Capitolo 9

NP-completezza

SOMMARIO

In questo capitolo finale riprendiamo in esame le classi di complessità introdotte nel primo capitolo, dandone una definizione formale basata sul concetto di problema decisionale e su quello di verifica di una soluzione. Introduciamo inoltre il concetto di riduzione polinomiale e quello di problema NP-completo, dimostrando la NP*-completezza di alcuni problemi computazionali esaminati nel resto del libro e fornendo alcuni suggerimenti su come dimostrare un nuovo risultato di* NP*-completezza. Infine, mostriamo come affrontare la difficoltà computazionale intrinseca di un problema facendo uso di algoritmi polinomiali di approssimazione: forniamo un tale algoritmo per il problema del ricoprimento tramite vertici e per quello del commesso viaggiatore ristretto a istanze metriche e mostriamo come, in generale, quest'ultimo problema non ammetta algoritmi di approssimazione ma possa essere risolto, in pratica, mediante euristiche basate sul paradigma della ricerca locale.*

DIFFICOLTÀ

1,5 CFU

9.1 Problemi NP-completi

Abbiamo più volte incontrato nei capitoli precedenti problemi per i quali non conosciamo algoritmi di risoluzione polinomiali, ma per i quali non siamo neanche in grado di dimostrare che tali algoritmi non esistano. In particolare, nel primo capitolo abbiamo introdotto il problema del gioco del Sudoku, per poi considerare nel capitolo successivo i problemi della partizione di numeri interi e della bisaccia e, nel quinto capitolo, i problemi della colorazione di grafi e della ricerca del massimo insieme indipendente. In tutti questi casi, abbiamo detto che i problemi erano NP-completi, intendendo con questo termine problemi per i quali, nonostante gli sforzi ripetuti di molti ricercatori, non

conosciamo algoritmi di risoluzione polinomiale e tali che se uno solo di essi è risolvibile in tempo polinomiale, allora ogni problema NP-completo è risolvibile in tempo polinomiale. Per questo motivo, è opinione diffusa che un problema NP-completo non può essere risolto in tempo polinomiale, anche se non conosciamo ancora una dimostrazione di tale affermazione.

Il concetto di problema NP-completo consente di affrontare la risoluzione di un problema computazionale da due punti di vista complementari. Da un lato, il nostro obiettivo è chiaramente quello di progettare un algoritmo di risoluzione efficiente per il problema in esame: le strutture di dati e i paradigmi algoritmici discussi in questo libro sono gli strumenti più adatti per tentare di raggiungere quest'obiettivo. Dall'altro lato, avendo a disposizione la nozione di problema NP-completo, possiamo tentare di dimostrare che quello che stiamo analizzando è, probabilmente, *intrinsecamente difficile* e che la progettazione di un algoritmo polinomiale per esso sarebbe un risultato con conseguenze molto più significative (ovvero, migliaia di altri problemi risulterebbero improvvisamente risolvibili in tempo polinomiale).

Sebbene mostrare l'intrinseca difficoltà di un problema computazionale possa sembrare meno interessante che progettare per esso un algoritmo polinomiale di risoluzione, un risultato di NP-completezza ha l'indubbio vantaggio di far rivolgere i nostri sforzi verso obiettivi meno ambiziosi: come vedremo al termine di questo capitolo, esistono diversi modi per affrontare la difficoltà di un problema NP-completo, tra cui uno dei più esplorati consiste nella progettazione di algoritmi di approssimazione, ovvero algoritmi efficienti le cui prestazioni, in termini di qualità della soluzione calcolata (non necessariamente ottima), siano in qualche modo garantite.

9.1.1 Classi P e NP

Nel primo capitolo abbiamo introdotto in modo informale le classi di complessità P e NP, evidenziando che un problema contenuto nella prima classe ammette un algoritmo di risoluzione efficiente, ovvero polinomiale, mentre un problema contenuto nella seconda classe ammette un algoritmo efficiente di verifica di una soluzione.[1]

Per formalizzare queste definizioni, restringiamo la nostra attenzione (per ora) ai soli problemi di decisione, ossia ai problemi la cui soluzione è una risposta binaria — sì o no. Utilizzando i meccanismi di codifica binaria (Paragrafo 1.2.1) delle istanze di un problema decisionale, identifichiamo un problema decisionale con il corrispettivo insieme (potenzialmente infinito) di istanze la cui risposta è "sì": risolvere tale problema consiste quindi nel decidere l'appartenenza di una sequenza binaria all'insieme (osserviamo che non è restrittivo restringersi alle sole sequenze binarie, in quanto il calcolatore stesso ope-

[1] L'acronimo NP sta per *non-deterministico polinomiale*, motivato storicamente dalla definizione della classe NP usando la macchina di Turing non-deterministica.

ra solo su tale tipo di sequenze). Pertanto, un **problema di decisione** Π non è altro che un sottoinsieme dell'insieme di tutte le possibili sequenze binarie, in particolare di quelle che soddisfano una determinata proprietà.

Ad esempio, il problema di decidere se un dato numero intero è primo consiste di tutte le sequenze binarie che codificano numeri interi primi, per cui la sequenza 1101 (13 in decimale) appartiene a tale problema mentre la stringa 1100 (12 in decimale) non vi appartiene. Il problema di decidere se un grafo può essere colorato con tre colori consiste di tutte le sequenze binarie che codificano grafi che possono essere colorati con tre colori.

Nel seguito, per motivi di chiarezza, continueremo a definire un problema decisionale facendo uso di descrizioni formulate in linguaggio naturale, anche se implicitamente intenderemo definirlo come uno specifico insieme di sequenze binarie, facendo riferimento a opportuni meccanismi di codifica (Paragrafo 1.2.1).

Un problema decisionale Π appartiene alla **classe** P se esiste un algoritmo polinomiale A che, presa in ingresso una sequenza binaria x, determina se x appartiene a Π o meno. Ad esempio, sappiamo che il problema di decidere se, dato un grafo (orientato) G e due suoi nodi s e t, esiste un cammino semplice da s a t appartiene a P, in quanto abbiamo visto nel Paragrafo 7.4.1 come visitare tutti i nodi raggiungibili da s operando una visita in ampiezza del grafo: se t è incluso tra questi vertici, allora la risposta al problema è affermativa, altrimenti è negativa.

Non sappiamo se il problema di decidere se un grafo può essere colorato con tre colori appartiene a P, ma possiamo mostrare che lo stesso problema ristretto a due colori vi appartiene: a tale scopo, mostriamo come utilizzare il fatto che se un vertice è colorato con il primo colore, allora tutti i suoi vicini devono essere colorati con il secondo colore. L'idea dell'algoritmo consiste nel colorare un vertice i con uno dei due colori e dedurre tutte le colorazioni degli altri vertici che ne conseguono: se riusciamo a colorare tutti i vertici, possiamo concludere che il grafo è colorabile con due colori. Altrimenti, se arriviamo a una contraddizione (ovvero, siamo costretti a colorare un vertice con due colori diversi), possiamo dedurre che il grafo non è colorabile con due colori (in realtà, questo algoritmo deve essere applicato a ogni componente connessa del grafo).

Il Codice 9.1 realizza l'algoritmo appena descritto ipotizzando che il grafo in ingresso sia connesso: dopo aver cancellato i colori di tutti i vertici (righe 2 e 3), il codice prova ad assegnare al primo vertice il colore 1 e aggiorna il numero di vertici colorati, memorizzato nella variabile `fatti` (riga 4). Il ciclo `while` successivo determina le eventuali conseguenze della colorazione attuale: ogni vertice i viene esaminato all'interno del ciclo `for` alle righe 6–21, per vedere se non è già stato colorato (riga 7). In tal caso, il codice controlla se i vicini di i hanno usato entrambi i colori (righe 8–13): se è così, allora abbiamo trovato una contraddizione (riga 15). Se invece i vicini di i hanno usato uno solo dei due colori, l'altro viene assegnato a i stesso (righe 17 e 18): ovviamente, vi

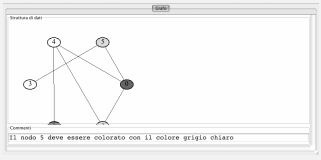
```
1  DueColorazione( A ): ⟨pre: A matrice di adiacenza di un grafo connesso di n nodi⟩
2    FOR (i = 0; i < n; i = i+1)
3      colore[i] = -1;
4    colore[0] = fatti = 1;
5    WHILE (fatti < n)
6      FOR (i = 0; i < n; i = i+1) {
7        IF (colore[i] < 0) {
8          usato[0] = usato[1] = FALSE;
9          FOR (j = 0; j < i; j = j + 1) {
10           IF (A[j][i] == 1 && colore[j] >= 0) {
11             usato[colore[j]] = TRUE;
12           }
13         }
14         IF (usato[0] && usato[1]) {
15           RETURN FALSE;
16         } ELSE {
17           IF (usato[0]) { colore[i] = 1; fatti = fatti + 1; }
18           IF (usato[1]) { colore[i] = 0; fatti = fatti + 1; }
19         }
20       }
21     }
22   RETURN TRUE;
```

Codice 9.1 Algoritmo per decidere se un grafo connesso è colorabile con due colori.

è anche la possibilità che nessun colore sia stato usato dai vicini di i, nel qual caso non possiamo ancora concludere nulla sulla sua colorazione. Al termine del ciclo `while`, se non troviamo una contraddizione, concludiamo che il grafo è colorabile con due colori (in quanto tutti i nodi sono stati colorati): altrimenti deduciamo che non lo è (riga 22).

ALVIE: colorazione di un grafo con due colori

Osserva, sperimenta e verifica
`TwoColoring`



Poiché a ogni iterazione del ciclo `while` coloriamo un nuovo nodo oppure incontriamo una contraddizione, abbiamo che il numero di iterazioni eseguite dal ciclo è al più n. Ciascuna iterazione richiede $O(n^2)$ tempo, per cui la complessità temporale dell'algoritmo è $O(n^3)$. Facendo riferimento alla rappresentazione mediante liste di adia-

cenza e utilizzando una variante della procedura di visita in ampiezza di un grafo vista nel Paragrafo 7.4.1, possiamo mostrare che tale algoritmo può essere implementato più efficientemente in tempo $O(n + m)$, dove m indica il numero degli archi del grafo.

Migliaia di problemi decisionali appartengono alla classe P e in questo libro ne abbiamo incontrati diversi. Da questo punto di vista, uno dei risultati recenti più interessanti è stato ottenuto dagli indiani Manindra Agrawal, Neeraj Kayal e Nitin Saxena e consiste nella dimostrazione che appartiene a P il problema di decidere se un dato numero intero è primo: tale problema aveva resistito all'attacco di centinaia di valenti ricercatori matematici e informatici, senza che nessuno fosse stato in grado di progettare un algoritmo di risoluzione polinomiale o di dimostrare che un tale algoritmo non poteva esistere.

La classe P, tuttavia, non esaurisce l'intera gamma dei problemi decisionali: come abbiamo visto nel primo capitolo, esistono molti problemi per i quali non conosciamo un algoritmo di risoluzione polinomiale e ve ne sono diversi per i quali siamo sicuri che tale algoritmo non esiste (come il problema delle torri di Hanoi).

Introduciamo ora la classe NP che include, oltre a tutti i problemi in P, molti altri problemi computazionali. Intuitivamente, un problema Π in NP non necessariamente ammette un algoritmo di risoluzione polinomiale, ma è tale che se una sequenza x appartiene a Π allora deve esistere una *dimostrazione* breve di questo fatto, la quale può essere *verificata* in tempo polinomiale. Formalmente, la **classe NP** include tutti i problemi di decisione Π per i quali esiste un algoritmo polinomiale V e un polinomio p che, per ogni sequenza binaria x, soddisfano le seguenti due condizioni:

completezza: se x appartiene a Π, allora esiste una sequenza y di lunghezza $p(|x|)$ tale che V con x e y in ingresso termina restituendo il valore TRUE;

consistenza: se x non appartiene a Π, allora, per ogni sequenza y, V con x e y in ingresso termina restituendo il valore FALSE.

