1. Teoria dell'NP-completezza

Per trascorrere il tempo durante un viaggio in aereo, ci si può essere cimentati con il seguente rompicapo, o con qualche sua variante semplificata:

DOMINO LIMITATO (SQUARE TILING). *Dati un intero positivo n ed un insieme finito D di m tipi di "tessere orientate", cioè di quadrati di lato unitario, divisi in quattro parti dalle diagonali con ciascun quarto colorato con un colore, è possibile ricoprire un'area quadrata di lato n con copie delle tessere in D in modo che:*

(1) nessuna tessera è "ruotata";
(2) una particolare tessera $d \in D$ occupa la posizione più in basso a sinistra;
(3) due tessere che si toccano hanno i lati adiacenti dello stesso colore?

Esempio (Domino limitato) La Figura 1.1 illustra due insiemi D e D', con $m = 3$ tipi di tessere ciascuno. Posto $n = 3$, è facile vedere che esiste una copertura legale con tessere di D (Figura 1.1(a)), mentre non ne esiste alcuna con tessere di D' (Figura 1.1(b), dove tutte le scelte effettuate sono obbligate e non c'è alcun modo di coprire la casella vuota). Si noti che se fossero ammesse rotazioni, allora la sola tessera d, qualunque essa fosse, sarebbe sufficiente a coprire ogni area di dimensione arbitraria! ■

Per dati di ingresso banali, come quelli dell'Esempio 1, il DOMINO LIMITATO può essere risolto a colpo d'occhio, ma per n ed m "grandi" e tessere "perverse", cercare una soluzione può essere alquanto frustrante. Ciò accade specialmente quando esistono molte soluzioni "parziali", che coprono cioè solo alcune porzioni del quadrato di lato n, ma ci sono pochissime soluzioni "complete" che coprono l'intero quadrato.

Un algoritmo banale che risolve il problema "provando tutte le possibilità" non è però difficile da trovare. Si osservi infatti che nel quadrato di lato n ci sono n^2 caselle da coprire con copie delle tessere. Per la casella in basso a sinistra la scelta è obbligata, mentre ognuna delle rimanenti caselle può essere coperta con una copia di una tessera scelta tra m possibili tipi diversi. Il numero di coperture distinte è pertanto m^k, con $k = n^2 - 1$, e l'algoritmo deve semplicemente generarle tutte e verificare, per ognuna di esse, che la copertura sia "legale", ovvero che tessere adiacenti abbiano

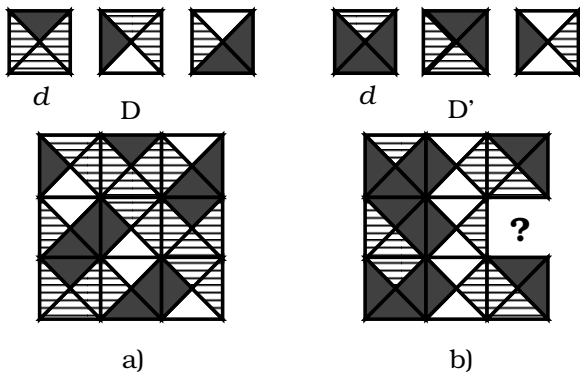

Figura 1.1: DOMINO LIMITATO ($m = 3$, $n = 3$). a) Insieme D di tessere con copertura (risposta SÌ); b) Insieme D' di tessere senza copertura (risposta NO).

i lati che si toccano dello stesso colore. Purtroppo, questo banale algoritmo non è utilizzabile in pratica, poiché richiede un inaccettabile tempo esponenziale!

Il DOMINO LIMITATO è il problema "più difficile" tra tutti quelli visti finora. Infatti, per quasi tutti i problemi trattati nei capitoli precedenti (per esempio CAMMINI MINIMI, ORDINAMENTO, INVILUPPO CONVESSO, STRING MATCHING), oltre ad algoritmi banali del tipo "prova tutte le possibilità", erano stati introdotti algoritmi molto efficienti se non addirittura ottimi. Per il DOMINO LIMITATO, al contrario, non solo non è conosciuto un algoritmo migliore di quello che "prova tutte le possibilità", ma si suppone addirittura che un tale algoritmo non esista! Nella stessa situazione stanno molti altri problemi, come, per esempio, i seguenti:

COLORAZIONE (COLORING). *Dati un grafo non orientato ed un intero k, è possibile colorarne i nodi usando al più k colori in modo che ogni nodo sia colorato con un colore diverso da tutti i nodi adiacenti?*

CRICCA (CLIQUE). *Dati un grafo non orientato ed un intero k, esiste un sottoinsieme di almeno k nodi tutti mutuamente adiacenti?*

COMMESSO VIAGGIATORE (TRAVELING SALESPERSON, TSP). *Date n città, le distanze tra esse, ed un intero k, è possibile partire da una città, attraversare ogni città esattamente una volta tornando alla città di partenza, percorrendo una distanza totale non superiore a k?*

PROGRAMMAZIONE LINEARE 0/1 (0/1 LINEAR PROGRAMMING). *Data una matrice A di elementi interi e di dimensione $m \times n$, ed un vettore b di m elementi interi, esiste un vettore x di n elementi* 0/1 *tale che $Ax \leq b$?*

SODDISFATTIBILITÀ (SATISFIABILITY). *Data un'espressione booleana in forma normale congiuntiva, esiste un'assegnazione di valori di verità vero/falso alle variabili booleane che rende l'espressione vera?*

Esempio (COLORAZIONE) La Figura 1.2 mostra una colorazione con 3 colori per un grafo che rappresenta la relazione di adiacenza tra le 10 province della Toscana. Per ogni k maggiore o uguale a 3 il problema ha risposta SÌ, mentre per k minore di 3 la risposta è NO. ■

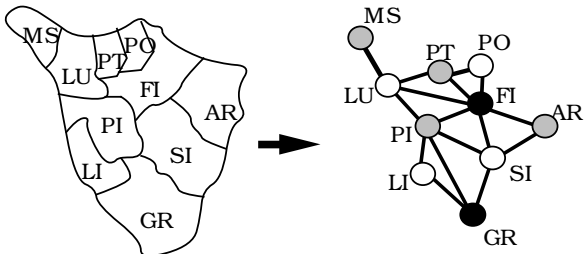

Figura 1.2: Una colorazione con 3 colori del grafo che rappresenta la relazione di adiacenza tra province della Toscana. **TODO**: Provincia di Grosseto "tagliata", da "riparare"

Esempio (CRICCA) Nel grafo illustrato nella Figura 1.2, esiste una cricca con 3 nodi (per esempio {FI, PT, PO} o {PI, GR, SI}), ma non esiste alcuna cricca con più di 3 nodi. ■

Esempio (COMMESSO VIAGGIATORE) La Figura 1.3 mostra la tabella delle distanze (in linea d'aria) tra sette località della Toscana ed un percorso ottimo per il commesso viaggiatore. Per ogni k inferiore alla distanza ottima il problema ha risposta NO, mentre per ogni k maggiore o uguale a tale distanza la risposta è ovviamente SÌ. ■

Esempio (PROGRAMMAZIONE LINEARE 0/1) Il sistema:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &\geq 2 \\ x_1 - x_2 - x_3 + x_4 &\geq 0 \\ x_1 \qquad + x_3 + x_4 &\leq 1 \end{aligned}$$

è verificato per $x_1 = x_2 = 1$ ed $x_3 = x_4 = 0$. ■

Esempio (SODDISFATTIBILITÀ) La formula su u_1, u_2 e u_3:

$$(\textbf{not } u_1 \textbf{ or not } u_2 \textbf{ or not } u_3) \textbf{ and } (u_2 \textbf{ or not } u_3) \textbf{ and } (u_1 \textbf{ or not } u_2 \textbf{ or } u_3)$$

è soddisfatta per $u_1 = \textbf{false}$ e $u_2 = u_3 = \textbf{true}$. ■

Oltre a questi, centinaia di altri problemi di rilevante interesse pratico si trovano nella stessa "barca". Tutti sono tra loro computazionalmente correlati e destinati ad "affogare" o a "salvarsi" insieme. Ognuno può essere risolto in tempo superpolinomiale, ma per nessuno è noto un algoritmo polinomiale. Inoltre, vedremo in questo capitolo che se uno solo fosse risolubile in tempo polinomiale, allora lo sarebbero tutti, ma si suppone che non esista alcun algoritmo polinomiale per nessuno di essi!

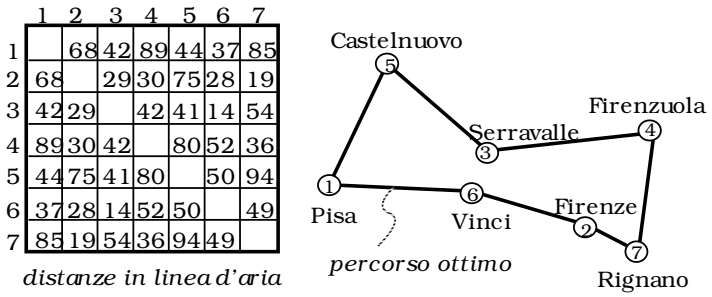

Figura 1.3: Il percorso ottimo per il COMMESSO VIAGGIATORE tra sette località della Toscana.

