 stabilire se G_1 e G_2 sono isomorfi (cioè uguale a meno di una ridenominazione dei nodi). Provare che tale problema appartiene alla classe $\mathcal{NP}$.
8. Per ridurre un problema A a un problema B per il quale si ha a disposizione un oracolo, quante volte si può invocare l'oracolo che risolve B? Si dia una motivazione della risposta.

9. Si consideri il grafo della Figura 1.22. Si trovi l'insieme indipendente più grande. Quale è il corrispondente insieme ricoprente?

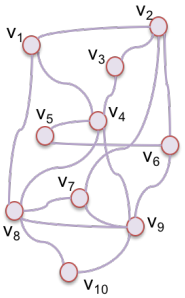

Figura 1.22: Grafo Esercizio 9

10. Abbiamo visto che VertexCover e IndependentSet sono equivalenti, nel senso che l'uno si riduce all'altro grazie al Lemma 1.5.7 che asserisce che un insieme $S \subset V$ è un insieme indipendente se e solo se $V \setminus S$ è un insieme ricoprente. Visto che il problema SetCover è una generalizzazione del problema VertexCover e che il problema SetPacking è una generalizzazione del problema IndependentSet, è abbastanza naturale pensare che SetCover e SetPacking potrebbero essere equivalenti. Una naturale generalizzazione del Lemma 1.5.7 adattata ai problemi SetCover e SetPacking è la seguente: un insieme $S_{i_1}, S_{i_2}, \ldots, S_{i_\ell}$ è ricoprente (cioè è una soluzione a SetCover) se solo se $S_{i_1}, S_{i_2}, \ldots, S_{i_m} \setminus S_{i_1}, S_{i_2}, \ldots, S_{i_\ell}$ è un insieme indipendente (cioè una soluzione a SetPacking). Fornire una prova di tale asserzione oppure mostrare che non è vera.

11. Si consideri il problema 4Sat, definito come 3Sat, ma con 4 letterali in ogni clausola. Si riduca 4Sat a IndependentSet.

12. Si consideri il circuito riportato nella figura 1.23. Si fornisca la formula 3Sat della riduzione da CircuitSat a 3Sat.

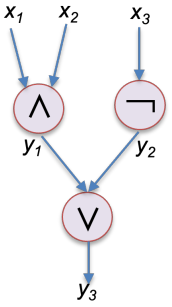

Figura 1.23: Circuito

13. (⋆) Si provi che il problema Clique, definito nell'Esercizio 3 è $\mathcal{NP}$-completo.

14. (⋆) Si provi che il problema SubsetSum è $\mathcal{NP}$-completo.

15. Un grafo $G = (V, E)$ è bipartito se V può essere partizionato in $V = V_1 \cup V_2$ in modo tale che gli archi hanno tutti un vertice in V_1 e uno in V_2. Provare che un grafo G è 2-colorabile se e solo se G è bipartito. Fornire un algoritmo efficiente per stabilire se un grafo è bipartito.

16. Nella riduzione di 3Sat a GraphColoring i 3 nodi corrispondenti ai 3 letterale di una clausola vengono uniti in un sottografo con 6 nodi fittizi. Quale è la proprietà garantita da tale sttografo? Riesci a ricostruire il sottografo?

17. Nella riduzione $3\text{Sat} \leq_P$ HamCycle, Lemma 1.5.19, sono stati usati dei nodi definiti "di transito". Quale è la loro funzione? Si fornisca una motivazione.

18. Abbiamo visto che per i problemi $\mathcal{NP}$-completi la versione decisionale è computazionalmente equivalente alla versione di ricerca. In particolare abbiamo visto come Sat sia *auto-riducibile*, cioà come è possibile riolvere la versione di ricerca di Sat usando come subroutine la versione deicisionale. Mostra come anche Clique sia auto-riducibile.