Osserviamo che P è contenuta in NP. Dato un problema Π in P, sia A un algoritmo di risoluzione polinomiale per Π. Possiamo allora definire V nel modo seguente: per ogni x e y, l'algoritmo V con x e y in ingresso restituisce il valore TRUE se A con x in ingresso risponde in modo affermativo, altrimenti restituisce il valore FALSE. Chiaramente, V (assieme a un qualunque polinomio e senza aver bisogno di usare y) soddisfa le condizioni di completezza e consistenza sopra descritte: quindi, Π appartiene a NP.

Non sappiamo, invece, se NP è contenuta in P. Data l'enorme quantità di problemi in NP per i quali non conosciamo un algoritmo polinomiale di risoluzione, la congettura più accreditata è che P sia diversa da NP. Non avendo una dimostrazione di quest'affermazione, possiamo solamente individuare all'interno della classe NP i problemi decisionali che maggiormente si prestano a fungere da problemi "separatori" delle due classi: tali problemi sono i problemi NP-completi, che costituiscono i problemi più difficili all'interno della classe NP.

9.1.2 Riducibilità polinomiale

Per poter definire il concetto di problema NP-completo, abbiamo bisogno della nozione di riducibilità polinomiale. Intuitivamente, un problema di decisione Π è riducibile polinomialmente a un altro problema di decisione Π' se l'esistenza di un algoritmo di risoluzione polinomiale per Π' implica l'esistenza di un tale algoritmo anche per Π. Abbiamo già visto come il concetto di riducibilità può essere applicato per dimostrare che un dato problema computazionale è risolvibile in modo efficiente (pensiamo ad esempio al problema del minimo antenato comune analizzato nel Paragrafo 4.2.1). Forniamo ora un ulteriore esempio di tale applicazione considerando il problema della soddisfacibilità ristretto a clausole con due letterali.

Sia $X = \{x_0, x_1, \ldots, x_{n-1}\}$ un insieme di n variabili booleane. Una *formula booleana in forma normale congiuntiva* su X è un insieme $C = \{c_0, c_1, \ldots, c_{m-1}\}$ di m *clausole*, dove ciascuna clausola c_i, per $0 \leqslant i < m$, è a sua volta un insieme di *letterali*, ovvero un insieme di variabili in X e/o di loro negazioni (indicate con $\overline{x_i}$). Un'*assegnazione di valori* per X è una funzione $\tau : X \to \{\text{TRUE}, \text{FALSE}\}$ che assegna a ogni variabile un valore di verità. Un letterale l è soddisfatto da τ se $l = x_j$ e $\tau(x_j) = \text{TRUE}$ oppure se $l = \overline{x_j}$ e $\tau(x_j) = \text{FALSE}$, per qualche $0 \leqslant j < n$. Una clausola è soddisfatta da τ se *almeno* un suo letterale lo è e una formula è soddisfatta da τ se *tutte* le sue clausole lo sono.

Il **problema della soddisfacibilità** (indicato con SAT) consiste nel decidere se una formula booleana in forma normale congiuntiva è soddisfacibile. In particolare, il problema 2-SAT è la restrizione di SAT al caso in cui le clausole contengano esattamente due letterali. Per mostrare che 2-SAT è risolvibile in tempo polinomiale, definiamo una riduzione da 2-SAT al problema di decidere se, dato un grafo orientato G e due nodi s e t, esiste un cammino in G da s a t: poiché quest'ultimo problema ammette un algoritmo di risoluzione polinomiale, abbiamo che anche 2-SAT ammette un tale algoritmo.

Data un formula booleana φ in forma normale congiuntiva formata da m clausole $c_0, c_1, \ldots, c_{m-1}$ su n variabili $x_0, x_1, \ldots, x_{n-1}$, costruiamo un grafo orientato $G = (V, E)$ contenente $2n$ vertici (due per ogni variabile booleana) e $2m$ archi (due per ogni clausola).[2] In particolare, per ogni variabile x_i, G include due vertici v_i^{pos} e v_i^{neg} corrispondenti, rispettivamente, ai letterali x_i e $\overline{x_i}$. Inoltre, per ogni clausola $\{l_1, l_2\}$ della formula, G include un arco tra il vertice corrispondente a $\overline{l_1}$ e quello corrispondente a l_2 e un arco tra il vertice corrispondente a $\overline{l_2}$ e quello corrispondente a l_1 (Figura 9.1): intuitivamente, questi due archi modellano il fatto che se uno dei due letterali della clausola non è soddisfatto, allora lo deve essere l'altro letterale.

Notiamo che il grafo G soddisfa la seguente proprietà: se esiste un cammino tra il vertice corrispondente a un letterale l e il vertice corrispondente a un letterale l', allora

[2] Senza perdita di generalità, possiamo supporre che nessuna clausola è formata da una variabile e dalla sua negazione: in effetti, una tale clausola è soddisfatta da qualunque assegnazione di valori e può, quindi, essere eliminata dalla formula.

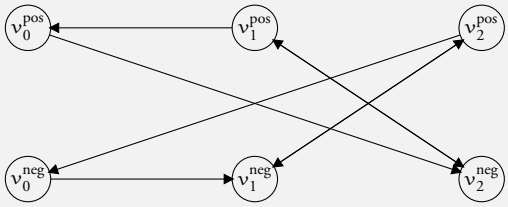


Figura 9.1 Un esempio di riduzione da 2-SAT al problema della ricerca di cammini in un grafo: le clausole sono $\{x_0, \overline{x_1}\}$, $\{\overline{x_0}, \overline{x_2}\}$, $\{x_1, x_2\}$ e $\{\overline{x_1}, \overline{x_2}\}$.

esiste un cammino tra il vertice corrispondente a $\overline{l'}$ e il vertice corrispondente a $\bar{l}$. Ad esempio, nella Figura 9.1, esiste il cammino da v_0^{neg} a v_2^{pos} (che passa per v_1^{neg}) ed esiste anche il cammino da v_2^{neg} a v_0^{pos} (che passa per v_1^{pos}). Mostriamo ora che φ è soddisfacibile se e solo se, per ogni variabile x_i con $0 \leqslant i < n$, abbiamo che v_i^{neg} non è raggiungibile da v_i^{pos} e v_i^{pos} non è raggiungibile da v_i^{neg} (ovvero, se non vi sono contraddizioni in φ).

Sia τ un'assegnazione di verità che soddisfa φ e supponiamo, per assurdo, che esista una variabile x_i tale che v_i^{neg} è raggiungibile da v_i^{pos} e che v_i^{pos} è raggiungibile da v_i^{neg}: supponiamo che $\tau(x_i) = \text{TRUE}$ (possiamo gestire il caso opposto in modo simile). Poiché esiste un cammino da v_i^{pos} a v_i^{neg} e poiché τ soddisfa x_i ma non soddisfa $\overline{x_i}$, deve esistere un arco (p, q) di tale cammino tale che il letterale l_p a cui corrisponde p è soddisfatto mentre il letterale l_q a cui corrisponde q non lo è: per definizione di G, φ include la clausola $\{\overline{l_p}, l_q\}$ la quale non è soddisfatta da τ, contraddicendo l'ipotesi.

Viceversa, supponiamo che, per ogni variabile x_i con $0 \leqslant i < n$, v_i^{neg} non è raggiungibile da v_i^{pos} oppure v_i^{pos} non è raggiungibile da v_i^{neg}, e mostriamo come costruire un'assegnazione di valori τ che soddisfa φ ripetendo il seguente procedimento fino a quando a tutte le variabili abbiamo assegnato un valore (notiamo la similitudine tra questo procedimento e l'algoritmo per decidere se un grafo è colorabile con due colori).

Sia l un letterale alla cui variabile corrispondente non abbiamo assegnato un valore e tale che il vertice p corrispondente a $\bar{l}$ non è raggiungibile dal vertice q corrispondente a l (per ipotesi, tale letterale deve esistere se vi sono ancora variabili a cui non abbiamo assegnato un valore). Estendiamo τ in modo da soddisfare tutti i letterali a cui corrispondono vertici raggiungibili da q e in modo da non soddisfare tutti i letterali a cui corrispondono vertici raggiungibili da p.

Tale estensione dell'assegnazione non crea contraddizioni, in quanto se i vertici corrispondenti a un letterale e alla sua negazione fossero entrambi raggiungibili da q, allora

(per simmetria) p sarebbe da essi raggiungibile e, quindi, esisterebbe un cammino da q a p. Inoltre, se un letterale l' corrispondente a un vertice raggiungibile da q fosse già stato non soddisfatto in un passo precedente, allora p sarebbe raggiungibile dal vertice corrispondente alla negazione di l' e, quindi, τ avrebbe già assegnato un valore alla variabile corrispondente a l.

Poiché a ogni passo, estendiamo τ soddisfacendo l e tutti i letterali che corrispondono a vertici raggiungibili da q, abbiamo che al termine del procedimento non può esistere un arco (r, s) tale che il letterale a cui corrisponde r è soddisfatto mentre quello a cui corrisponde s non lo è. Quindi, τ soddisfa tutte le clausole.

Nel caso del grafo mostrato nella Figura 9.1, possiamo scegliere $l = v_0^{pos}$, in quanto v_0^{neg} non è raggiungibile da v_0^{pos}. Quindi, estendiamo l'assegnazione τ (attualmente vuota) in modo da soddisfare tutti i letterali a cui corrispondono vertici raggiungibili da v_0^{pos}, che sono v_2^{neg} e v_1^{pos}. Pertanto, $\tau(x_0) = \text{TRUE}$, $\tau(x_1) = \text{TRUE}$ e $\tau(x_2) = \text{FALSE}$. Poiché a ogni variabile abbiamo assegnato un valore, il procedimento ha termine: in effetti, τ soddisfa le quattro clausole $\{x_0, \overline{x_1}\}$, $\{\overline{x_0}, \overline{x_2}\}$, $\{x_1, x_2\}$ e $\{\overline{x_1}, \overline{x_2}\}$.

9.1.3 Problemi NP-completi

In questo capitolo, siamo principalmente interessati a utilizzare il concetto di riducibilità per ottenere risultati negativi piuttosto che positivi, ovvero per dimostrare che un problema *non è* risolvibile facendo uso di risorse temporali limitate. Un semplice esempio di tale applicazione consiste nel dimostrare che il problema geometrico del minimo insieme convesso ha, in generale, una complessità temporale $\Omega(n \log n)$. Tale problema consiste nel trovare, dato un insieme di punti sul piano, il più piccolo (rispetto all'inclusione insiemistica) insieme convesso S che li contiene tutti:[3] nella Figura 9.2 mostriamo due esempi di insiemi convessi di cardinalità minima. Un punto $p \in S$ è un *estremo* di S, se esiste un semipiano passante per p tale che p è l'unico punto che giace sulla retta che delimita il semipiano: il problema del **minimo insieme convesso** consiste nel calcolare gli estremi di S come una lista (ciclicamente) ordinata di punti (ad esempio, la soluzione nella parte sinistra della Figura 9.2 è data da p_0, p_1, p_2 e p_3).

Notiamo che se i punti dell'istanza del problema giacciono su una parabola (come nella parte destra della Figura 9.2), allora la soluzione al problema del minimo insieme convesso consiste nella lista dei punti ordinata in base alle loro ascisse. Questa osservazione ci permette di ridurre il problema dell'ordinamento di n numeri interi $a_0, a_1, \ldots, a_{n-1}$ a quello del calcolo del minimo insieme convesso nel modo seguente: per ogni i con $0 \leqslant i \leqslant n-1$, definiamo un punto di coordinate (a_i, a_i^2). Chiaramente, gli n punti così costruiti giacciono sulla parabola di equazione $y = x^2$ e quindi il loro

[3] Ricordiamo che un un insieme S è detto *convesso* se, per ogni coppia di punti in S, il segmento che li unisce è interamente contenuto in S.