1.1 Certificati polinomiali

Oltre ad essere tutti di tipo decisionale, i problemi elencati precedentemente hanno in comune un'altra semplice proprietà. Benché sia estremamente difficile rispondere SÌ o NO, è abbastanza facile convincersi se un'ipotetica soluzione dia risposta SÌ. Formalmente, tutti questi problemi possiedono un certificato polinomiale, cioè un algoritmo che, data una presunta soluzione del problema, verifica in tempo polinomiale che tale soluzione sia effettivamente una soluzione che dà risposta SÌ.

Esempio (Certificato per DOMINO LIMITATO) Dati D, n, ed una copertura, è possibile scrivere un certificato che verifica in tempo $O(n^2)$ che tale copertura sia "legale", cioè che una tessera di tipo d copra l'angolo in basso a sinistra, che ogni tessera usata sia una copia di una in D, e che le 4 tessere ad essa adiacenti abbiano i lati colorati con gli opportuni colori. ■

Esempio (Certificato per CRICCA) Dati $G = (V,E)$, k ed un insieme S sottoinsieme di V, il certificato certificatoCricca() verifica che per ogni coppia $u,v \in S$ risulti $[u,v] \in E$ e che la cardinalità di S sia maggiore o uguale a k. L'algoritmo opera in tempo $O(k^2)$ se l'insieme è realizzato con una lista, e in tempo $O(n^2)$ se è realizzato con un vettore di booleani. ■

boolean certificatoCricca(GRAPH G, SET S, **int** k)

```
if S.=ize() < k then return false
foreach u ∈ S do
    foreach v ∈ S − {u} do
        if v ∉ G.adj(u) then return false
return true
```

Esempio (Certificato per COMMESSO VIAGGIATORE) Dati k, la matrice D di dimensione $n \times n$ delle distanze tra le n città, ed una permutazione π di $1,2,\ldots,n$, certificatoTsp() verifica in tempo $O(n)$ che la somma delle distanze non superi k. ■

boolean certificatoTsp(**int**[][] D, **int** n, **int** k, **int**[] P)

int $t = D[P[n], P[1]]$
for $i = 2$ **to** n **do** $t = t + D[P[i-1], P[i]]$

return $t \leq k$

Un modo equivalente di descrivere questo fenomeno consiste nella possibilità di risolvere tutti questi problemi con algoritmi non deterministici di complessità polinomiale.

1.2 Non determinismo

Un *algoritmo non deterministico* è un algoritmo che, posto di fronte alla necessità di dover effettuare una "decisione", ha la "virtù magica" di scegliere sempre la strada giusta! In termini equivalenti, è come se l'algoritmo, di fronte a più alternative, le seguisse tutte contemporaneamente, generando più "copie" di se stesso. Ciascuna copia procede la computazione, indipendentemente dalle altre, seguendo una e una sola delle alternative possibili.

Esempio (Non determinismo) In un racconto dello scrittore francese Marcel Aymé, Sabina, la protagonista, ha il dono sovrannaturale di generare una nuova copia di se stessa ogni volta che vuole sedurre un nuovo amante. Le varie Sabine hanno avventure indipendenti le une dalle altre e, al termine del racconto, il numero di Sabine è salito a qualche migliaio! ■

Gli algoritmi non deterministici, pur di interesse teorico, non sono ovviamente utilizzabili in pratica. Per definirli in modo formale, si supponga di avere a disposizione le tre seguenti istruzioni elementari, ciascuna delle quali richieda $O(1)$ tempo di computazione:
choice(C)**,** che sceglie arbitrariamente un elemento dell'insieme finito C;
failure, che blocca la computazione in uno stato di "fallimento";
success, che blocca la computazione in uno stato di "successo".

Un algoritmo non deterministico è un algoritmo che, per risolvere un problema decisionale, utilizza queste tre istruzioni. Quando viene eseguita un'istruzione **choice**(C), l'algoritmo produce $|C|$ copie di se stesso, ed ogni copia procede indipendentemente dalle altre dopo aver "scelto" un elemento distinto di C. L'algoritmo fornisce risposta SÌ se e solo se esiste una sequenza di scelte che portano allo stato di "successo", mentre fornisce risposta NO altrimenti. Si noti che, per definizione, è data risposta SÌ se una qualsiasi copia esegue un'istruzione **success**, mentre è data risposta NO se tutte le copie eseguono un'istruzione **failure**!

Esempio (Algoritmo non deterministico per CRICCA) Sia S un vettore di n elementi booleani, con n uguale al numero di nodi del grafo$G = (V,E)$. Un algoritmo non deterministico per CRICCA è illustrato nella procedura ndCricca(). Nel ciclo **for** vengono effettuate le scelte ed il nodo u va a far parte della soluzione parziale se **choice**$(\{$**true**,**false**$\}) =$ **true**. A quel punto, non ci resta che verificare, tramite il certificato polinomiale, che l'insieme S così generato sia una cricca di dimensione k o maggiore. Il costo computazionale è $O(n^2)$, se per esempio l'insieme è realizzato con un vettore booleano. ■

Esempio (Algoritmo non deterministico per COMMESSO VIAGGIATORE) Presi in input la matrice delle distanze D, la dimensione del problema n e il valore k, sia S un vettore di n elementi che rappresenta la permutazione e sia C un insieme contenente le città non ancora

```
ndCricca(GRAPH G, int k)
    SET S = Set()
    for i = 1 to G.n do
        if choice({true, false}) then
            S.insert(i)
    if certificatoCricca(G, S, k) then success
    else failure
```

scelte, inizializzato con $C = \{1, 2, \ldots, n\}$. Un algoritmo non deterministico è illustrato nella procedura ndTsp(). Nel ciclo **for** vengono effettuate le scelte, e la città scelta $S[i]$ è eliminata dall'insieme C in modo che non possa essere più scelta. In base alla chiamata a certificatoTsp() si decide il successo o il fallimento della computazione. La complessità della procedura dipende da come è realizzato l'insieme C. Per esempio, se la cancellazione di $S[i]$ da C richiede $O(n)$ tempo, allora la complessità di ndTsp() risulta essere $O(n^2)$. ■

```
ndTsp(int[][] D, int n, int k)
    int[] S = new int[1...n]
    SET C = {1, 2, ..., n}
    for i = 1 to n do
        S[i] = choice(C)
        C.remove(S[i])
    if certificatoTsp(D, n, k, S) then success
    else failure
```

Come nel caso dell'enumerazione, un algoritmo non deterministico è caratterizzato da un albero delle scelte. I nodi interni dell'albero rappresentano "soluzioni parziali", gli archi rappresentano "scelte", e le foglie rappresentano un "successo" o un "fallimento". Ogni percorso radice-foglia dell'albero corrisponde ad una sequenze di scelte, ovvero alla computazione di una "copia" dell'algoritmo non deterministico.

Esempio (Albero delle scelte per ndCricca()) La Figura 1.4 mostra un grafo G e il corrispondente albero delle scelte ottenuto eseguendo ndCricca() su G per $k = 3$. ■

Gli algoritmi non deterministici operano una sequenza di scelte $S[1], S[2], \ldots$ per ottenere la soluzione. Tale sequenza di scelte è talmente correlata alla soluzione stessa da venire spesso identificata con essa. Ogni scelta $S[i]$ avviene su un sottoinsieme C che è funzione delle scelte precedenti $S[1, \ldots, i-1]$ effettuate dall'algoritmo. Supponendo per semplicità che il numero delle scelte sia limitato da n, lo "scheletro" generale di una procedura non deterministica è illustrato nella procedura nonDeterministica().

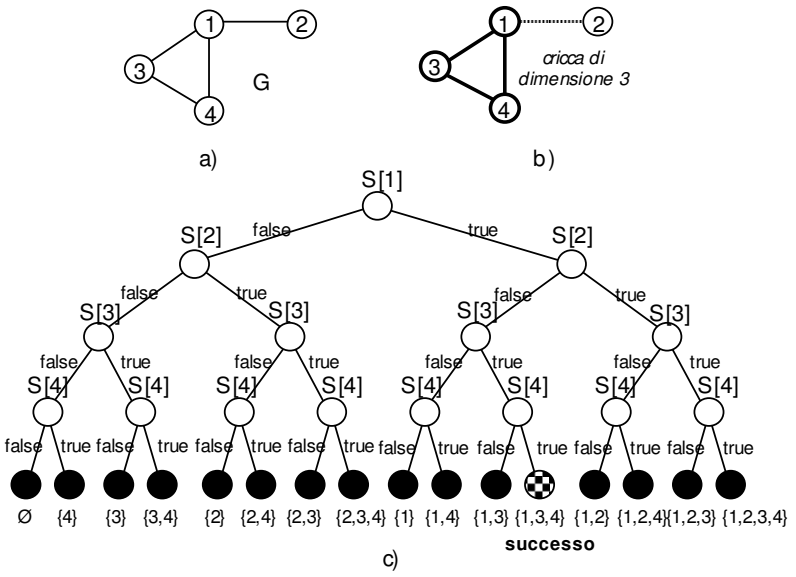

Figura 1.4: Procedura ndCricca(). a) Il grafo G; b) una cricca di dimensione 3; c) albero delle scelte: i nodi bianchi corrispondono ad esecuzioni di istruzioni **choice**, quelli neri ad esecuzioni di **failure**, e quello a scacchi a una **success**.

nonDeterministica({*opportuni parametri*})

```
int[] S = new int[1...n]
for i = 1 to n do
    SET C = choices(S,n,i,...)        % Determina C in funzione di S[1...i-1]
    if C = ∅ then
        failure
    else
        S[i] = choice(C)
        if S[1...i] è una soluzione then
            success
```

Si noti che la procedura è caratterizzata da due fasi distinte: nella prima fase viene costruita un'ipotetica soluzione, mentre nella seconda viene certificata. Se entrambe queste fasi richiedono tempo polinomiale, allora la complessità della procedura è polinomiale. Ciò accade per tutti i problemi precedentemente elencati!

