1. Tecniche risolutive per problemi intrattabili

Se si deve risolvere un particolare problema, e si riesce a dimostrare la sua NP-completezza, che cosa si può fare? In generale, bisogna adottare un approccio diverso, che può essere riassunto dalla massima seguente:

"Chi si accontenta gode".

Non si può avere tutto dalla vita; bisogna rinunciare a qualcosa. Ad esempio,

- *generalità*: si può vedere se esiste qualche valore particolare dell'input che renda il problema risolubile in tempo polinomiale, utilizzando un algoritmo *pseudopolinomiale*;
- *ottimalità*: si può cercare di individuare un algoritmo di *approssimazione* che non restituisca necessariamente la soluzione ottima, ma che garantisca di ottenerne una non troppo "distante" da essa;
- *efficienza*: si possono progettare algoritmi esponenziali di tipo branch-&-bound, che cercano di evitare di enumerare tutte le soluzioni con una accurata potatura;
- *formalità*: si possono ricercare algoritmi euristici, di solito basati su tecniche *greedy* o di ricerca locale, che siano particolarmente facili da programmare, abbiano se possibile complessità polinomiale, e forniscano sperimentalmente risultati buoni – anche se non vi è alcuna dimostrazione formale che questo avvenga sempre.

1.1 Algoritmi pseudopolinomiali

Si consideri il problema SOMMA DI SOTTOINSIEME, del quale è stata menzionata l'NP-completezza nel Capitolo 18:

SOMMA DI SOTTOINSIEME. *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 $\sum_{i \in S} a_i = k$?*

Si supponga di memorizzare l'insieme A in un vettore $A[1 \ldots n]$ di interi e si introduca un vettore booleano $M[0 \ldots k]$ di $k+1$ elementi. Il problema può essere facilmente risolto dalla funzione subsetSum(), basata sulla programmazione dinamica.

Il problema viene risolto via via per $i = 1, 2, \ldots, n$. All'iterazione i-esima, il valore booleano di $M[j]$ indica se esiste un sottoinsieme di $\{a_1, \ldots, a_i\}$ la cui somma dà esattamente j. Inizialmente

```
boolean subsetSum(int[] A, int n, int k)
    boolean[] M = new boolean[0...k]
    for j = 1 to k do M[j] = false
    M[0] = true
    for i = 1 to n do
        for j = k downto A[i] do
            M[j] = M[j - A[i]] or M[j]
    return M[k]
```

$M[0]$ è posto a **true** perché l'insieme vuoto dà somma 0. All'iterazione i-esima, viene considerata la possibilità di aggiungere a_i. L'assegnamento "$M[j] = M[j-A[i]]$ **or** $M[j]$" prevede i due casi possibili in cui a_i sia aggiunto ad un sottoinsieme di $\{a_1, \ldots, a_{i-1}\}$ che dà somma $j - a_i$, oppure non sia aggiunto ad un sottoinsieme di $\{a_1, \ldots, a_{i-1}\}$ che dà somma j. Nel primo caso, $M[j]$ è uguale ad $M[j-A[i]]$, mentre nel secondo caso $M[j]$ mantiene il suo vecchio valore. Effettuando l'operazione **or**, $M[j]$ indica correttamente se c'è un sottoinsieme di $\{a_1, \ldots, a_i\}$ con somma j. Si noti che il ciclo **for** più interno procede per valori decrescenti di j, onde evitare di considerare più occorrenze di a_i nel medesimo sottoinsieme, come invece potrebbe accadere procedendo per valori crescenti di j. La funzione restituisce il valore **true** se e solo se $M[k] =$ **true**, ovvero se esiste un sottoinsieme di $\{a_1, \ldots, a_n\}$ la cui somma dà k.