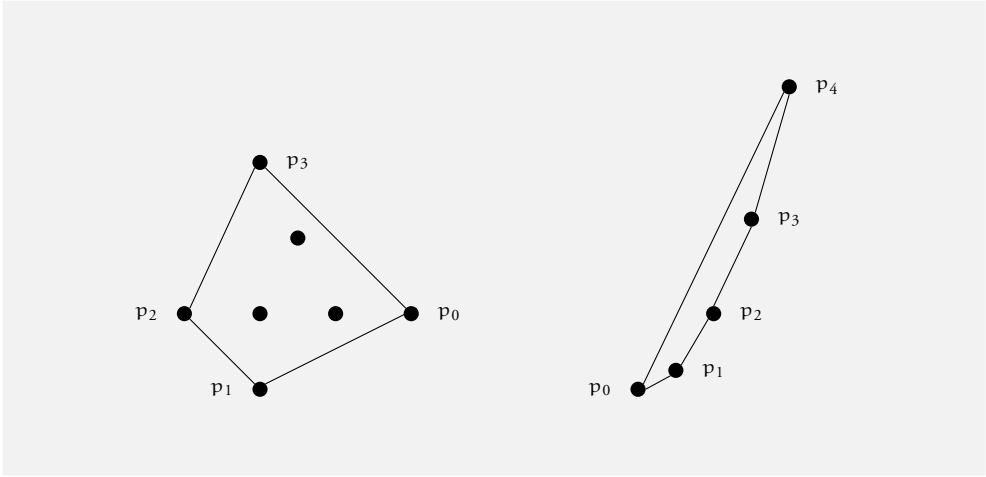


Figura 9.2 Due esempi di minimo insieme convesso.

minimo insieme convesso consiste nel loro elenco ordinato in base alle loro ascisse: tale elenco è dunque l'ordinamento degli n numeri interi. Poiché la costruzione suddetta può essere eseguita in tempo $O(n)$, se il problema del minimo insieme convesso è risolvibile in tempo $o(n \log n)$, allora anche il problema dell'ordinamento di n numeri interi è risolvibile in tempo $o(n \log n)$: nel Paragrafo 2.5.3 abbiamo visto che ciò non è in generale possibile, per cui abbiamo appena dimostrato un limite inferiore alla complessità temporale del problema del minimo insieme convesso.

In generale, se un problema Π è riducibile polinomialmente a un problema Π' e se sappiamo che Π non ammette un algoritmo di risoluzione polinomiale, allora possiamo concludere che neanche Π' ammette un tale algoritmo.

In altre parole, Π' è almeno tanto difficile quanto Π (notiamo che l'uso "positivo" del concetto di riducibilità consiste nell'affermare che Π è almeno tanto facile quanto Π'): quindi, le cattive notizie, ovvero, la non trattabilità di un problema, si propagano da sinistra verso destra (mentre le buone notizie lo fanno da destra verso sinistra).

Per definire la nozione di NP-completezza, introduciamo una restrizione del concetto di riducibilità in cui ogni istanza del problema di partenza viene trasformata in un'istanza del problema di arrivo, in modo che le due istanze siano entrambe positive oppure entrambe negative. Formalmente, un problema Π è **polinomialmente trasformabile** in un problema Π' se esiste un algoritmo polinomiale T tale che, per ogni sequenza binaria x, vale $x \in \Pi$ se e solo se T con x in ingresso restituisce una sequenza binaria in Π'. Chiaramente, se Π è polinomialmente trasformabile in Π', allora Π è polinomialmente

riducibile a Π': infatti, se esiste un algoritmo polinomiale A di risoluzione per Π', allora la composizione di T con A fornisce un algoritmo polinomiale di risoluzione per Π.

Un problema di decisione Π è **NP-completo** se Π appartiene a NP e se ogni altro problema in NP è polinomialmente trasformabile in Π. Quindi, Π è almeno tanto difficile quanto ogni altro problema in NP: in altre parole, se dimostriamo che Π è in P, allora abbiamo che l'intera classe NP è contenuta in P (e quindi le due classi coincidono).

È naturale a questo punto chiedersi se esistono problemi NP-completi (anche se il lettore avrà già intuito la risposta a tale domanda). Inoltre, se $P \neq NP$, possono esistere problemi che né appartengono a P e né sono NP-completi (un possibile candidato di questo tipo è il **problema aperto** dell'isomorfismo tra grafi, discusso nel Capitolo 6, del quale non conosciamo un algoritmo polinomiale di risoluzione e nemmeno una dimostrazione di NP-completezza).

Prima di discutere l'esistenza di un problema NP-completo, però, osserviamo che una volta dimostratane l'esistenza, possiamo sfruttare la proprietà di transitività della trasformabilità polinomiale per estendere l'insieme dei problemi siffatti. La definizione di trasformabilità soddisfa infatti la seguente proprietà: se Π_0 è polinomialmente trasformabile in Π_1 e Π_1 è polinomialmente trasformabile in Π_2, allora Π_0 è polinomialmente trasformabile in Π_2. A questo punto, volendo dimostrare che un certo problema computazionale Π è NP-completo, possiamo procedere in tre passi: prima dimostriamo che Π appartiene a NP mostrando l'esistenza del suo certificato polinomiale; poi individuiamo un altro problema Π', che già sappiamo essere NP-completo; infine, trasformiamo polinomialmente Π' in Π.

9.1.4 Teorema di Cook-Levin

Per applicare la strategia sopra esposta dobbiamo necessariamente trovare un primo problema NP-completo. Il **teorema di Cook-Levin** afferma che SAT è NP-completo. Osserviamo che SAT appartiene a NP, in quanto ogni formula soddisfacibile ammette una dimostrazione breve e facile da verificare che consiste nella specifica di un'assegnazione di valori che soddisfa la formula. La parte difficile del teorema di Cook-Levin consiste, quindi, nel mostrare che ogni problema in NP è polinomialmente trasformabile in SAT.

Non diamo la dimostrazione del suddetto teorema, ma ci limitiamo a fornire una breve descrizione dell'approccio utilizzato. Dato un problema $\Pi \in NP$, sappiamo che esiste un algoritmo polinomiale V e un polinomio p tali che, per ogni sequenza binaria x, se $x \in \Pi$, allora esiste una sequenza y di lunghezza $p(|x|)$ tale che V con x e y in ingresso termina restituendo il valore TRUE, mentre se $x \notin \Pi$, allora, per ogni sequenza y, V con x e y in ingresso termina restituendo il valore FALSE. L'idea della dimostrazione consiste nel costruire, per ogni x, in tempo polinomiale una formula booleana φ_x le cui uniche variabili *libere* sono $p(|x|)$ variabili $y_0, y_1, \ldots, y_{p(|x|)-1}$, intendendo con ciò che la soddisfacibilità della formula dipende solo dai valori assegnati a tali variabili:

intuitivamente, la variabile y_i corrisponde al valore del bit in posizione i all'interno della sequenza y. La formula φ_x in un certo senso *simula* il comportamento di V con x e y in ingresso ed è soddisfacibile solo se tale computazione termina in modo affermativo (ovvero, se y è una dimostrazione che x appartiene a Π). Il fatto che possiamo costruire una tale formula non dovrebbe sorprenderci più di tanto, se consideriamo che, in fin dei conti, l'esecuzione di un algoritmo all'interno di un calcolatore avviene attraverso circuiti logici le cui componenti di base sono porte logiche che realizzano la disgiunzione (*or*), la congiunzione (*and*) e la negazione (*not*).

Per convincerci ulteriormente che la dimostrazione del teorema di Cook-Levin così tracciata può effettivamente essere realizzata, mostriamo, ad esempio, come un'istanza del problema di decidere se un grafo G può essere colorato con tre colori (che è chiaramente un problema in NP) può essere descritta mediante una formula booleana φ_G. In particolare, φ_G non sarà in forma normale congiuntiva: tuttavia, possiamo facilmente dimostrare che una formula booleana ψ può essere trasformata in tempo polinomiale in una formula booleana ψ' in forma normale congiuntiva, tale che ψ è soddisfacibile se e solo se ψ' è soddisfacibile.

Sappiamo che $G = (V, E)$ può essere rappresentato mediante la matrice di adiacenza A tale che $A[i][j] = 1$ se e solo se $(i, j) \in E$. A tale matrice facciamo corrispondere $n \times n$ variabili booleane $a_{i,j}$ e usiamo in φ_G le seguenti formule booleane, per $0 \leqslant i, j < n$:

$$A_{i,j} = \begin{cases} a_{i,j} & \text{se } A[i][j] = 1 \\ \overline{a_{i,j}} & \text{altrimenti} \end{cases}$$

(in altre parole, queste formule forzano le variabili $a_{i,j}$ a rappresentare la matrice di adiacenza di G). Per ogni vertice $i \in V$, introduciamo poi tre variabili booleane r_i, g_i e b_i che corrispondono ai tre possibili colori che possono essere assegnati al vertice (quindi, queste sono le variabili libere i cui valori di verità forniscono la dimostrazione y). Per impedire che due colori vengano assegnati allo stesso vertice, usiamo in φ_G le seguenti formule booleane, per $0 \leqslant i < n$:

$$B_i = (r_i \wedge \overline{g_i} \wedge \overline{b_i}) \vee (\overline{r_i} \wedge g_i \wedge \overline{b_i}) \vee (\overline{r_i} \wedge \overline{g_i} \wedge b_i)$$

Infine, per verificare che l'assegnazione dei colori ai vertici sia compatibile con gli archi del grafo, φ_G usa le seguenti formule booleane, per $0 \leqslant i, j < n$:

$$C_{i,j} = a_{i,j} \Rightarrow [(r_i \wedge \overline{r_j}) \vee (g_i \wedge \overline{g_j}) \vee (b_i \wedge \overline{b_j})] = \overline{a_{i,j}} \vee (r_i \wedge \overline{r_j}) \vee (g_i \wedge \overline{g_j}) \vee (b_i \wedge \overline{b_j})$$

(informalmente, $C_{i,j}$ afferma che se vi è un arco tra i due vertici i e j, allora questi due vertici non possono avere lo stesso colore). In conclusione, la formula φ_G è la seguente:

$$\bigwedge_{0 \leqslant i,j < n} A_{i,j} \wedge \bigwedge_{0 \leqslant i < n} B_i \wedge \bigwedge_{0 \leqslant i,j < n} C_{i,j}$$

Chiaramente, φ_G è soddisfacibile se e solo se i vertici di G possono essere colorati con tre colori. Il teorema di Cook-Levin afferma sostanzialmente che quanto abbiamo appena fatto per il problema della colorazione può in realtà essere fatto per qualunque problema in NP.

Notiamo che la NP-completezza di SAT non implica che il problema della soddisfacibilità di formule booleane in forma normale disgiuntiva sia anch'esso NP-completo: se una formula è in **forma normale disgiuntiva**, allora una clausola è soddisfatta da un'assegnazione di valori τ se tutti i suoi letterali lo sono e la formula è soddisfatta da τ se almeno una sua clausola lo è. In tal caso, possiamo mostrare che il problema della soddisfacibilità è risolvibile in tempo polinomiale e, quindi, non è molto probabilmente NP-completo.

9.1.5 Problemi di ottimizzazione

Prima di passare a dimostrare diversi risultati di NP-completezza, introduciamo il concetto di problema di ottimizzazione, inteso in qualche modo come estensione di quello di problema decisionale.

In un **problema di ottimizzazione**, a ogni istanza del problema associamo un insieme di soluzioni possibili e a ciascuna soluzione associamo una misura (che può essere un costo oppure un profitto): il problema consiste nel trovare, data un'istanza, una soluzione ottima, ovvero una soluzione di misura minima se la misura è un costo, una di misura massima altrimenti.