1.3 Enumerazione

Poiché il non determinismo non è realistico, occorre "simulare" il comportamento di una procedura non deterministica con una deterministica. L'approccio da seguire è già noto: enumerazione basata su *backtrack*, per esplorare sistematicamente l'albero delle scelte corrispondente alla computazione deterministica. Vediamo quindi come utilizzare gli esempi visti nel Capitolo 16 per risolvere i problemi della CRICCA e del COMMESSO VIAGGIATORE.

Esempio (Enumerazione per CRICCA) Per CRICCA, possiamo partire dall'algoritmo per generare tutti gli insiemi di esattamente k membri (infatti, se esiste un insieme di nodi tutti mutuamente adiacenti e di dimensione maggiore di k, allora ne esiste uno di dimensione k) e utilizzare il certificato polinomiale per verificare se abbiamo trovato una cricca o meno; se questo avviene, l'enumerazione termina restituendo **true**, altrimenti l'enumerazione proverà tutti i possibili casi e restituirà **false**. Per semplificare lo pseudocodice e utilizzare la procedura certificatoCricca() definita in precedenza, utilizziamo un generico insieme senza dettagliare la sua realizzazione. La chiamata nel programma principale è enCricca$(G,k,S,1)$, dove S è un insieme vuoto. Sia che l'insieme sia realizzato con liste ordinate, liste non ordinate o vettori booleani, la complessità è $O(n^2 2^n)$. ■

```
boolean enCricca(GRAPH G, int k, SET S, int i)
    SET C = iif(S.size() < k and S.size() + (G.n − i + 1) ≥ k, {true, false}, ∅)
    foreach c ∈ C do
        if c then S.insert(i)
        if certificatoCricca(G, S, k) then return true
        if enCricca(G, k, S, i + 1) then return true
        if c then S.remove(i)
    return false
```

Esempio (Enumerazione per COMMESSO VIAGGIATORE) La procedura enTsp() descrive un algoritmo di enumerazione per il problema del COMMESSO VIAGGIATORE. Per semplificare il codice, assumiamo che la matrice delle distanze D, la dimensione del problema n e il parametro k siano variabili globali; non dovranno quindi essere passate nelle chiamate ricorsive. Il vettore S contiene una permutazione (parziale) della città da visitare; l'insieme C contiene le città ancora da visitare, ovvero le possibili scelte; l'indice i identifica la posizione in cui scrivere in S. L'algoritmo prova tutte le scelte rimanenti, rimuovendo la città da C e inserendola nel vettore S. Se la permutazione è completa e il certificato restituisce **true**, la computazione può terminare con successo. Altrimenti, si chiama ricorsivamente enTsp(); nel caso quest'ultimo restituisca **false**, si fa un passo di *backtrack* e si continua, fino all'esaurimento di tutte le permutazioni. La chiamata iniziale è enTsp$(S,C,1)$, dove $C = \{1,\ldots,n\}$. La complessità della procedura enTsp() è $O(nn!)$. Lasciamo per esercizio le possibili "potature" che possono essere applicate quando il costo di un cammino parziale eccede il valore k. ■

```
boolean enTsp(int[] S, SET C, int i)
    foreach c ∈ C do
        C.remove(c)
        S[i] = c
        if i = n and certificatoTsp(D, S, n, k) then return true
        if enTsp(S, C, i + 1) then return true
        C.insert(c)
    return false
```

1.4 Le classi $\mathbb{P}$ ed $\mathbb{NP}$

Per poter trattare in modo omogeneo problemi estremamente eterogenei tra loro (come problemi su grafi, algebrici, geometrici, numerici) facciamo alcune ipotesi semplificative:

(1) i problemi siano di tipo decisionale;
(2) il linguaggio di programmazione sia dotato delle operazioni **choice**, **success** e **failure**;
(3) la codifica dei numeri utilizzati nel calcolo sia *concisa*, ovvero si adotti una rappresentazione in base $b > 1$.

Vediamo di giustificare queste tre ipotesi e di mostrare come esse non siano affatto restrittive. Innanzitutto, un problema di tipo decisionale non può essere più difficile di uno di ricerca o di ottimizzazione. Pertanto, se si dimostra l'intrattabilità computazionale del problema in forma decisionale, allora si è dimostrata l'intrattabilità del problema formulato in uno degli altri due modi.

Esempio (Formulazioni per CRICCA) Il problema della CRICCA è stato formulato sotto forma decisionale. Nella formulazione di ricerca, si richiede di trovare una cricca di dimensione maggiore o uguale a k e in quella di ottimizzazione di trovare una cricca di dimensione massima. Ovviamente, se si risolve il problema di ricerca, allora si risolve anche quello decisionale, perché l'esibizione di una cricca è una prova della sua esistenza. Inoltre, se si risolve il problema di ottimizzazione, allora si risolvono anche gli altri due. Infatti, una cricca di dimensione massima h permette sia di rispondere al problema decisionale per ogni k, in quanto la risposta è SÌ per ogni $k \leq h$ e NO per ogni $k > h$, che di esibire una soluzione del problema di ricerca per ogni $k \leq h$ e di affermare che non esiste una soluzione per $k > h$. ■

In secondo luogo, sebbene non possiamo fornire qui una dimostrazione matematica, lo pseudocodice utilizzato in questo testo è un linguaggio di programmazione "universale", cioè non ci sono problemi risolubili con algoritmi scritti in linguaggi diversi dal nostro pseudocodice. Non solo, ma lo pseudocodice è "polinomialmente correlato" agli altri linguaggi universali, cioè non ci sono problemi di complessità polinomiale che non siano risolubili con procedure in pseudocodice anch'esse di complessità polinomiale (al più, può cambiare il grado del polinomio). Pertanto, non è restrittivo considerare lo pseudocodice, integrato dalle operazioni **choice**, **success** e **failure**.

Infine, è ben noto che, utilizzando i calcolatori elettronici, tutti i dati sono rappresentati alla fin fine con sequenze di bit. Comunque, rappresentare i dati di un problema in base 2 o in base 5 non influenza la risolubilità polinomiale del problema. Infatti, è facilmente verificabile come si possa passare in tempo polinomiale dalla rappresentazione in base 2 di un dato alla sua rappresentazione in base 5, e viceversa. In altri termini, finché si codificano numeri con basi $b > 1$, la dimensione dell'input è polinomialmente correlata alla dimensione dello stesso input rappresentata con un'altra base qualsiasi $b' > 1$. Questa proprietà invece non è verificata rappresentando l'input con una base unaria, in cui il numero k è codificato con una sequenza di k "uno". Evitando di usare la base $b = 1$, si impedisce che il problema diventi "difficile" solo a causa di un'inopportuna codifica non concisa.

Esempio (Codifica in base $b > 1$) Nell'ORDINAMENTO, abbiamo assunto la dimensione del problema pari al numero n di elementi da ordinare. A voler essere rigorosi, la dimensione dovrebbe essere $n \log_b a$, con $a = \max\{a_i\}$, poiché per codificare il numero a in base $b > 1$ occorrono $\log_b a$ caratteri.[a] Anche per a molto grande, comunque, la funzione $\log_b a$ cresce polinomialmente e le due dimensioni $d_1 = n$ e $d_2 = n \log_b a$ sono tra loro polinomialmente correlate. Per esempio, anche se il valore a del più grande elemento in ingresso cresce esponenzialmente col numero n di elementi da ordinare, cioè se $a = c^n$ con $c > 1$, allora $\log_b a$ è $O(n)$ e le due dimensioni d_1 e d_2 sono tali che d_2 è $O(d_1^2)$ mentre d_1 è $O(\sqrt{d_2})$. Usando la base $b = 1$, questo non è più vero. La codifica di a richiede a caratteri e la dimensione $d_3 = na$ non è più

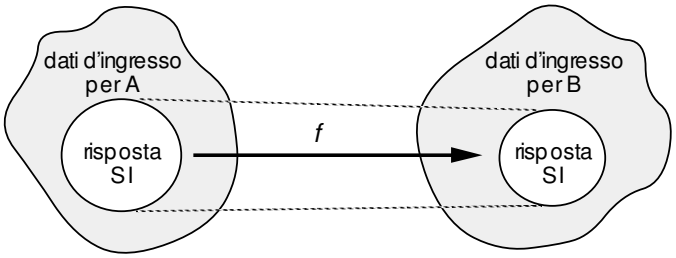

Figura 1.5: Riduzione da un problema A ad un problema B.

polinomialmente correlata né a $d_1 = n$ né a $d_2 = n\log_b a$. Infatti, per $a = c^n$, si ha $d_3 = na = nc^n$, che è esponenziale in n, e quindi d_3 è $O(d_1 n^{d_1})$. ■

[a] Nel provare la limitazione inferiore $\Omega(n\log n)$ per l'ORDINAMENTO, si è implicitamente assunto che i numeri a_i fossero codificati con una quantità costante di caratteri. Questa ipotesi, benché teoricamente scorretta, è realistica. Infatti, una parola di memoria di un moderno computer ha 64 bit, che permettono di codificare numeri grandi fino a $2^{63} - 1$ (un bit è usato per il segno). Numeri più grandi di $2^{63} - 1$ assai raramente si incontrano in pratica.