Abbiamo già incontrato diversi problemi di ottimizzazione nei capitoli precedenti (ad esempio, il problema della sequenza ottima di moltiplicazioni di matrici del Paragrafo 2.6.2 oppure quello della sotto-sequenza comune più lunga del Paragrafo 2.7.1).

Osserviamo che a ogni problema di ottimizzazione corrisponde in modo abbastanza naturale un problema di decisione definito nel modo seguente: data un'istanza del problema e dato un valore k, decidere se la misura della soluzione ottima è inferiore a k (nel caso di costi) oppure superiore a k (nel caso di profitti).

Nella maggior parte dei problemi di ottimizzazione che sorgono nella realtà, abbiamo che se il corrispondente problema di decisione è risolvibile in tempo polinomiale, allora anche il problema di ottimizzazione è risolvibile in tempo polinomiale.

Ciò è principalmente dovuto al fatto che il valore massimo che la misura di una soluzione può assumere è limitato esponenzialmente dalla lunghezza dell'istanza: questa osservazione ci consente di ridurre polinomialmente un problema di ottimizzazione al suo corrispondente problema di decisione, operando in base a un meccanismo simile a quello di ricerca binaria descritto nel Paragrafo 2.4.1.

Quindi, se il problema di decisione associato a un problema di ottimizzazione è NP-completo, abbiamo che quest'ultimo non può essere risolto in tempo polinomiale a

meno che P = NP: nel paragrafo finale di questo capitolo, analizzeremo in dettaglio uno dei più famosi problemi di ottimizzazione intrinsecamente difficili.

9.2 Esempi e tecniche di NP-completezza

A partire da SAT, mostriamo ora come sia possibile verificare la NP-completezza di altri problemi computazionali: alcuni che abbiamo esaminato nei capitoli precedenti e che abbiamo dichiarato essere NP-completi e altri che useremo come problemi di passaggio nelle catene di trasformazioni polinomiali.

Per prima cosa, dimostriamo che la restrizione 3-SAT di SAT a clausole formate da *esattamente* tre letterali è anch'esso un problema NP-completo (chiaramente 3-SAT è in NP per lo stesso motivo per cui lo è SAT).

9.2.1 Tecnica di sostituzione locale

Sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di variabili booleane $\{x_0, \ldots, x_{n-1}\}$. Vogliamo costruire, in tempo polinomiale, un nuovo insieme D di clausole, ciascuna di cardinalità 3, costruite a partire da un insieme Z di variabili booleane e tali che C è soddisfacibile se e solo se D è soddisfacibile. A tale scopo usiamo una tecnica di trasformazione detta di **sostituzione locale**, in base alla quale costruiremo D e Z sostituendo ogni clausola $c \in C$ con un sottoinsieme D_c di D in modo indipendente dalle altre clausole di C: l'insieme D è quindi uguale a $\cup_{c \in C} D_c$ e Z è l'unione di tutte le variabili booleane che appaiono in D.

Data una clausola $c = \{l_0, \ldots, l_{k-1}\}$ dell'insieme C, definiamo D_c distinguendo i seguenti quattro casi:

1. $k = 1$: in questo caso, $D_c = \{\{l_0, y_0^c, y_1^c\}, \{l_0, y_0^c, \overline{y_1^c}\}, \{l_0, \overline{y_0^c}, y_1^c\}, \{l_0, \overline{y_0^c}, \overline{y_1^c}\}\}$ (chiaramente D_c è soddisfacibile se e solo se l_0 è soddisfatto);

2. $k = 2$: in questo caso, $D_c = \{\{l_0, l_1, y_0^c\}, \{l_0, l_1, \overline{y_0^c}\}$ (chiaramente D_c è soddisfacibile se e solo se l_0 oppure l_1 è soddisfatto);

3. $k = 3$: in questo caso, D_c è formato dalla sola clausola c;

4. $k > 3$: in questo caso, che è il più difficile, l'insieme D_c contiene un insieme di $k-2$ clausole collegate tra di loro attraverso nuove variabili booleane e tali che la loro soddisfacibilità sia equivalente a quella di c. Formalmente, D_c è l'insieme

$$\{\{l_0, l_1, y_0^c\}, \{\overline{y_0^c}, l_2, y_1^c\}, \{\overline{y_1^c}, l_3, y_2^c\}, \ldots, \{\overline{y_{k-5}^c}, l_{k-3}, y_{k-4}^c\}, \{\overline{y_{k-4}^c}, l_{k-2}, l_{k-1}\}$$

Dalla definizione di D_c abbiamo che

$$Z = X \cup \bigcup_{c \in C \wedge |c|=1} \{y_0^c, y_1^c\} \cup \bigcup_{c \in C \wedge |c|=2} \{y_0^c\} \cup \bigcup_{c \in C \wedge |c|>3} \{y_0^c, \ldots, y_{|c|-4}^c\}$$

e che la costruzione dell'istanza di 3-SAT può essere eseguita in tempo polinomiale.

Supponiamo che esista un'assegnazione τ di verità alle variabili di X che soddisfa C. Quindi, τ soddisfa c per ogni clausola $c \in C$: mostriamo che tale assegnazione può essere estesa alle nuove variabili di tipo y^c introdotte nel definire D_c, in modo che tutte le clausole in esso contenute siano soddisfatte (da quanto detto sopra, possiamo supporre che $|c| > 3$). Poiché c è soddisfatta da τ, deve esistere h tale che τ soddisfa l_h con $0 \leqslant h \leqslant |c| - 1$: estendiamo τ assegnando il valore TRUE a tutte le variabili y_i^c con $0 \leqslant i \leqslant h-2$ e il valore FALSE alle rimanenti variabili. In questo modo, siamo sicuri che la clausola di D_c contenente l_h è soddisfatta (da l_h stesso), le clausole che la precedono sono soddisfatte grazie al loro terzo letterale e quelle che la seguono lo sono grazie al loro primo letterale.

Viceversa, supponiamo che esista un'assegnazione τ di verità alle variabili di Z che soddisfi tutte le clausole in D e, per assurdo, che tale assegnazione ristretta alle sole variabili di X non soddisfi almeno una clausola $c \in C$, ovvero che tutti i letterali contenuti in c non siano soddisfatti (di nuovo, ipotizziamo che $|c| > 3$). Ciò implica che tutte le variabili di tipo y^c devono essere vere, perché altrimenti una delle prime $|c| - 3$ clausole in D_c non è soddisfatta, contraddicendo l'ipotesi che τ soddisfa tutte le clausole in D. Quindi, $\tau(y_{|c|-4}^c) = \text{TRUE}$, ovvero $\tau(\overline{y_{|c|-4}^c}) = \text{FALSE}$: poiché abbiamo supposto che anche $l_{|c|-2}$ e $l_{|c|-1}$ non sono soddisfatti, l'ultima clausola in D_c non è soddisfatta, contraddicendo nuovamente l'ipotesi che τ soddisfa tutte le clausole in D.

In conclusione, abbiamo dimostrato che C è soddisfacibile se e solo se D lo è e, quindi, che il problema SAT è trasformabile in tempo polinomiale nel problema 3-SAT: quindi, quest'ultimo è NP-completo. Notiamo che, in modo simile a quanto fatto per 3-SAT, possiamo mostrare la NP-completezza del problema della soddisfacibilità nel caso in cui le clausole contengono esattamente k letterali, per ogni $k \geqslant 3$: tale affermazione non si estende però al caso in cui $k = 2$, in quanto, come abbiamo visto nel Paragrafo 9.1.2, in questo caso il problema diviene risolvibile in tempo polinomiale e, quindi, difficilmente esso è anche NP-completo.

La NP-completezza di 3-SAT ha un duplice valore: da un lato esso mostra che la difficoltà computazionale del problema della soddisfacibilità non dipende dalla lunghezza delle clausole (fintanto che queste contengono almeno tre letterali), dall'altro ci consente nel seguito di usare 3-SAT come problema di partenza, il quale, avendo istanze più regolari, è più facile da utilizzare per sviluppare trasformazioni volte a dimostrare risultati di NP-completezza.

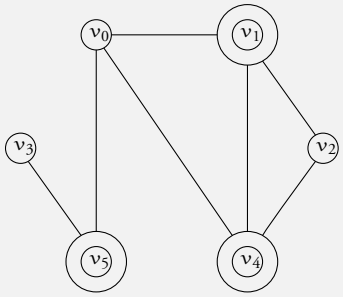


Figura 9.3 Un esempio di minimo ricoprimento tramite vertici.

9.2.2 Tecnica di progettazione di componenti

Per dimostrare la NP-completezza del problema del massimo insieme indipendente in un grafo (o meglio della sua versione decisionale), mostriamo prima che il seguente problema, detto **minimo ricoprimento tramite vertici** è NP-completo: dato un grafo $G = (V, E)$ e un intero $k \geqslant 0$, esiste un sottoinsieme V' di V con $|V'| \leqslant k$, tale che ogni arco del grafo è coperto da V' ovvero, per ogni arco $(u, v) \in E$, $u \in V'$ oppure $v \in V'$? Nella Figura 9.3 mostriamo un esempio di ricoprimento tramite 3 vertici del grafo delle conoscenze discusso nel Paragrafo 6.1.1. Notiamo che, in questo caso, un qualunque sottoinsieme di vertici di cardinalità minore di 3, non può essere un ricoprimento: in effetti, due vertici sono necessari per coprire il triangolo formato da v_1, v_2 e v_4 e un ulteriore vertice è necessario per coprire l'arco tra v_3 e v_5.

Il problema del minimo ricoprimento tramite vertici ammette dimostrazioni brevi e verificabili in tempo polinomiale: tali dimostrazioni sono i sottoinsiemi dell'insieme dei vertici del grafo che costituiscono un ricoprimento degli archi di cardinalità al più k.

Mostriamo ora che 3-SAT è trasformabile in tempo polinomiale nel problema del minimo ricoprimento tramite vertici: a tale scopo, faremo uso di una tecnica più sofisticata di quella vista nel paragrafo precedente, che viene generalmente indicata con il nome di **progettazione di componenti**. In particolare, la trasformazione opera definendo, per ogni variabile, una componente (*gadget*) del grafo il cui scopo è quello di modellare l'assegnazione di verità alla variabile e, per ogni clausola, una componente il cui scopo è quello di modellare la soddisfacibilità della clausola. I due insiemi di componenti sono poi collegati tra di loro per garantire che l'assegnazione alle variabili soddisfi tutte le clausole.

Più precisamente, sia $C = \{c_0, \ldots, c_{m-1}\}$ un insieme di m clausole costruite a partire dall'insieme X di variabili booleane $\{x_0, \ldots, x_{n-1}\}$, tali che $|c_i| = 3$ per $0 \leqslant i < m$. Vogliamo definire un grafo G e un intero k tale che C è soddisfacibile se e solo se G include un ricoprimento di esattamente k vertici.

Per ogni variabile x_i con $0 \leqslant i < n$, G include due vertici v_i^{vero} e v_i^{falso} collegati tra di loro mediante un arco. Queste sono le componenti di verità del grafo, in quanto ogni ricoprimento di G deve necessariamente includere almeno un vertice tra v_i^{vero} e v_i^{falso} per $0 \leqslant i < n$: il valore di k sarà scelto in modo tale che ne includa esattamente uno, ovvero quello corrispondente al valore di verità della variabile corrispondente.

Per ogni clausola c_j con $0 \leqslant j < m$, G include una cricca di tre vertici v_j^0, v_j^1 e v_j^2. Queste sono le componenti corrispondenti alla soddisfacibilità delle clausole, in quanto ogni ricoprimento di G deve necessariamente includere almeno due vertici tra v_j^0, v_j^1 e v_j^2 per $0 \leqslant j < m$: il valore di k sarà scelto in modo tale che ne includa esattamente due, in modo che quello non selezionato corrisponda a un letterale certamente soddisfatto all'interno della clausola corrispondente.