Possiamo così definire le classi $\mathbb{P}$ ed $\mathbb{NP}$ come segue:

– $\mathbb{P}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi deterministici;
– $\mathbb{NP}$ è la classe di tutti i problemi decisionali risolvibili in tempo polinomiale con algoritmi non deterministici.

Ovviamente, la classe $\mathbb{P}$ è contenuta nella classe $\mathbb{NP}$, poiché ogni algoritmo deterministico è anche un algoritmo non deterministico (che semplicemente non usa mai **choice**), ma non si sa se $\mathbb{P}$ sia propriamente contenuta in $\mathbb{NP}$ oppure se le due classi coincidano. Tale questione resta a tutt'oggi la più importante e famosa questione irrisolta dell'Informatica!

1.5 Riducibilità polinomiale

Tra problemi in $\mathbb{NP}$ è utile definire una relazione $\propto$ di riducibilità polinomiale. Siano A e B due problemi decisionali. Si dice che A si riduce in tempo polinomiale a B, e si scrive $A \propto B$, se esiste una funzione f di trasformazione

$$f\text{: (dati d'ingresso per } A) \to \text{(dati d'ingresso per } B)$$

tale che:

(1) f è computabile in tempo polinomiale con un algoritmo deterministico;
(2) x è un dato d'ingresso per cui A dà risposta SÌ se e solo se $f(x)$ è un dato d'ingresso per cui B dà risposta SÌ (si veda la Figura 1.5).

In pratica, se si dimostra che $A \propto B$, si trasforma un qualsiasi dato d'ingresso per A in un particolare dato d'ingresso per B e quindi A risulta essere, in un certo senso, un sottoproblema di B! Si hanno le seguenti conseguenze:

(1) Se $B \in \mathbb{NP}$, allora anche $A \in \mathbb{NP}$. Infatti, per risolvere il problema A con un algoritmo polinomiale non deterministico su un dato x, basta trasformare x in $f(x)$ e risolvere il problema B sul dato $f(x)$: l'algoritmo risultante è ovviamente polinomiale non deterministico.

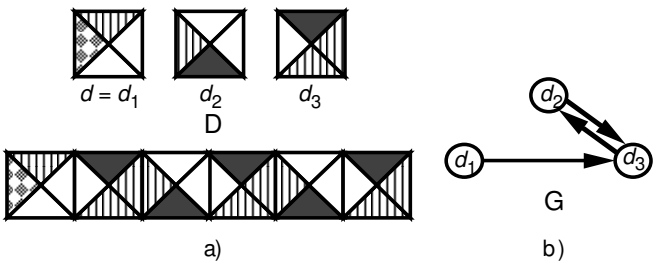

Figura 1.6: DOMINO LINEARE ($n = 6$, $m = 3$). a) Insieme D dei tipi di tessere con copertura (risposta SÌ). b) Il grafo G corrispondente con un ciclo raggiungibile da $d = d_1$.

(2) Se $B \in \mathbb{P}$, allora anche $A \in \mathbb{P}$. Infatti, si opera come nel caso precedente e poiché la trasformazione f è computabile con un algoritmo polinomiale deterministico, l'algoritmo risultante è ovviamente polinomiale deterministico.
(3) Se una limitazione inferiore alla complessità di A è $\Omega(p(n))$ e la trasformazione f richiede al più tempo $O(p(n))$, allora $\Omega(p(n))$ è anche una limitazione inferiore alla complessità del problema B.

Vediamo un paio di esempi che, sebbene non siano problemi decisionali, ma problemi di ricerca e di ottimizzazione, si prestano bene ad illustrare la tecnica di riduzione polinomiale e le sue conseguenze. [1]

Esempio (DOMINO LINEARE $\in \mathbb{P}$) Si consideri una semplificazione del DOMINO LIMITATO, nota come DOMINO LINEARE, in cui si chiede di coprire un rettangolo di altezza unitaria e larghezza uguale ad n. Questo problema può essere risolto in tempo polinomiale con un algoritmo deterministico riducendolo ad un problema di "visita" di un grafo orientato che, come abbiamo visto nel Capitolo 9, è in $\mathbb{P}$. Infatti, dati l'insieme D di m tipi di tessere, inclusa la tessera $d \in D$ da sistemare a sinistra, e l'intero n, si costruisca un grafo orientato $G = (V,E)$ il cui insieme di nodi V corrisponde all'insieme delle tessere D, ed esiste l'arco $(d_u, d_v) \in E$ se e solo se il lato destro della tessera d_u ha lo stesso colore del lato sinistro della tessera d_v. Chiaramente, il grafo G può essere costruito in $O(m^2)$ tempo, con $m = |D| = |V|$. È facile verificare che esiste una copertura "legale" per il DOMINO LINEARE se e solo se esiste un cammino in G a partire dal nodo corrispondente a d che include $n-1$ archi e/o esiste un ciclo in G i cui nodi sono raggiungibili con un cammino che parte dal nodo corrispondente a d. Poiché queste due condizioni sono verificabili in tempo polinomiale applicando una BFS, il DOMINO LINEARE è risolubile con un algoritmo polinomiale deterministico. Un esempio di riduzione è fornito nella Figura 1.6 per $m = 3$ tipi di tessere ed $n = 6$. ■

[1] Il concetto di riduzione è già stato implicitamente utilizzato nei Capitolo 11 e Capitolo 15, dove si richiedeva di formulare alcuni problemi "incogniti" (l'INSIEME DOMINANTE per grafi di intervalli e l'ABBINAMENTO per grafi bipartiti [cfr. Esercizi 11.4, 11.5, 11.6 e 15.9]), come problemi di CAMMINI MINIMI o di FLUSSO MASSIMO. In effetti, la riduzione potrebbe essere considerata come un'ulteriore tecnica di progettazione di algoritmi, che però in generale non porta ad algoritmi molto efficienti (con l'eccezione notevole dell'ABBINAMENTO per grafi bipartiti). Infatti, riducendo un problema particolare ad uno più generale, in genere si perdono alcune proprietà specifiche del problema di partenza che possono essere sfruttate per progettare algoritmi specializzati molto più efficienti.

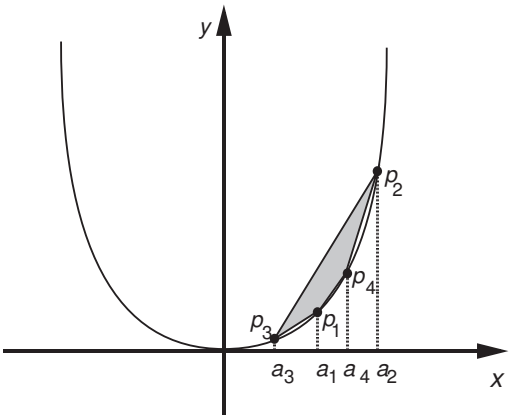

Figura 1.7: Esempio di riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO ($n = 4$).

Esempio (INVILUPPO CONVESSO è $\Omega(n \log n)$) Dimostriamo che INVILUPPO CONVESSO è $\Omega(n \log n)$ fornendo una riduzione ORDINAMENTO $\propto$ INVILUPPO CONVESSO in cui la trasformazione f ha complessità $O(n)$. Siano dati n numeri $a_1, \ldots, a_n$ per l'ORDINAMENTO, che per semplicità assumiamo interi positivi e tra loro distinti. Si costruisca un insieme di n punti $p_1, \ldots, p_n$ del piano per l'INVILUPPO CONVESSO come segue. Si consideri la parabola $y = x^2$, e si ponga $p_i = (a_i, a_i^2)$, $1 \leq i \leq n$. La soluzione dell'INVILUPPO CONVESSO è data dai vertici del medesimo che, per i dati costruiti, sono tutti gli n punti ordinati per ascisse crescenti. Pertanto, ogni algoritmo che risolva l'INVILUPPO CONVESSO deve poter implicitamente ordinare le n ascisse $a_1, \ldots, a_n$, cioè risolvere l'ORDINAMENTO. Un esempio è illustrato nella Figura 1.7 per $n = 4$. Poiché quest'ultimo problema è $\Omega(n \log n)$, come si è visto nel Capitolo 5, e la trasformazione f è $O(n)$, anche l'INVILUPPO CONVESSO è $\Omega(n \log n)$, e quindi l'algoritmo di Graham descritto nel Capitolo 16 è ottimo! ■

1.6 Teorema di Cook-Levin

Agli inizi degli anni settanta, S. Cook e L. Levin si sono chiesti se esistesse un particolare problema decisionale in $\mathbb{NP}$ tale che, se si dimostrasse la sua appartenenza a $\mathbb{P}$, allora dovrebbe risultare sicuramente $\mathbb{P} = \mathbb{NP}$. La risposta a questo quesito è affermativa: il problema esiste ed è il DOMINO LIMITATO! Il risultato fondamentale, dimostrato indipendentemente dall'uno e dall'altro, recita:

Teorema 1.1 (COOK-LEVIN) Ogni problema in $\mathbb{NP}$ si riduce in tempo polinomiale al DOMINO LIMITATO.