Le componenti di verità e quelle di soddisfacibilità sono collegate tra di loro aggiungendo un arco tra i vertici contenuti nelle prime componenti con i corrispondenti vertici contenuti nelle seconde componenti. Più precisamente, per ogni i, j e k con $0 \leqslant i < n$, $0 \leqslant j < m$ e $0 \leqslant h < 3$:

- il vertice v_i^{vero} è collegato al vertice v_j^h se e solo se l'$(h+1)$-esimo letterale della clausola c_j è x_i;
- il vertice v_i^{falso} è collegato al vertice v_j^h se e solo se l'$(h+1)$-esimo letterale della clausola c_j è $\overline{x_i}$.

Nella Figura 9.4 mostriamo il grafo così ottenuto a partire dal seguente insieme di clausole: $\{x_0, x_1, \overline{x_2}\}$, $\{\overline{x_0}, x_1, \overline{x_2}\}$ e $\{\overline{x_0}, \overline{x_1}, \overline{x_2}\}$. Rimane da definire il valore di k: come abbiamo già detto, vogliamo che tale valore ci costringa a prendere esattamente un vertice per ogni componente di verità ed esattamente due vertici per ogni componente di soddisfacibilità. Poiché abbiamo n componenti del primo tipo e m componenti del secondo tipo, poniamo $k = n + 2m$.

Sia τ un'assegnazione di verità che soddisfa C, ovvero tale che, per ogni clausola c_j con $0 \leqslant j < m$, esiste un letterale soddisfatto contenuto in c_j: indichiamo con p_j la posizione del primo tale letterale, dove $0 \leqslant p_j < 3$. Costruiamo un ricoprimento V' nel modo seguente: per $0 \leqslant i < n$, V' include v_i^{vero} se $\tau(x_i) = \text{TRUE}$, altrimenti include v_i^{falso}; inoltre, per $0 \leqslant j < m$, V' include v_j^h dove $0 \leqslant h < 3$ e $h \neq p_j$ (notiamo che l'arco tra $v_j^{p_j}$ e il corrispondente vertice contenuto in una componente di verità è coperto da quest'ultimo). Poiché, per ogni componente di verità, V' include un vertice e, per

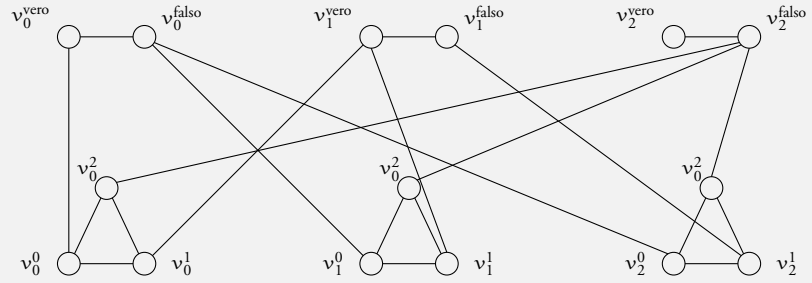


Figura 9.4 Un esempio di riduzione da 3-SAT al problema del minimo ricoprimento tramite vertici: le clausole sono $\{x_0, x_1, \overline{x_2}\}$, $\{\overline{x_0}, x_1, \overline{x_2}\}$ e $\{\overline{x_0}, \overline{x_1}, \overline{x_2}\}$.

ogni componente di soddisfacibilità, ne include due, abbiamo che V' è un ricoprimento e che $|V'| = n + 2m = k$.

Facendo riferimento all'esempio mostrato nella Figura 9.4, supponiamo che τ assegni il valore TRUE alla sola variabile x_0: in tal caso, V' include i vertici v_0^{vero}, v_1^{falso} e v_2^{falso}. Il primo letterale soddisfatto contenuto nella prima clausola è x_0, che si trova in posizione 0: quindi, V' include i due vertici v_0^1 e v_0^2. Analogamente, possiamo mostrare che V' include i vertici v_1^0, v_1^1, v_2^0 e v_2^2 e che, quindi, è un ricoprimento del grafo di cardinalità $3 + 6 = 9$.

Viceversa, supponiamo che V' sia un ricoprimento di G che include esattamente $n + 2m$ nodi. Ciò implica che V' deve includere un vertice per ogni componente di verità e due vertici per ogni componente di soddisfacibilità. Definiamo un'assegnazione di verità τ tale che $\tau(x_i) = \text{TRUE}$ se e solo se $v_i^{vero} \in V'$ per $0 \leqslant i < n$: chiaramente, τ è un'assegnazione di verità corretta (ovvero, non assegna alla stessa variabile due valori di verità diversi). Inoltre, per ogni clausola c_j con $0 \leqslant j < m$, deve esistere h con $0 \leqslant h < 3$ tale che $v_j^h \notin V'$: l'arco che unisce v_j^h al vertice corrispondente contenuto in una componente di verità deve, quindi, essere coperto da quest'ultimo che è incluso in V'. Pertanto, l'$(h+1)$-esimo letterale in c_j è soddisfatto da τ e la clausola c_j è anch'essa soddisfatta. Facendo sempre riferimento all'esempio mostrato nella Figura 9.4, supponiamo che V' includa i vertici v_0^{falso}, v_1^{falso}, v_2^{falso}, v_0^0, v_0^1,v_1^0, v_1^1, v_2^0 e v_2^1: in tal caso, τ assegna il valore FALSE a tutte e tre le variabili booleane. Tale assegnazione soddisfa tutte le clausole di C: ad esempio, della componente corrispondente alla prima clausola V' non include il vertice v_0^2 ma della componente corrispondente a x_2 include il vertice v_2^{falso}, per cui $\tau(\overline{x_2}) = \text{TRUE}$ e la clausola è soddisfatta.

Poiché il grafo G può essere costruito in tempo polinomiale a partire dall'insieme di clausole C, abbiamo che 3-SAT è polinomialmente trasformabile nel problema del minimo ricoprimento tramite vertici e, quindi, che quest'ultimo è NP-completo.

9.2.3 Tecnica di similitudine

A partire dal problema del minimo ricoprimento tramite vertici siamo ora in grado di dimostrare la NP-completezza del seguente: dato un grafo $G = (V, E)$ e un intero $k \geqslant 0$, esiste un sottoinsieme V' di V con $|V'| \geqslant k$, tale che V' è un insieme indipendente ovvero, per ogni arco $(u, v) \in E$, $u \notin V'$ oppure $v \notin V'$? In questo caso, la trasformazione è molto più semplice di quelle viste finora e si basa sulla tecnica della **similitudine**, che consiste appunto nel mostrare come un problema sia simile a uno già precedentemente dimostrato essere NP-completo.

Nel nostro caso, dato un grafo $G = (V, E)$, il concetto di similitudine si manifesta nell'equivalenza tra il fatto che $V' \subseteq V$ è un insieme indipendente di G e quello che $V - V'$ è un ricoprimento tramite vertici di G. Infatti, se V' è un insieme indipendente, allora $V - V'$ è un ricoprimento in quanto, se così non fosse, esisterebbe un arco $(u, v) \in E$ tale che $u \notin V - V'$ e $v \notin V - V'$: quindi, esisterebbe un arco $(u, v) \in E$ tale che $u \in V'$ e $v \in V'$ contraddicendo l'ipotesi che V' è un insieme indipendente. Viceversa, se $V - V'$ è un ricoprimento tramite vertici, allora V' è un insieme indipendente in quanto, se così non fosse, esisterebbe un arco $(u, v) \in E$ tale che $u \in V'$ e $v \in V'$: quindi, esisterebbe un arco $(u, v) \in E$ tale che $u \notin V - V'$ e $v \notin V - V'$ contraddicendo l'ipotesi che $V - V'$ è un ricoprimento. Ad esempio, nel caso della Figura 9.3, abbiamo che l'insieme formato dai vertici v_1, v_4 e v_5 è un ricoprimento tramite vertici, mentre l'insieme complementare formato dai vertici v_0, v_2 e v_3 è un insieme indipendente. Pertanto, il problema del minimo ricoprimento tramite vertici è trasformabile in quello del massimo insieme indipendente e viceversa.

Notiamo che il problema del massimo insieme indipendente ammette dimostrazioni brevi e verificabili in tempo polinomiale, che altro non sono se non i sottoinsiemi di vertici che formano un insieme indipendente. Abbiamo pertanto aggiunto, alla nostra lista di problemi NP-completi, il problema del massimo insieme indipendente.

9.2.4 Tecnica di restrizione

L'ultimo esempio di dimostrazione di NP-completezza che forniamo si basa sulla tecnica più semplice in assoluto, detta della **restrizione**, che consiste nel mostrare come un problema già noto essere NP-completo sia un caso speciale di un altro problema: da ciò ovviamente deriva la NP-completezza di quest'ultimo.

Come esempio, consideriamo il problema del **minimo insieme di campionamento**, che è definito nel modo seguente: dato un insieme C di sottoinsiemi di un insieme A e

dato un numero intero $k \geqslant 0$, esiste un sottoinsieme A' di A tale che $|A'| \leqslant k$ e A' è un campionamento di C ovvero, per ogni insieme $c \in C$, $c \cap A' \neq \emptyset$? Possiamo restringere questo problema a quello del minimo ricoprimento tramite vertici, limitandoci a considerare istanze in cui ciascun elemento di C contiene esattamente due elementi di A: intuitivamente, A corrisponde all'insieme dei vertici del grafo e C all'insieme degli archi.

Poiché il problema del minimo ricoprimento tramite vertici è NP-completo e poiché quello del minimo insieme di campionamento ammette dimostrazioni brevi e verificabili in tempo polinomiale (costituite dal campione A'), abbiamo che anche quest'ultimo problema è NP-completo: d'altra parte, se riuscissimo a progettare un algoritmo polinomiale per questo, potremmo ugualmente risolvere il problema del minimo ricoprimento tramite vertici in tempo polinomiale, applicando tale algoritmo alle sole istanze ristrette.

9.2.5 Come dimostrare risultati di NP-completezza

Il concetto di NP-completezza è stato introdotto alla metà degli anni '70. Da allora, migliaia di problemi computazionali sono stati dimostrati essere NP-completi, di tipologie diverse e provenienti da molte aree applicative. Un punto cruciale nel cercare di dimostrare che un nuovo problema Π è NP-completo consiste nella scelta del problema da cui partire, ovvero il problema NP-completo Π' che deve essere trasformato in Π (notiamo che un tipico errore che si commette inizialmente è quello di pensare che Π deve essere trasformato in Π' e non viceversa). A tale scopo, nel loro libro *Algorithm Design*, Jon Kleinberg e Eva Tardos identificano i seguenti sei tipi primitivi di problema, suggerendo per ciascuno uno o più potenziali candidati a svolgere il ruolo del problema computazionale Π'.

Problemi di sottoinsiemi massimali. Dato un insieme di oggetti, cerchiamo un suo sottoinsieme di cardinalità massima che soddisfi determinati requisiti: un tipico esempio di problemi siffatti è il problema del massimo insieme indipendente.

Problemi di sottoinsiemi minimali. Dato un insieme di oggetti, cerchiamo un suo sottoinsieme di cardinalità minima che soddisfi determinati requisiti: due tipici esempi di problemi siffatti sono il problema del minimo ricoprimento tramite vertici e quello del minimo insieme di campionamento.

Problemi di partizionamento. Dato un insieme di oggetti, cerchiamo una sua partizione nel minor numero possibile di sottoinsiemi disgiunti che soddisfino determinati requisiti (in alcuni casi, viene anche richiesto che i sottoinsiemi della partizione siano scelti tra una collezione specificata nell'istanza del problema): un tipico esempio di problemi siffatti è il problema della colorazione di grafi.

Problemi di ordinamento. Dato un insieme di oggetti, cerchiamo un suo ordinamento che soddisfi determinati requisiti: tipici esempi di problemi di questo tipo sono

il problema del circuito hamiltoniano e quello del commesso viaggiatore (di cui parleremo nel prossimo paragrafo).

Problemi numerici. Dato un insieme di numeri interi, cerchiamo un suo sottoinsieme che soddisfi determinati requisiti: tipici esempi di problemi di questo tipo sono il problema della partizione e quello della bisaccia. Notiamo che la difficoltà di questi problemi risiede principalmente nel dover trattare numeri arbitrariamente grandi: in effetti, i due problemi suddetti, ristretti a istanze in cui i numeri in gioco sono polinomialmente limitati rispetto alla lunghezza dell'istanza, sono risolvibili in tempo polinomiale (Paragrafo 2.7.4).

Problemi di soddisfacimento di vincoli. Dato un sistema di vincoli espressi, generalmente, mediante formule booleane o equazioni lineari su uno specifico insieme di variabili, cerchiamo un'assegnazione alle variabili che soddisfi il sistema: un tipico esempio di problemi siffatti è il problema della soddisfacibilità (eventualmente ristretto a istanze con clausole contenenti esattamente tre letterali).

Una volta scelto il problema Π' da cui partire, la progettazione della trasformazione di Π' in Π è un compito difficile tanto quanto quello di progettare un algoritmo polinomiale di risoluzione per Π: in effetti, in *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Michael Garey e David Johnson suggeriscono di procedere parallelamente nelle due attività, in quanto le difficoltà che si incontrano nella progettazione di un algoritmo possono fornire suggerimenti alla progettazione della trasformazione e viceversa. Sebbene la capacità di dimostrare risultati di NP-completezza sia un'abilità che, una volta acquisita, può risultare poi di facile applicazione, non è certo possibile, come nel caso della capacità di sviluppare algoritmi efficienti, spiegarla in modo formale. Ciò nondimeno, Steven Skiena, nelle sue dispense di un corso su algoritmi, fornisce i seguenti suggerimenti di cui possiamo tener conto quando ci accingiamo a voler dimostrare che un dato problema è NP-completo:

- rendiamo Π' il più semplice possibile (ad esempio, conviene usare 3-SAT invece di SAT);
- rendiamo Π il più difficile possibile, eventualmente aggiungendo (temporaneamente) vincoli ulteriori;
- identifichiamo in Π le soluzioni *canoniche* e introduciamo qualche forma di penalizzazione nei confronti di una qualunque soluzione che non sia canonica (ad esempio, nel caso del problema del minimo ricoprimento tramite vertici, una soluzione canonica è formata da un vertice per ogni componente di verità e due vertici per ogni componente di soddisfacibilità);
- prima di produrre gadget (nel caso della tecnica della progettazione di componenti), ragioniamo ad alto livello chiedendoci cosa e come intendiamo fare per forzare a scegliere soluzioni canoniche.

Per quanto utili, questi suggerimenti sono abbastanza vaghi: nella realtà, non esiste altro modo di imparare a progettare trasformazioni tra problemi computazionali se non facendolo. Per questo motivo, in questo capitolo abbiamo preferito fornire pochi esempi di tali trasformazioni, lasciando al lettore il compito di cimentarsi con altri problemi, presi magari dalla lista di problemi NP-completi presenti nel libro di Garey e Johnson.

9.3 Algoritmi di approssimazione

Dimostrare che un problema è NP-completo significa rinunciare a progettare per esso un algoritmo polinomiale di risoluzione (a meno che non crediamo che P sia uguale a NP). A questo punto, però, ci chiediamo come dobbiamo comportarci: dopo tutto, il problema deve essere risolto. A questo interrogativo possiamo rispondere in diversi modi. Il primo e il più semplice consiste nell'ignorare la complessità temporale intrinseca del problema, sviluppare comunque un algoritmo di risoluzione e sperare che nella pratica, ovvero con istanze provenienti dal mondo reale, il tempo di risoluzione sia significativamente minore di quello previsto: dopo tutto, l'analisi nel caso pessimo, in quanto tale, non ci dice come si comporterà il nostro algoritmo nel caso di specifiche istanze.

Un secondo approccio, in linea con quello precedente ma matematicamente più fondato, consiste nell'analizzare l'algoritmo da noi progettato nel caso medio rispetto a una specifica distribuzione di probabilità: questo è quanto abbiamo fatto nel caso dell'algoritmo di ordinamento per distribuzione (Paragrafo 2.5.4). Vi sono due tipi di problematiche che sorgono quando vogliamo perseguire tale approccio. La prima consiste nel fatto che un'analisi probabilistica del comportamento dell'algoritmo è quasi sempre difficile e richiede strumenti di calcolo delle probabilità talvolta molto sofisticati. La seconda e, probabilmente, più grave questione è che l'analisi probabilistica richiede la conoscenza della distribuzione di probabilità con cui le istanze si presentano nel mondo reale: purtroppo, quasi mai conosciamo tale distribuzione e, pertanto, siamo costretti a ipotizzare che essa sia una di quelle a noi più familiari, come la distribuzione uniforme.

Un terzo approccio si applica al caso di problemi di ottimizzazione, per i quali a ogni soluzione è associata una misura e il cui scopo consiste nel trovare una soluzione di misura ottimale: in tal caso, possiamo rinunciare alla ricerca di soluzioni ottime e accontentarci di progettare algoritmi efficienti che producano sì soluzioni peggiori, ma non troppo. In particolare, diremo che A è un **algoritmo di r-approssimazione** per il problema di ottimizzazione Π se, per ogni istanza x di Π, abbiamo che A con x in ingresso restituisce una soluzione di x la cui misura è al più r volte quella di una soluzione ottima (nel caso la misura sia un costo) oppure almeno $\frac{1}{r}$-esimo di quella di una soluzione ottima (nel caso la misura sia un profitto), dove r è una costante reale strettamente maggiore di 1.

Per chiarire meglio tale concetto, consideriamo il problema del minimo ricoprimento tramite vertici, che nella sua versione di ottimizzazione consiste nel trovare un sottoin-

sieme dei vertici di un grafo di cardinalità minima che copra tutti gli archi del grafo stesso. Il Codice 9.2 realizza un algoritmo di approssimazione per tale problema basato sul paradigma dell'algoritmo goloso (abbiamo già detto che nel caso degli algoritmi di approssimazione, il paradigma dell'algoritmo goloso risulta spesso essere efficace). In particolare, dopo aver inizializzato la soluzione ponendola uguale all'insieme vuoto (righe 3 e 4), il codice esamina uno dopo l'altro tutti gli archi del grafo (righe 5–11): ogni qualvolta ne trova uno i cui due estremi non sono stati selezionati (riga 7), include entrambi gli estremi nella soluzione (righe 8 e 9).

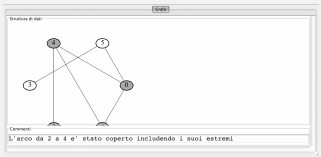
La complessità temporale del Codice 9.2 è $O(n^2)$, in quanto ogni iterazione dei due cicli annidati uno dentro l'altro richiede un numero costante di operazioni. Inoltre, la soluzione prodotta dall'algoritmo è un ricoprimento tramite vertici, poiché ogni arco viene coperto con due vertici se, al momento in cui viene esaminato, questi sono entrambi non inclusi nella soluzione e con almeno un vertice in caso contrario.

D'altra parte, non possiamo garantire che tale soluzione sia di cardinalità minima: considerando il grafo mostrato nella Figura 9.3, la soluzione prodotta dall'algoritmo include tutti e sei i vertici mentre quella di cardinalità minima ne ha solamente tre. Possiamo però mostrare che la soluzione calcolata dal Codice 9.2 include un numero di vertici che è sempre minore oppure uguale al doppio della cardinalità di una soluzione ottima.

A tale scopo, dato un grafo G, notiamo che il sottografo indotto dalla soluzione S calcolata dall'algoritmo con G in ingresso, è formato da $\frac{|S|}{2}$ archi a due a due disgiunti, ovvero senza estremi in comune. Chiaramente, un qualunque ricoprimento di tale sottografo (e quindi di G) deve includere almeno $\frac{|S|}{2}$ vertici: pertanto, $|S|$ è minore oppure uguale al doppio della cardinalità di un qualunque ricoprimento tramite vertici di G e, quindi, della cardinalità minima.

ALVIE: minimo ricoprimento tramite vertici

Osserva, sperimenta e verifica
`VertexCover`



9.4 Opus libri: il problema del commesso viaggiatore

Concludiamo questo capitolo e il libro con un'ultima opera algoritmica, relativa a uno dei problemi di ottimizzazione più analizzati (in tutte le sue varianti) nel campo dell'informatica e della ricerca operativa, ovvero il **problema del commesso viaggiatore**.

```
1  RicoprimentoVertici( A ):
2                              ⟨pre: A è la matrice di adiacenza di un grafo di n nodi⟩
3    FOR (i = 0; i < n; i = i + 1)
4      preso[i] = FALSE;
5    FOR (i = 0; i < n; i = i + 1)
6      FOR (j = 0; j < n; j = j + 1) {
7        IF (A[i][j] == 1 && !preso[i] && !preso[j]) {
8          preso[i] = TRUE;
9          preso[j] = TRUE;
10       }
11     }
12   RETURN preso;
```

Codice 9.2 Algoritmo per il calcolo di un ricoprimento tramite vertici.

Dato un insieme di città e specificato, per ogni coppia di città, la distanza chilometrica per andare dall'una all'altra o viceversa, un commesso viaggiatore si chiede quale sia il modo più breve per visitare tutte le città una e una sola volta, tornando al termine del giro alla città di partenza. Consideriamo, ad esempio, la seguente istanza del problema in cui 9 città olandesi sono analizzate e in cui le distanze chilometriche sono tratte da una nota guida turistica internazionale:

	A	B	D	E	H	L	M	R	U
Amsterdam	0	101	98	121	20	55	213	73	37
Breda		0	30	57	121	72	146	51	73
Dordrecht			0	92	94	45	181	24	61
Eindhoven				0	136	134	86	113	88
Haarlem					0	51	228	70	54
L'Aia						0	223	21	62
Maastricht							0	202	180
Rotterdam								0	57
Utrecht									0

Se il commesso viaggiatore decide di percorrere le città secondo il loro ordine alfabetico, allora percorre un numero di chilometri pari a

$$101 + 30 + 92 + 136 + 51 + 223 + 202 + 57 + 37 = 929$$

Supponiamo, invece, che decida di percorrerle nel seguente ordine: Amsterdam, Haarlem, L'Aia, Rotterdam, Dordrecht, Breda, Maastricht, Eindhoven e Utrecht. In tal caso, il commesso viaggiatore percorre un numero di chilometri pari a

$$20 + 51 + 21 + 24 + 30 + 146 + 86 + 88 + 37 = 503$$

Mediante una ricerca esaustiva di tutte le possibili $9! = 362880$ permutazioni delle nove città, possiamo verificare che quest'ultima è la soluzione migliore possibile. Sfortunatamente, il commesso viaggiatore non ha altra scelta che applicare un algoritmo esaustivo per trovare la soluzione al suo problema, in quanto la sua versione decisionale è un problema NP-completo. Più precisamente, consideriamo il seguente problema di decisione: dati un grafo completo $G = (V, E)$, una funzione p che associa a ogni arco del grafo un numero intero non negativo e un numero intero $k \geqslant 0$, esiste un *tour* del commesso viaggiatore di peso non superiore a k, ovvero un ciclo hamiltoniano in G (Paragrafo 6.1.1) la somma dei cui archi è minore oppure uguale a k?