La dimostrazione del teorema, benché concettualmente non molto difficile, è un po' macchinosa e viene qui presentata soltanto per sommi capi. Si assume di avere un calcolatore elettronico nella cui memoria sono caricati un programma non deterministico per risolvere il problema A ed un dato d'ingresso x per A di dimensione n. Poiché A è in $\mathbb{NP}$, il calcolatore risolverà A con dato x in tempo $p(n)$, con p funzione polinomiale in n. Poiché ogni istruzione del calcolatore manipola una quantità costante d'informazione, il calcolatore utilizzerà complessivamente un numero $cp(n)$ di bit di memoria, con c costante. Si costruisce allora un insieme opportuno D di $(cp(n))^k$ tessere, con k costante, tali che la regione da "coprire" corrisponda grossomodo a (numero di bit di memoria) $\times$ (numero di istruzioni eseguite). Le tessere sono definite "astutamente" in modo che la prima riga sia ricopribile se e solo se l'input x è corretto per A, la riga i sia ricopribile dopo la riga $i-1$ se e solo se il calcolatore effettua un'operazione corretta per risolvere A, e l'ultima riga sia ricopribile se e solo se il calcolatore si arresta su **success**.

Il teorema di Cook-Levin ha come conseguenza il seguente risultato:

Corollario 1.1 $\mathbb{P} = \mathbb{NP}$ se e solo se DOMINO LIMITATO $\in \mathbb{P}$.

Dimostrazione. Infatti, se DOMINO LIMITATO $\in \mathbb{P}$ ed un problema $A \in \mathbb{NP}$, allora anche $A \in \mathbb{P}$, poiché $A \propto$ DOMINO LIMITATO. Viceversa, se DOMINO LIMITATO $\notin \mathbb{P}$, allora $\mathbb{P} \neq \mathbb{NP}$, poiché DOMINO LIMITATO $\in \mathbb{NP}$. ■

In pratica, questo corollario del teorema di Cook-Levin asserisce che il DOMINO LIMITATO è il più difficile problema in $\mathbb{NP}$. Formalmente, un problema siffatto viene detto *NP-completo*.

Un problema A è detto *NP-arduo* se $B \propto A$ per ogni problema $B \in \mathbb{NP}$. A è detto NP-completo se, oltre ad essere NP-arduo, appartiene ad $\mathbb{NP}$. Il DOMINO LIMITATO è dunque NP-completo, ma non è il solo problema NP-completo. Da quando è stato dimostrato il teorema di Cook-Levin, sono stati scoperti centinaia di altri problemi NP-completi, inclusi CRICCA, COLORAZIONE, COMMESSO VIAGGIATORE, PROGRAMMAZIONE LINEARE 0/1 e SODDISFATTIBILITÀ. Poiché la relazione $\propto$ è transitiva (cioè se $P \propto Q$ e $Q \propto R$, allora anche $P \propto R$), tutti i problemi NP-completi sono computazionalmente equivalenti: o sono tutti risolubili in tempo polinomiale con algoritmi deterministici (e allora $\mathbb{P} = \mathbb{NP}$) oppure sono tutti intrinsecamente superpolinomiali (e quindi $\mathbb{P} \neq \mathbb{NP}$). Fino al giorno d'oggi, non è noto alcun algoritmo polinomiale deterministico per un problema NP-completo, e si pensa che non lo sarà mai! L'esistenza di un tale algoritmo è legata alla possibilità di "simulare" un algoritmo non deterministico che opera in tempo polinomiale con un algoritmo deterministico che operi anch'esso in tempo polinomiale, ma allo stato attuale non si conoscono che simulazioni superpolinomiali. Abbiamo visto infatti che la complessità di un algoritmo non deterministico è proporzionale all'altezza dell'albero delle scelte, cioè al massimo livello delle foglie, e che una simulazione deterministica richiede tempo esponenziale in tale altezza. La tesi più accreditata, quindi, è che tale simulazione sia intrinsecamente superpolinomiale e che quindi la classe $\mathbb{P}$ sia propriamente contenuta nella classe $\mathbb{NP}$.

1.7 Prove di NP-completezza

Per dimostrare che un problema A è NP-completo, non è necessario mostrare ogni volta che $B \propto A$ per ogni problema $B \in \mathbb{NP}$. Infatti, per la transitività della relazione $\propto$, "basta" mostrare che:

(1) $A \in \mathbb{NP}$;
(2) esiste un problema B, già notoriamente NP-completo, tale che $B \propto A$.

In pratica, poiché $B \propto A$ implica da sola che A è almeno tanto difficile quanto B, si fa a meno di mostrare la parte (1), e si dimostra direttamente, e in modo costruttivo, la parte (2). In linea di principio, un qualsiasi problema B che sia NP-arduo può essere un candidato per provare la riduzione, ma in pratica occorre "fiuto" per scegliere un problema B la cui "somiglianza" con A agevoli la dimostrazione. Un problema che si presta abbastanza bene, per generalità e flessibilità, a provare riduzioni polinomiali è comunque il DOMINO LIMITATO. Vediamo come questo problema possa essere ridotto a CRICCA, SODDISFATTIBILITÀ e PROGRAMMAZIONE LINEARE 0/1.

Teorema 1.2 CRICCA *è NP-completa.*

Dimostrazione. Mostriamo che DOMINO LIMITATO $\propto$ CRICCA. Siano dati $n > 0$, $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$, e indichiamo il quadrato di lato n da ricoprire con $[0, n-1] \times [0, n-1]$. Definiamo un grafo non orientato $G = (V, E)$ ed un intero k come segue:

(1) Per ogni casella $(i, j) \neq (0,0)$ in $[0, n-1] \times [0, n-1]$ ed ogni $d_h \in D$, con $0 \leq h \leq m-1$, V contiene un nodo u_{hij} (l'appartenenza di u_{hij} ad una cricca "codifica" il fatto che la tessera d_h copre la casella (i, j));
(2) Per $(i, j) = (0,0)$, V contiene il nodo u_{000} (che "codifica" la copertura di $(0,0)$ con la tessera $d = d_0$);
(3) Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in E$ ad ogni nodo $u_{h'i'j'}$ tale che:

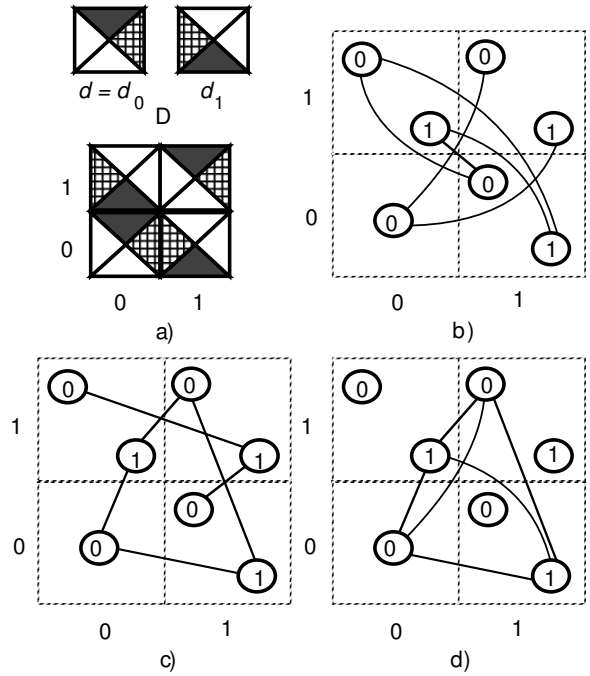

Figura 1.8: DOMINO LIMITATO $\propto$ CRICCA. (a) Insieme D di $m = 2$ tipi di tessere e copertura di un quadrato di lato $n = 2$. (b) Archi di G corrispondenti a caselle non adiacenti (condizione 3). (c) Archi di G corrispondenti a caselle adiacenti (condizione 4). (d) Una cricca di dimensione $k = n^2 = 4$ corrispondente alla copertura mostrata in (a).

(3.a) $i' = i$, $j' \neq j$ e $j' \neq j \pm 1$, oppure
(3.b) $j' = j$, $i' \neq i$ e $i' \neq i \pm 1$, oppure
(3.c) $j' \neq j$ e $i' \neq i$;
(cioè a nodi corrispondenti a caselle (i', j') distinte da (i, j) e che non sono adiacenti ad (i, j));

(4) Ciascun nodo u_{hij} è connesso con un arco $[u_{hij}, u_{h'i'j'}] \in E$ ad ogni nodo $u_{h'i'j'}$ tale che:
(4.a) d_h e $d_{h'}$ hanno i lati adiacenti dello stesso colore, ed inoltre
(4.b) $i' = i$ e $j' = j \pm 1$, oppure
(4.c) $j' = j$ e $i' = i \pm 1$,
(cioè a nodi corrispondenti a caselle (i', j') che sono adiacenti alla casella (i, j) e che possono essere coperti in modo "legale" con due tessere);

(5) $k = n^2$.

È facile verificare che la riduzione richiede tempo polinomiale in n ed m. Per costruzione, il problema DOMINO LIMITATO ha una copertura "legale" se e solo se il grafo G ha una cricca di dimensione k. Poiché CRICCA $\in \mathbb{NP}$, se n'è dimostrata l'NP-completezza. ■

Esempio (DOMINO LIMITATO $\propto$ CRICCA) La Figura 1.8 mostra un esempio di riduzione da un insieme D contenente $m = 2$ tessere per coprire un quadrato di lato $n = 2$. Per semplicità, sono disegnati separatamente gli archi corrispondenti a caselle non adiacenti (condizione (3): Figura 1.8(b) e quelli corrispondenti a caselle adiacenti (condizione (4): Figura 1.8(c). La

Figura 1.8(a) mostra una copertura legale e la Figura 1.8(d) la corrispondente cricca. ■

Teorema 1.3 SODDISFATTIBILITÀ è NP-completa.