Per dimostrare che tale problema è NP-completo, consideriamo il problema del circuito hamiltoniano che consiste nel decidere se un grafo qualsiasi include un ciclo hamiltoniano. Utilizzando la tecnica della progettazione di componenti possiamo dimostrare che tale problema è NP-completo. Mostriamo ora che la ricerca di un ciclo hamiltoniano è trasformabile in tempo polinomiale nella versione decisionale del problema del commesso viaggiatore. Dato un grafo $G = (V, E)$, definiamo un grafo completo $G' = (V, E')$ e una funzione p tale che, per ogni arco e in E', $p(e) = 1$ se $e \in E$, altrimenti $p(e) = 2$. Scegliendo $k = |V|$, abbiamo che se esiste un ciclo hamiltoniano in G, allora esiste un tour in G' il cui costo è uguale a k. Viceversa, se non esiste un ciclo hamiltoniano in G, allora ogni tour in G' deve includere almeno un arco il cui peso sia pari a 2, per cui ogni tour ha un costo almeno pari a $k + 1$. In conclusione, il problema del commesso viaggiatore (nella sua forma decisionale) è NP-completo.

Sfortunatamente, possiamo mostrare che il problema di ottimizzazione non ammette neanche un algoritmo efficiente di approssimazione. A tale scopo, consideriamo nuovamente il problema del circuito hamiltoniano e, facendo uso della tecnica detta del **gap**, dimostriamo che se il problema del commesso viaggiatore ammette un algoritmo efficiente di approssimazione, allora il problema del circuito hamiltoniano è risolvibile in tempo polinomiale.

Sia $r > 1$ una costante e sia A un algoritmo polinomiale di r-approssimazione per il problema del commesso viaggiatore. Dato un grafo $G = (V, E)$, definiamo un grafo completo $G' = (V, E')$ e una funzione p tale che, per ogni arco e in E', $p(e) = 1$ se $e \in E$, altrimenti $p(e) = 1 + s|V|$ dove $s > r - 1$. Notiamo che G' ammette un tour del commesso viaggiatore di costo pari a $|V|$ se e solo se G include un circuito hamiltoniano: infatti, un tale tour deve necessariamente usare archi di peso pari a 1, ovvero archi contenuti in E.

Sia T il tour del commesso viaggiatore che viene restituito da A con G' in ingresso. Dimostriamo che T può essere usato per decidere se G ammette un ciclo hamiltoniano, distinguendo i seguenti due casi.

1. Il costo di T è uguale a $|V|$, per cui T è un tour ottimo. Quindi, G ammette un ciclo hamiltoniano.

```
1  CommessoViaggiatore( P ):
2            ⟨pre: P è la matrice di adiacenza e dei pesi di un grafo G completo di n nodi⟩
3    mst = Jarník-Prim( P );
4    cicloEuleriano = Euler( mst );
5    FOR (i = 0; i < n; i = i +1)
6      visitato[i] = -1;
7    posizione = 0;
8    FOR (i = 0; i < 2 × n - 1; i = i +1) {
9      IF (visitato[ cicloEuleriano[i] ] < 0) {
10       visitato[ cicloEuleriano[i] ] = posizione;
11       posizione = posizione + 1;
12     }
13   }
14   RETURN visitato;
```

Codice 9.3 Algoritmo per il calcolo di un tour approssimato del commesso viaggiatore.

2. Il costo di T è maggiore di $|V|$, per cui il suo costo deve essere almeno pari a $|V| - 1 + 1 + s|V| = (1 + s)|V| > r|V|$. In questo caso, il tour ottimo non può avere costo pari a $|V|$, in quanto altrimenti il costo di T è maggiore di r volte il costo ottimo, contraddicendo il fatto che A è un algoritmo di r-approssimazione: quindi, non esiste un ciclo hamiltoniano in G.

In conclusione, applicando l'algoritmo A al grafo G' e verificando se la soluzione restituita da A ha un costo pari oppure maggiore al numero dei vertici, possiamo decidere in tempo polinomiale se G ammette un circuito hamiltoniano, contraddicendo il fatto che il problema del circuito hamiltoniano è NP-completo.

9.4.1 Problema del commesso viaggiatore su istanze metriche

Sebbene il problema del commesso viaggiatore non sia, in generale, risolvibile in modo approssimato mediante un algoritmo polinomiale, possiamo mostrare che tale problema, ristretto al caso in cui la funzione che specifica la distanza tra due città soddisfi la disuguaglianza triangolare, ammette un algoritmo di 2-approssimazione.

Un'istanza del problema del commesso viaggiatore soddisfa la **disuguaglianza triangolare** se, per ogni tripla di vertici i, j e k, $p(i,j) \leqslant p(i,k) + p(k,j)$: intuitivamente, ciò vuol dire che andare in modo diretto da una città i a una città j non può essere più costoso che andare da i a j passando prima per un'altra città k. L'esempio delle città olandesi visto in precedenza soddisfa la disuguaglianza triangolare, anche se tale disuguaglianza non sempre è soddisfatta quando si tratta di distanze stradali.

La disuguaglianza triangolare è invece sempre soddisfatta se i vertici del grafo rappresentano punti del piano euclideo e le distanze tra due vertici corrispondono alle loro distanze nel piano, in quanto in ogni triangolo la lunghezza di un lato è sempre minore della somma delle lunghezze degli altri due lati. Inoltre, la disuguaglianza è soddisfatta nel caso in cui il grafo completo G sia ottenuto nel modo seguente, a partire da un grafo G′ connesso non necessariamente completo: G ha gli stessi vertici di G′ e la distanza tra due suoi vertici è uguale alla lunghezza del cammino minimo tra i corrispondenti vertici di G′. Per questo motivo, la risoluzione del problema del commesso viaggiatore, ristretto al caso di istanze che soddisfano la disuguaglianza triangolare, è un problema di per sé interessante che sorge abbastanza naturalmente in diverse aree applicative.

La versione decisionale di tale problema è NP-completo, in quanto la trasformazione che abbiamo mostrato in precedenza a partire dal problema del circuito hamiltoniano genera istanze che soddisfano la disuguaglianza triangolare: se i pesi degli archi sono solo 1 e 2, ovviamente tale disuguaglianza è sempre soddisfatta. Mostriamo ora un algoritmo polinomiale di 2-approssimazione per il problema del commesso viaggiatore, ristretto al caso di istanze che soddisfano la disuguaglianza triangolare.

L'idea alla base dell'algoritmo è che la somma dei pesi degli archi di un minimo albero ricoprente R di un grafo completo G costituisce un limite inferiore al costo di un tour ottimo. Infatti, cancellando un arco di un qualsiasi tour T, otteniamo un cammino hamiltoniano e, quindi, un albero ricoprente: la somma dei pesi degli archi di questo cammino deve essere, per definizione, non inferiore a quella dei pesi degli archi di R. Quindi, il costo di T (che include anche il peso dell'arco cancellato) è certamente non inferiore alla somma dei pesi degli archi di R.

L'algoritmo (realizzato nel Codice 9.3) costruisce un minimo albero ricoprente di G (riga 3) e, in modo analogo a quanto fatto nel caso del problema del minimo antenato comune (Paragrafo 4.2.1), visita tale albero in profondità creando un ciclo euleriano (riga 4). Ricordiamo che, in tale ciclo, ogni vertice dell'albero può essere visitato più di una volta, ma che ogni arco dell'albero viene percorso esattamente due volte: una andando dal padre verso il figlio e una tornando dal figlio al padre. Pertanto, il ciclo euleriano è formato da $2n - 1$ elementi, di cui il primo e l'ultimo coincidono, come mostrato nella Figura 9.5.

A questo punto, l'algoritmo percorre l'intero ciclo euleriano (righe 8–13): ogni qualvolta incontra un vertice non ancora visitato (riga 9), lo inserisce nel tour del commesso viaggiatore nella posizione attuale (riga 10) e aggiorna quest'ultima (riga 11). In altre parole, il codice costruisce il tour del commesso viaggiatore viaggiando attraverso gli archi che uniscono le prime occorrenze di ciascun nodo nel ciclo euleriano.

L'algoritmo restituisce, infine, l'array delle posizioni dei vertici all'interno del tour (riga 14), supponendo implicitamente che il vertice in ultima posizione (ovvero, posizione $n - 1$) deve essere connesso a quello in prima posizione (ovvero, posizione 0).

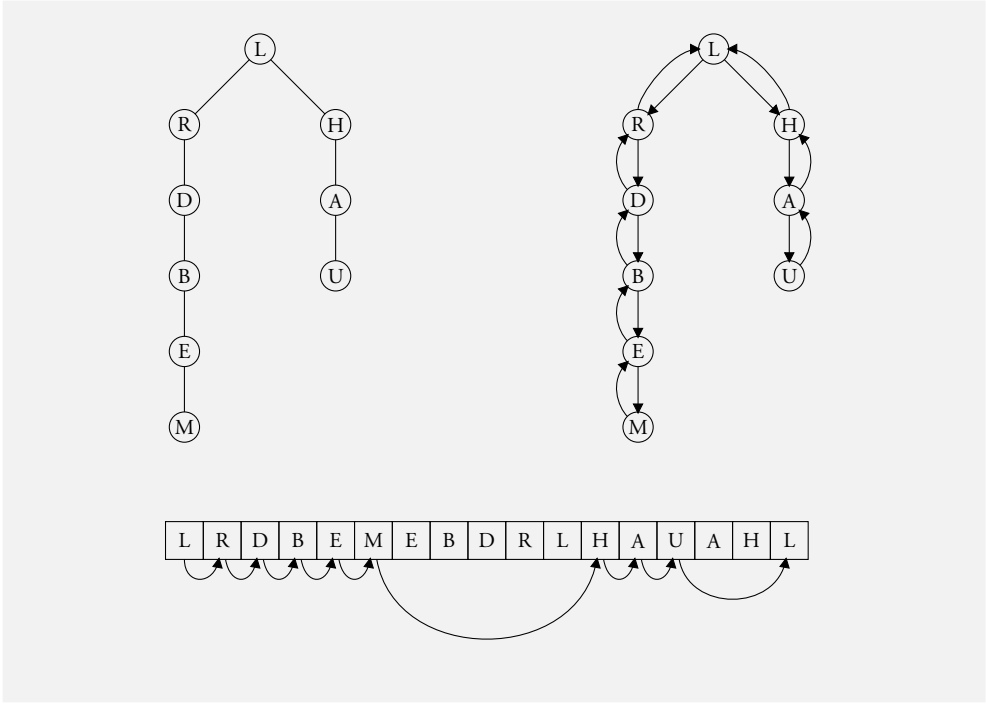


Figura 9.5 Un minimo albero ricoprente del grafo delle città olandesi, il corrispondente ciclo euleriano e il tour del commesso viaggiatore.

Poiché il calcolo del minimo albero ricoprente e del corrispondente ciclo euleriano possono essere realizzati in $O(n^2 \log n)$ e poiché il resto del codice richiede $O(n)$ tempo, abbiamo che l'algoritmo appena descritto è polinomiale. Per dimostrare che è anche un algoritmo di 2-approssimazione, notiamo anzitutto che la somma dei pesi degli archi inclusi nel ciclo euleriano è al più due volte la somma dei pesi degli archi presenti nel minimo albero ricoprente.

Inoltre, in base alla disuguaglianza triangolare, i salti che vengono eseguiti per costruire il tour non possono essere più costosi della parte di ciclo euleriano su cui essi passano sopra: quindi, il costo complessivo del tour è al più pari alla somma dei pesi degli archi inclusi nel ciclo euleriano, la quale a sua volta è al più due volte la somma dei pesi degli archi presenti nel minimo albero ricoprente che, come abbiamo osservato in precedenza, non è superiore al costo del tour ottimale.