Dimostrazione. Forniamo una riduzione dal DOMINO LIMITATO. Siano dati $n > 0$ e $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$. Reinterpretiamo le condizioni di ricoprimento del DOMINO LIMITATO in termini di una formula booleana composta da "clausole" tra loro in "**and**". Si introducono le variabili logiche u_{hij}, $0 \leq h \leq m-1$, $0 \leq i, j \leq n-1$, con il seguente significato:

$$u_{hij} = \textbf{true} \text{ se e solo se la tessera } d_h \text{ copre la casella } (i, j).$$

Si definiscono quindi le seguenti clausole:

(1) u_{000}: questa clausola "codifica" la copertura di $(0,0)$ con la tessera d;
(2) $\textbf{and}_{0 \leq i,j \leq n-1}(\textbf{or}_{0 \leq h \leq m-1}\ u_{hij})$: queste clausole asseriscono che almeno una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$;
(3) $\textbf{and}_{0 \leq i,j \leq n-1}(\textbf{and}_{0 \leq h < h' \leq m-1}\ (\textbf{not}\ u_{hij} \textbf{or not}\ u_{h'ij}))$: queste clausole asseriscono che al più una tessera copre ciascuna casella del quadrato $[0, n-1] \times [0, n-1]$;
(4) $\textbf{and}_{0 \leq i \leq n-2, 0 \leq j \leq n-1}(\textbf{and}_{h,h'}(\textbf{not} u_{hij}\ \textbf{or not}\ u_{h'i+1j}))$: per ogni coppia di tessere d_h e d'_h che non possono essere adiacenti orizzontalmente, con d_h alla sinistra di $d_{h'}$;
(5) $\textbf{and}_{0 \leq i \leq n-1, 0 \leq j \leq n-2}(\textbf{and}_{h,h'}(\textbf{not}\ u_{hij} \textbf{or not}\ u_{h'ij+1}))$: per ogni coppia di tessere d_h e $d_{h'}$ che non possono essere adiacenti verticalmente, con d_h sotto $d_{h'}$.

La formula booleana F data dall'**and** di tutte le clausole definite in (1)-(5) è in forma normale congiuntiva e contiene un numero polinomiale, in n ed m, di variabili logiche, di clausole, e di variabili (affermate o negate) per clausola. Inoltre, il problema DOMINO LIMITATO ha una copertura "legale" per i dati n e D se e solo se la formula F è soddisfattibile. Poiché la riduzione richiede tempo polinomiale in n ed m e SODDISFATTIBILITÀ $\in \mathbb{NP}$, si è dimostrato che tale problema è NP-completo. ■

Esempio (DOMINO LIMITATO $\propto$ SODDISFATTIBILITÀ) Consideriamo un altro esempio di riduzione a partire dagli stessi dati di input del DOMINO LIMITATO mostrati nella Figura 1.8(a). Le clausole definite in (1)-(5) sono le seguenti:

(1) u_{000}
(2) (u_{000} **or** u_{100}) **and** (u_{001} **or** u_{101}) **and** (u_{010} **or** u_{110}) **and** (u_{011} **or** u_{111})
(3) (**not** u_{000} **or not** u_{100}) **and** (**not** u_{001} **or not** u_{101}) **and**
(**not** u_{010} **or not** u_{110}) **and** (**not** u_{011} **or not** u_{111})
(4) (**not** u_{000} **or not** u_{010}) **and** (**not** u_{001} **or not** u_{011}) **and** (**not** u_{100} **or not** u_{010}) **and**
(**not** u_{101} **or not** u_{011}) **and** (**not** u_{100} **or not** u_{110}) **and** (**not** u_{101} **or not** u_{111})
(5) (**not** u_{000} **or not** u_{001}) **and** (**not** u_{010} **or not** u_{011}) **and**
(**not** u_{100} **or not** u_{101}) **and** (**not** u_{110} **or not** u_{111})

Mettendo tutti gli spezzoni in **and** si ottiene la formula F che è soddisfatta per l'assegnamento dei valori di verità

$$u_{000} = u_{011} = u_{101} = u_{110} = \textbf{true},\ u_{001} = u_{010} = u_{100} = u_{111} = \textbf{false},$$

che corrisponde alla copertura del DOMINO LIMITATO mostrata nella Figura 1.8(a). ■

Teorema 1.4 PROGRAMMAZIONE LINEARE 0/1 è NP-completa.

Dimostrazione. La dimostrazione è analoga a quella del Teorema 1.3. Siano dati $n > 0$ e $D = \{d_0, d_1, \ldots, d_{m-1}\}$, con $d = d_0$, per il DOMINO LIMITATO. Reinterpretiamo le condizioni di

ricoprimento del DOMINO LIMITATO in termini di un sistema di disequazioni, introducendo le variabili intere 0/1 x_{hij}, $0 \leq h \leq m-1$, $0 \leq i,j \leq n-1$, con il seguente significato:

$$x_{hij} = 1 \text{ se e solo se la tessera } d_h \text{ copre la casella } (i,j).$$

Si definiscono le seguenti disequazioni:

(1) $x_{000} = 1$: "codifica" la copertura di $(0,0)$ con la tessera d;

(2) $\sum_{0 \leq h \leq m-1} x_{hij} = 1, 0 \leq i,j \leq n-1$: queste equazioni asseriscono che esattamente una tessera copre ciascuna casella di $[0,n-1] \times [0,n-1]$;

(3) $x_{hij} + \sum_{h'} x_{h'i+1j} \leq 1$, $0 \leq h \leq m-1$, $0 \leq i \leq n-2$, $0 \leq j \leq n-1$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti orizzontalmente a d_h, con d_h alla sinistra di $d_{h'}$;

(4) $x_{hij} + \sum_{h'} x_{h'ij} + 1 \leq 1$, $0 \leq h \leq m-1$, $0 \leq i \leq n-1$, $0 \leq j \leq n-2$, per tutte le tessere $d_{h'}$ che non possono essere adiacenti verticalmente a d_h, con d_h sotto $d_{h'}$.

Poiché una equazione $a = b$ può essere espressa come coppia di disequazioni $a \leq b$ e $a \geq b$, il sistema definito in (1)-(4) è un sistema di disequazioni. Tale sistema contiene un numero polinomiale, in n ed m, di variabili 0/1, di disequazioni, e di variabili per disequazione. È facile verificare che il DOMINO LIMITATO ha una copertura "legale" per n e D se e solo se il sistema di disequazioni ha soluzione. La riduzione richiede tempo polinomiale in n ed m. Essendo PROGRAMMAZIONE LINEARE 0/1 $\in \mathbb{NP}$, si è provato che questo problema è NP-completo. ■

Esempio **(DOMINO LIMITATO $\propto$ PROGRAMMAZIONE LINEARE 0/1)** Riconsiderando i dati di input del DOMINO LIMITATO mostrati nella Figura 1.8(a), si ottiene il sistema

$$\begin{array}{lll}
x_{000} = 1 \quad (1) & x_{000} + x_{010} \leq 1 \quad (3) & x_{101} + x_{111} \leq 1 \quad (3) \\
x_{000} + x_{100} = 1 \quad (2) & x_{001} + x_{011} \leq 1 \quad (3) & x_{000} + x_{001} \leq 1 \quad (4) \\
x_{001} + x_{101} = 1 \quad (2) & x_{100} + x_{010} \leq 1 \quad (3) & x_{010} + x_{011} \leq 1 \quad (4) \\
x_{010} + x_{110} = 1 \quad (2) & x_{101} + x_{011} \leq 1 \quad (3) & x_{100} + x_{101} \leq 1 \quad (4) \\
x_{011} + x_{111} = 1 \quad (2) & x_{100} + x_{110} \leq 1 \quad (3) & x_{110} + x_{111} \leq 1 \quad (4)
\end{array}$$

che è verificato per

$$x_{000} = x_{011} = x_{101} = x_{110} = 1, x_{001} = x_{010} = x_{100} = x_{111} = 0,$$

che corrisponde ancora alla copertura mostrata nella Figura 1.8(a). Il numero fra parentesi indica la regola da cui è stata generata la disequazione. ■

Vediamo un breve elenco di nuovi problemi che, al pari dei precedenti, sono tutti NP-completi.

COPERTURA ESATTA DI INSIEMI (EXACT COVER). *Dato un insieme X e una famiglia di suoi sottoinsiemi $Y = \{Y_1, \ldots, Y_n\}$, esiste una sottofamiglia F di Y che partizioni X?*

ABBINAMENTO 3-DIMENSIONALE (3-DIMENSIONAL MATCHING). *Dato un insieme M, sottoinsieme di $W \times X \times V$, con W, X, V insiemi disgiunti di eguale cardinalità q, esiste un sottoinsieme M' di M, con $|M'| = q$, tale che se (w,x,v) e (w',x',v') sono elementi distinti di M', allora $w \neq w'$, $x \neq x'$ e $v \neq v'$?*

PARTIZIONE (PARTITION). *Dato un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi, esiste un sottoinsieme S di {1, ..., n} tale che $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?*

SOMMA DI SOTTOINSIEME (SUBSET SUM). *Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = k$?*

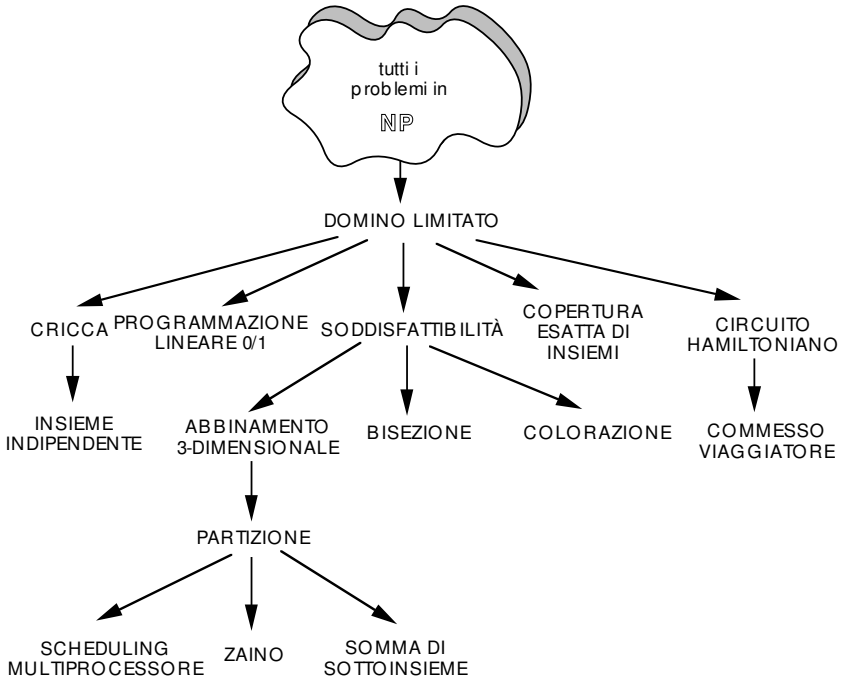

Figura 1.9: Riduzioni tra problemi in $\mathbb{NP}$.

ZAINO (KNAPSACK). *Dati un intero positivo C (la capacità dello zaino) e un insieme di n "oggetti", tali che l'oggetto i-esimo è caratterizzato da un "profitto" p_i e da un "volume" v_i, entrambi interi positivi, esiste un sottoinsieme S di $\{1,\ldots,n\}$ tale che il volume totale $v(S) = \sum_{i\in S} v_i \leq C$ e il profitto totale $p(S) = \sum_{i\in S} p_i$ è maggiore o uguale a k?*

SCHEDULING MULTIPROCESSORE (MULTIPROCESSOR SCHEDULING). *Dati un insieme $P = \{p_1,\ldots,p_n\}$ di "programmi", tali che p_i richiede un "tempo (intero) di esecuzione" t_i, m "processori" identici, ed un intero d, è possibile eseguire tutti i programmi sui processori in al più d unità di tempo?*

CIRCUITO HAMILTONIANO (HAMILTONIAN CIRCUIT). *Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?*

INSIEME INDIPENDENTE (INDEPENDENT SET). *Dati un grafo non orientato G ed un intero k, esiste un sottoinsieme di almeno k nodi mutuamente non connessi da archi?*

BISEZIONE (BISECTION). *Dati un grafo non orientato $G = (V,E)$ ed un intero k, esiste una partizione di V in due sottoinsiemi di $|V|/2$ nodi tale che il numero di archi con un estremo in un sottoinsieme ed un estremo nell'altro non supera k?*

Tra questi problemi, si possono dimostrare tutte le riduzioni illustrate nella Figura 1.9, dove una riduzione $A \propto B$ è mostrata con una freccia da A verso B. Le riduzioni

- DOMINO LIMITATO $\propto$ CRICCA
- DOMINO LIMITATO $\propto$ SODDISFATTIBILITÀ
- DOMINO LIMITATO $\propto$ PROGRAMMAZIONE LINEARE 0/1

sono state appena dimostrate. Vediamo di provarne altre molto più semplici.

Teorema 1.5 CRICCA $\propto$ INSIEME INDIPENDENTE.

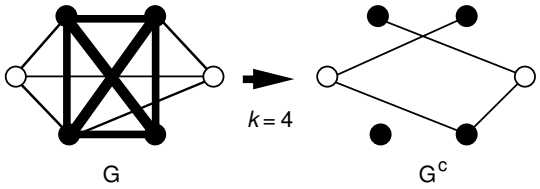

Figura 1.10: Riduzione CRICCA $\propto$ INSIEME INDIPENDENTE. I nodi neri formano una cricca di G ed un insieme indipendente di G^c di cardinalità maggiore o uguale a $k = 4$.

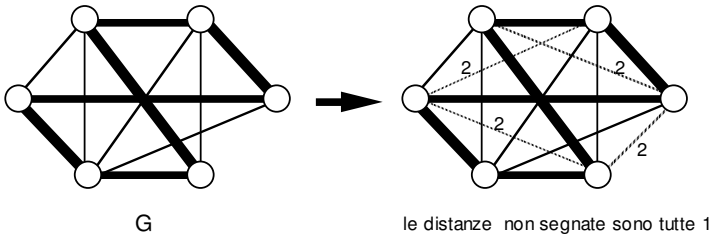

Figura 1.11: Riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE. Le distanze tra le città sono tutte 1, tranne quelle tratteggiate, che valgono 2 perché non corrispondono ad archi di G. Un circuito Hamiltoniano di G ed il percorso corrispondente per il commesso viaggiatore di lunghezza $k = n = 6$ sono mostrati in grassetto.

Dimostrazione. Dato un grafo non orientato $G = (V,E)$ per il problema CRICCA, si definisca il grafo "complemento" $G^c = (V,E^c)$ di G avente gli stessi nodi, ma in cui sono scambiati archi e "non archi": $[u,v] \in E^c$ se e solo se $[u,v] \notin E$. È immediato verificare che un sottoinsieme S di nodi è una soluzione di CRICCA per il grafo G se e solo è anche una soluzione di INSIEME INDIPENDENTE per G^c. Un esempio è fornito nella Figura 1.10. ■

Teorema 1.6 CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE.

Dimostrazione. Dato un grafo non orientato $G = (V,E)$, si definisca un insieme di dati per il COMMESSO VIAGGIATORE come segue. Si introducano n città, con $n = |V|$, si ponga $k = n$, e si definisca così la matrice $D = [d_{ij}]$ delle distanze tra le città i e j, per $1 \leq i, j \leq n$:

$$d_{ij} = \begin{cases} 1, & \text{se } [i,j] \in E, \\ 2, & \text{se } [i,j] \notin E. \end{cases}$$

È immediato verificare che esiste un circuito Hamiltoniano in G se e solo se esiste un percorso del COMMESSO VIAGGIATORE con distanza complessiva k. Un esempio è fornito nella Figura 1.11. ■

Teorema 1.7 PARTIZIONE $\propto$ SCHEDULING MULTIPROCESSORE.

Dimostrazione. Dato un insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si definisca un insieme $P = \{p_1, \ldots, p_n\}$ di "programmi", tali che p_i richiede tempo $t_i = a_i$, per $1 \leq i \leq n$. Si considerino $m = 2$

"processori" e si ponga $d = \frac{1}{2}\sum_{1\leq i\leq n} a_i$. È facile vedere che è possibile eseguire tutti i programmi sui due processori in al più d unità di tempo se e solo se esiste una soluzione per PARTIZIONE. ■

Teorema 1.8 PARTIZIONE $\propto$ SOMMA DI SOTTOINSIEME.

Dimostrazione. Dato l'insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si ponga $k = \frac{1}{2}\sum_{1\leq i\leq n} a_i$. ■

Teorema 1.9 PARTIZIONE $\propto$ ZAINO.

Dimostrazione. Dato un insieme $A = \{a_1, \ldots, a_n\}$ per PARTIZIONE, si pongano $p_i = v_i = a_i$, $1 \leq i \leq n$, e $k = c = \frac{1}{2}\sum_{1\leq i\leq n} a_i$. ■

Lo scopo delle precedenti riduzioni è mostrare come molti problemi, che nascono da contesti diversi, siano in realtà riformulazioni dello stesso problema o sue semplici varianti. In questo caso, è facile dimostrare l'NP-completezza di un problema fornendo una riduzione polinomiale da uno "che gli somigli il più possibile". La semplicità di queste riduzioni non deve però trarre in inganno, poiché molte riduzioni sono, al contrario, piuttosto complicate come, per esempio, SODDISFATTIBILITÀ $\propto$ ABBINAMENTO 3-DIMENSIONALE e DOMINO LIMITATO $\propto$ CIRCUITO HAMILTONIANO, che qui non stiamo a dimostrare.

1.8 Reality check

Il Clay Mathematics Institute (CMI) è una fondazione non-profit dedicata alla diffusione della conoscenza matematica. Nel 2000, il CMI ha istituito un premio da un milione di dollari per chiunque risolva uno qualsiasi dei sette "Millenium problems", una collezione di problemi matematici "classici" ancora aperti. Fra i sette problemi c'è la questione $\mathbb{P} = \mathbb{NP}$, ovvero la domanda se le due classi di complessità coincidono oppure no. Un motivo in più per fare ricerca in questo campo! Dei sette problemi originali, uno è stato risolto: la Congettura di Poincaré è stata dimostrata da Grigori Perelman nel 2003. Nel marzo 2010, il Clay Institute ha assegnato a Perelman il premio da un milione di dollari, ma lui lo ha rifiutato. In precedenza, aveva rifiutato la medaglia Fields, il più importante premio nel campo della matematica.