Più precisamente, sia $i_0, i_1, \ldots, i_{m-1}$ la sequenza dei nodi inclusi nel ciclo euleriano, dove $m = 2n - 1$. Inoltre, siano $j_0, j_1, \ldots, j_{n-1}$ le posizioni delle prime occorrenze degli

n vertici di G all'interno del ciclo euleriano: quindi, $i_{j_0} = 0$, $i_{j_h} \neq i_{j_k}$ per $0 \leqslant h < k < n$ e $i_{j_{n-1}} \neq i_{j_0}$. Infine, poniamo $j_n = m - 1$. Il tour del commesso viaggiatore costruito dal Codice 9.3 include gli archi $(i_{j_h}, i_{j_{h+1}})$ per $0 \leqslant h < n$. In base alla disuguaglianza triangolare, abbiamo che

$$p(i_{j_h}, i_{j_{h+1}}) \leqslant p(i_{j_h}, i_{j_h} + 1) + p(i_{j_h} + 1, i_{j_h} + 2) + \cdots + p(i_{j_{h+1}} - 1, i_{j_{h+1}})$$

per $0 \leqslant h < n$: quindi, il costo del tour calcolato dall'algoritmo è limitato superiormente dalla somma dei pesi degli archi inclusi nel ciclo euleriano. In conclusione, tale tour ha un costo pari al più a due volte il costo minimo.

Per concludere, osserviamo che l'algoritmo di approssimazione realizzato dal Codice 9.3 non è il migliore possibile. In effetti, con un opportuno accorgimento nello scegliere gli archi del minimo albero ricoprente da duplicare, possiamo modificare tale algoritmo ottenendone uno di 1,5-approssimazione. Inoltre, nel caso di istanze formate da punti sul piano euclideo, possiamo dimostrare che, per ogni $r > 1$, esiste un algoritmo polinomiale di r-approssimazione: in altre parole, il problema del commesso viaggiatore sul piano può essere approssimato tanto bene quanto vogliamo (ovviamente al prezzo di una complessità temporale che, pur mantenendosi polinomiale, cresce al diminuire di r).

ALVIE: problema del commesso viaggiatore

Osserva, sperimenta e verifica
`TravelingSalesman`

9.4.2 Paradigma della ricerca locale

Un algoritmo per la risoluzione di un problema di ottimizzazione basato sul **paradigma della ricerca locale** opera nel modo seguente: a partire da una soluzione iniziale del problema, esplora un insieme di soluzioni "vicine" a quella corrente e si sposta in una soluzione che è migliore di quella corrente, fino a quando non giunge a una che non ha nessuna soluzione vicina migliore. Pertanto, il comportamento di un tale algoritmo dipende dalla nozione di vicinato di una soluzione (solitamente generato applicando operazioni di cambiamento locale alla soluzione corrente), dalla soluzione iniziale (che può essere calcolata mediante un altro algoritmo) e dalla strategia di selezione delle soluzioni (ad esempio, scegliendo la prima soluzione vicina migliore di quella corrente oppure selezionando la migliore tra tutte quelle vicine alla corrente).

Non esistono regole generali per decidere quali siano le regole di comportamento migliori: per questo, ci limitiamo in questo paragrafo finale a descrivere due algoritmi

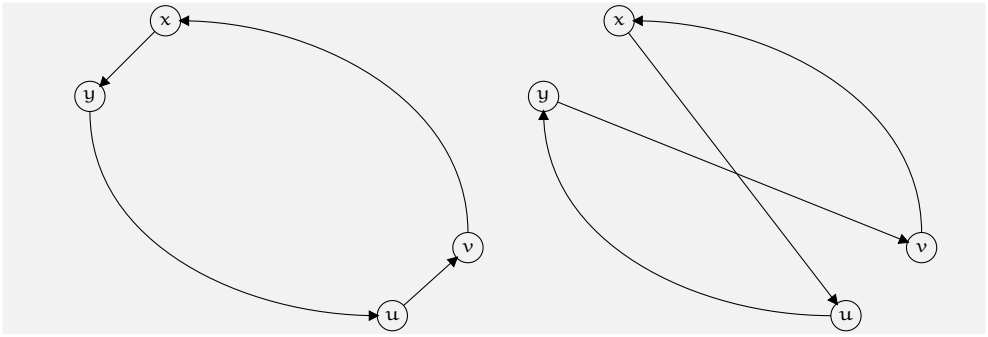


Figura 9.6 L'operazione di modifica di un tour realizzata dall'algoritmo 2-opt.

basati sul paradigma della ricerca locale per la risoluzione (non ottima) del problema del commesso viaggiatore. Notiamo sin d'ora che non siamo praticamente in grado di formulare nessuna affermazione (non banale) relativamente alle prestazioni di questi algoritmi né in termini di complessità temporale né in termini di qualità della soluzione ottenuta. Tuttavia, questo tipo di strategie (dette anche **euristiche**) risultano nella pratica estremamente valide e, per questo, molto utilizzate.

Entrambi gli algoritmi che descriviamo fanno riferimento a operazioni locali di cambiamento: essi tuttavia differiscono tra di loro per quello che riguarda la lunghezza massima della sequenza di tali operazioni. In particolare, i due algoritmi modificano la soluzione corrente selezionando un numero fissato di archi e sostituendoli con un altro insieme di archi (della stessa cardinalità) in modo da ottenere un nuovo tour.

Il primo algoritmo, detto **2-opt**, opera nel modo seguente: dato un tour T del commesso viaggiatore, il suo vicinato è costituito da tutti i tour che possono essere ottenuti cancellando due archi (x, y) e (u, v) di T e sostituendoli con due nuovi archi (x, u) e (y, v) in modo da ottenere un tour differente T' (notiamo che questo equivale a invertire la percorrenza di una parte del tour T, come mostrato nella Figura 9.6).

Se il nuovo tour T' ha un costo minore di quello di T, allora T' diviene la soluzione corrente, altrimenti l'algoritmo procede con una diversa coppia di archi: il procedimento ha termine nel momento in cui giungiamo a un tour che non può essere migliorato.

Il Codice 9.4 realizza l'algoritmo 2-opt. Dopo aver inizializzato il tour iniziale e il relativo costo, visitando i vertici nell'ordine in cui appaiono nel grafo (righe 3–5), il codice esamina tutte le coppie di archi $(i, i+1)$ e $(j, j+1)$ con $0 \leqslant i < j-1 < n-1$ (righe 6–26) e per ognuna di esse genera il nuovo tour operando la sostituzione precedentemente descritta (righe 9–15). Quindi, il codice calcola il costo del nuovo tour (righe 16–19): se tale costo è minore del costo precedente, allora il tour corrente viene

```
1  2-Opt( P ):
2            ⟨pre: P è la matrice di adiacenza e dei pesi di un grafo G completo di n nodi⟩
3    costo = 0;
4    FOR (i = 0; i < n; i = i +1)
5      { tour[i] = i; costo = costo + P[i][(i+1) % n]; }
6    FOR (i = 0; i < n; i = i +1) {
7      FOR (j = i+2; j < n-1; j = j + 1) {
8        FOR (h = 0; h <= i; h = h +1)
9          nuovo[h] = tour[h];
10       nuovo[i+1] = tour[j];
11       FOR (h = 1; h < j-i; h = h +1)
12         nuovo[i+1+h] = tour[j-h];
13       nuovo[j+1] = tour[j+1];
14       FOR (h = j+2; h < n; h = h +1)
15         nuovo[h] = tour[h];
16       nuovoCosto = 0;
17       FOR (h = 0; h < n; h = h +1) {
18         nuovoCosto = nuovoCosto + P[nuovo[h]][nuovo[(h+1) % n]];
19       }
20       IF (nuovoCosto < costo) {
21         { costo = nuovoCosto; i = 0; }
22         FOR (h = 0; h < n; h = h+1)
23           tour[h] = nuovoTour[h];
24       }
25     }
26   }
27   RETURN tour;
```

Codice 9.4 Algoritmo 2-opt.

aggiornato e il ciclo `for` più esterno viene fatto ripartire dall'inizio (righe 21 e 23). Se non troviamo nessun tour migliore di quello attuale, allora il codice restituisce il tour attuale come soluzione del problema (riga 27).

Poiché, ogni qualvolta viene trovato un tour di costo minore, tale costo diminuisce almeno di un'unità, abbiamo che il numero totale di iterazioni del ciclo `for` più esterno è limitato dal costo del tour iniziale: pertanto, il Codice 9.4 termina in tempo $O(n^3C)$ dove C indica il costo del tour iniziale.

Il secondo algoritmo di risoluzione del problema del commesso viaggiatore basato sul paradigma della ricerca locale è detto **3-opt**, in quanto opera in modo analogo a 2-opt, ma considera come vicinato di un tour T tutti i tour che possono essere ottenuti scambiando tre archi di T.

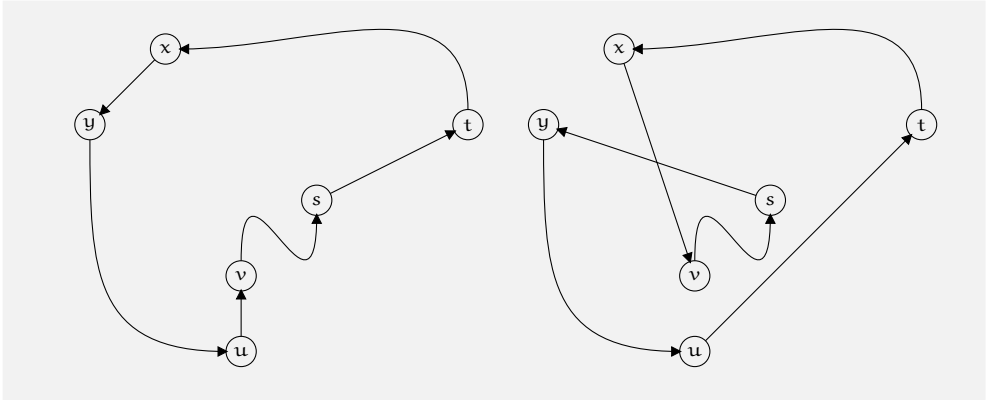


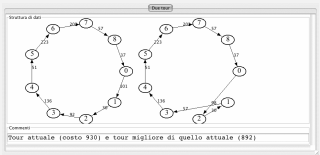
Figura 9.7 L'operazione di modifica di un tour realizzata dall'algoritmo 3-opt.

In particolare, se $i_0, i_1, \ldots, i_{n-1}$ è il tour corrente, l'algoritmo 3-opt sceglie tre indici j_0, j_1 e j_2 con $j_0 < j_1 - 1 < j_2 - 2$ e sostituisce i tre archi (i_{j_0}, i_{j_0+1}), (i_{j_1}, i_{j_1+1}) e (i_{j_2}, i_{j_2+1}) con gli archi (i_{j_0}, i_{j_1+1}), (i_{j_2}, i_{j_0+1}) e (i_{j_1}, i_{j_2+1}) (notiamo che in questo caso non abbiamo bisogno di invertire una parte del tour corrente, come mostrato nella Figura 9.7).

Possiamo verificare sperimentalmente che 3-opt ha delle prestazioni migliori di 2-opt per quello che riguarda la qualità della soluzione, ma richiede un tempo di calcolo superiore. Sebbene, in linea di principio, possiamo pensare di generalizzare i due algoritmi appena esposti definendo una strategia k-opt per qualunque $k \geqslant 2$, il miglioramento che si ottiene nella qualità della soluzione calcolata nel passare da 3-opt a 4-opt non sembra giustificare il significativo peggioramento delle prestazioni in termini di tempo di esecuzione.

ALVIE: algoritmo 2-opt

Osserva, sperimenta e verifica
`TwoOpt`



Gli algoritmi 2-opt e 3-opt risultano efficaci nella pratica, ma possono avere prestazioni molto scarse nel caso pessimo (anche in dipendenza della scelta del tour iniziale): in effetti, non conosciamo alcun limite superiore all'approssimazione raggiunta dalla soluzione calcolata da questi algoritmi nel caso generale.