Il premio Turing è invece il più prestigioso riconoscimento nel campo dell'informatica ed annovera tra i vincitori gli "algoritmisti" Edsger Dijkstra, Donald Knuth, Michael Rabin, Robert Floyd, Antony Hoare, Richard Karp, John Hopcroft, Robert Tarjan, Ronald Rivest, Adi Shamir, Leonard Adleman, nonché i "teorici della complessità" Stephen Cook, Juris Hartmanis, Richard Stearns e Andrew Yao. Nel 2012, il premio Turing è stato assegnato a Silvio Micali e Shafi Goldwasser, per i loro lavori nel campo della crittografia. Micali ha nazionalità italiana e si è laureato all'Università La Sapienza di Roma nel 1978.

1.9 Esercizi

Esercizio 1.1 (Tangentopoli)**.** Dati un insieme M di n "mazzette" ed un insieme P di n "politici corrotti" appartenenti al Partito dei Soldi (PS) oppure al Partito della Felicità (PDF), si vogliono assegnare tutte le mazzette a tutti i politici in modo che il PS riceva complessivamente il doppio del PDF. Si scriva una procedura non deterministica di complessità polinomiale.

Esercizio 1.2 (Zaino)**.** Si scriva una procedura non deterministica di complessità $O(n)$ per la versione decisionale del problema dello ZAINO.

Esercizio 1.3 (Colorazione)**.** Si scriva una procedura non deterministica di complessità polinomiale che risolva il problema della COLORAZIONE.

Esercizio 1.4 (Tangentopoli)**.** Si scriva una procedura deterministica di enumerazione per il problema dell'Esercizio 1.1.

Esercizio 1.5 (Zaino)**.** Si scriva una procedura deterministica di enumerazione per il problema dello ZAINO.

Esercizio 1.6 (Colorazione)**.** Si scriva una procedura deterministica di enumerazione per il problema della COLORAZIONE.

Esercizio 1.7 (3-Copertura di insiemi)**.** Si scriva una procedura deterministica di enumerazione per il problema della 3-COPERTURA DI INSIEMI: "Dati un insieme $X = \{1,2,\ldots,m\}$ ed una famiglia $Y = \{Y_1,\ldots,Y_n\}$ di n sottoinsiemi di X, ciascuno avente cardinalità 3, esiste una sottofamiglia S di insiemi di Y che partiziona X?"

Esercizio 1.8 (Dimensione)**.** Si stabilisca se le complessità delle funzioni Pippo() e Follia() dell'Esercizio 2.12 siano polinomiali nella dimensione dell'input oppure no.

Esercizio 1.9 (Counting Sort)**.** Si stabilisca se la complessità della procedura countingSort() sia polinomiale nella dimensione dell'input oppure no.

Esercizio 1.10 (Coefficiente binomiale)**.** Si stabilisca se la complessità della procedura tartaglia() definita nel Capitolo 13 sia polinomiale nella dimensione dell'input oppure no.

Esercizio 1.11 (Copertura con nodi)**.** Dati un grafo non orientato G ed un intero k, il problema della COPERTURA CON NODI chiede se esiste un sottoinsieme S di al più k nodi tale che ciascun arco abbia almeno un estremo in S. Si dimostri che INSIEME INDIPENDENTE $\propto$ COPERTURA CON NODI.

Esercizio 1.12 (Ciclo Hamiltoniano)**.** Dato un grafo orientato G, il problema del CICLO HAMILTONIANO chiede se esiste un ciclo che include ogni nodo di G esattamente una volta. Si dimostri che CIRCUITO HAMILTONIANO $\propto$ CICLO HAMILTONIANO.

Esercizio 1.13 (Cammino Hamiltoniano)**.** Dato un grafo orientato G, il problema del CAMMINO HAMILTONIANO chiede se esiste un cammino che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CAMMINO HAMILTONIANO. Si mostri che la riduzione vale anche se sono dati il nodo di partenza r ed il nodo di arrivo t per il CAMMINO HAMILTONIANO.

Esercizio 1.14 (Catena Hamiltoniana)**.** Dato un grafo non orientato G, il problema della CATENA HAMILTONIANA chiede se esiste una catena che include ogni nodo di G esattamente una volta. Si dimostri che CICLO HAMILTONIANO $\propto$ CATENA HAMILTONIANA (Suggerimento: si modifichi la riduzione dell'Esercizio 1.13, sostituendo ogni nodo del grafo orientato D di partenza con una catena di tre nodi del grafo non orientato G di arrivo, e sistemando opportunamente in G le connessioni relative agli archi di D).

Esercizio 1.15 (Cammino massimo)**.** Dati un grafo orientato G, due nodi r e t, ed un intero k, il problema del CAMMINO MASSIMO chiede se esiste un cammino semplice da r a t che include almeno k archi. Si dimostri che il problema è NP-completo. Si dimostri inoltre che il problema diventa polinomiale qualora G sia aciclico.

Esercizio 1.16 (Albero di copertura limitato)**.** Dati un grafo non orientato G ed un intero k, il problema dell'ALBERO DI COPERTURA LIMITATO chiede se esiste un albero di copertura T per G tale che in ogni nodo di T siano incidenti al più k archi. Si dimostri che il problema è NP-completo.

1.10 Soluzioni

Soluzione Esercizio 1.1

Si rappresentino le mazzette con un vettore M di n interi ed i politici con un vettore P di n booleani tale che $P[i] =$ **true** se il politico i è del PS e $P[i] =$ **false** se è del PDF. Si ottiene la seguente procedura di complessità $O(n)$. Si noti il modo con cui nel primo **for** si generano in tempo $O(n)$ tutte le permutazioni del vettore M nel vettore S.

```
mazzette(int[]M, int[]S, int[]P)
for i = 1 to n do
    j = choice({i, i+1, ..., n})
    S[i] = M[j]
    M[j] = M[i]
int M_PS = 0
int M_PDF = 0
for i = 1 to n do
    if P[i] then M_PS = M_PS + S[i]
            else M_PDF = M_PDF + S[i]
if M_PS = 2 · M_PDF then success
else failure
```

Soluzione Esercizio 1.2

La procedura ndZaino() risolve il problema, prendendo in input le variabili V (vettore dei volumi), P (vettore dei profitti), n (dimensione del problema), C capacità dello zaino e k (profitto che si vuole raggiungere).

```
ndZaino(int[] V, int[] P, int n, int C, int k)
SET S = Set()
for i = 1 to n do
    if choice ({true, false }) then S.insert(i)
int T_p = 0
int T_v = 0
foreach i ∈ S do
    T_p = T_p + P[i]
    T_v = T_v + V[i]
if T_p ≥ k and T_v ≤ C then success
else failure
```

Soluzione Esercizio 1.5

La procedura enZaino() risolve il problema. Per semplicità, si assume che le variabili V, P, n, C e k (le stesse dell'Esercizio 1.2) siano globali. Le variabili T_v e T_p rappresentano il volume e il profitto raggiunti finora, mentre S è l'insieme di oggetti scelti. La chiamata è enZaino$(S,0,0,1)$, con S vuoto.

Soluzione Esercizio 1.7

La procedura enCopertura() risolve il problema. Si assuma che la famiglia Y sia rappresentata con una matrice $n \times 3$, e che la sottofamiglia S sia rappresentata con un vettore booleano. Per verificare che S partizioni X, si utilizza un vettore booleano $B[1 \ldots m]$. Se Y_j è stato selezionato ($S[j] =$ **true**),

allora $B[Y[j,k]]$ è posto a **true** per $k = 1,2,3$. Se però $B[Y[j,k]]$ era già **true**, allora non si ha una partizione. Infine, si verifica che non sia rimasto alcun elemento di B a **false**, poiché anche in tal caso non si ha una partizione. La chiamata è enCopertura(i) e la complessità è $O((n+m)2^n)$. Si può migliorare l'algoritmo "potando" tutti quei casi in cui due sottoinsiemi sono intersecanti, prima di arrivare a $i = n$.

Soluzione Esercizio 1.8

Le funzioni Pippo() e Follia() ricevono in ingresso un intero n. Poiché per codificare n in modo conciso occorrono $O(\log n)$ caratteri, la dimensione d dell'input è $O(\log n)$. La funzione Pippo() richiede $O(\log n) = O(d)$ tempo, e quindi ha complessità polinomiale nella dimensione dell'input. La funzione Follia() invece richiede $O(n) = O(2^d)$ tempo, e quindi ha complessità esponenziale nella dimensione dell'input.

Soluzione Esercizio 1.9

La complessità di Counting Sort è $O(n+k)$, dove n è il numero di elementi e k è il valore dell'elemento maggiore. Poiché k è espresso con $d = \log k$ bit, la complessità è $O(n+2^d)$, superpolinomiale in d.

```
boolean enZaino(SET S, int T_p, int T_v, int i)
    if i ≤ n then
        if enZaino(S, T_p, T_v, i+1) then
            return true                                  % Elemento i non inserito
        S.insert(i)
        if T_v + V[i] ≤ C and T_p + P[i] ≥ k then
            return true
        if enZaino(S, T_p + P[i], T_v + V[i], i+1) then
            return true                                  % Elemento i inserito
        S.remove(i)
    return false
```

```
boolean enCopertura(int[][] Y, int n, int m, boolean[] S, int i)
    if i ≤ n then
        S[i] = false
        if enCopertura(Y, n, m, S, i+1) then return true
        S[i] = true
        if enCopertura(Y, n, m, S, i+1) then return true
    else
        for j = 1 to m do B[j] = false
        for j = 1 to n do
            if S[j] then
                for k = 1 to 3 do
                    if B[Y[j,k]] then return false       else B[Y[j,k]] = true
        for j = 1 to m do if not B[j] then return false

        return true
```