Esempio (Somma di sottoinsieme) Siano $n = 3$, $k = 24$, $a_1 = 7$, $a_2 = 11$, $a_3 = 13$. La seguente tabella illustra il contenuto del vettore M per $i = 0, 1, 2, 3$:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
		1																							
$i = 1$		1						1																	
$i = 2$		1						1				1							1						
$i = 3$		1						1				1		1					1		1				1

■

La complessità della funzione subsetSum() è ovviamente $O(nk)$, che non è polinomiale nella dimensione del problema. Infatti, assumendo $a_i \leq k$ per $1 \leq i \leq n$, poiché altrimenti a_i può essere eliminato, la dimensione d dei dati di ingresso è $O(n \log k)$, ma non $O(nk)$. Se k è limitata da un polinomio in n, cioè k è $O(n^c)$ con c costante, allora la funzione subsetSum() ha complessità polinomiale, poiché nk risulta essere $O(n^{c+1})$. Se invece k è superpolinomiale in n, allora la complessità della funzione risulta essere anch'essa superpolinomiale. Infatti, se k è $O(2^n)$, allora nk è $O(n2^n)$.

In altri termini, la complessità di subsetSum() non dipende soltanto dalla cardinalità n del sottoinsieme in ingresso, ma anche dai valori degli elementi del sottoinsieme. Tale complessità risulta polinomiale oppure no a seconda che i valori degli elementi in ingresso siano "piccoli" oppure "grandi" rispetto al loro numero. Una situazione analoga si è incontrata per la procedura countingSort() [cfr. Capitolo 2], che permette di ordinare n elementi in tempo $O(n)$ qualora i valori di tali elementi siano "piccoli", cioè limitati anch'essi da $O(n)$, e per il problema dello ZAINO [cfr. Capitolo 13].

Ci si potrebbe chiedere sotto quali condizioni un problema NP-completo può essere risolto con un algoritmo polinomiale per valori "piccoli" dei dati in ingresso.

Dato un problema decisionale P, con insieme di dati di ingresso I, distinguiamo tra la "dimen-

sione" d di I, cioè la lunghezza della stringa che codifica I, ed il "più grande numero" intero # che appare in I.

Esempio (dimensione d e numero #)
- SOMMA DI SOTTOINSIEME:
  $I = \{n, k, a_1, \ldots, a_n\}$, $\# = \max\{n, k, \max_i\{a_i\}\}$, $d = O(n \log \#)$;
- COMMESSO VIAGGIATORE:
  $I = \{n, k, [d_{ij}]\}$, $\# = \max\{n, k, \max_{ij}\{d_{ij}\}\}$, $d = O(n^2 \log \#)$;
- CRICCA:
  $I = \{n, m, k, G = (V, E)\}$, $\# = \max\{n, m, k\}$, $d = O(n + m + \log \#)$. ■

Sia P_p il problema P ristretto a quei dati d'ingresso per i quali # è limitato superiormente da $p(d)$, con p funzione polinomiale in d. P è *fortemente NP-completo* se P_p è NP-completo.

Esempio (NP-completezza forte)
- La SOMMA DI SOTTOINSIEME non è fortemente NP-completa; infatti, se $\# = n$ il problema è risolubile in tempo polinomiale, mentre se $\# = k$, allora # non può in generale essere limitato superiormente da alcun polinomio nella dimensione $d = O(n \log k)$ (ovviamente se $\# = a_i > k$ allora a_i non può far parte di alcuna soluzione e può essere eliminato);
- Il COMMESSO VIAGGIATORE è fortemente NP-completo, perché resta NP-completo anche quando $d_{ij} \in \{1, 2\}$, $1 \leq i, j \leq n$, e $k = n$ [si veda la dimostrazione del Teorema 18.6]; in tal caso, $\# = n$, che è ovviamente limitato superiormente da un polinomio nella dimensione $d = O(n^2 \log n)$;
- Anche la CRICCA è fortemente NP-completa, poiché k può essere assunto minore o uguale ad n (se $k > n$, la risposta del problema è sempre NO) e quindi # è limitato superiormente da un polinomio in $d = O(n + m + \log(m + n))$. ■

In pratica, sono "fortemente" NP-completi tutti quei problemi che possono essere dimostrati NP-completi con riduzioni che non usano numeri interi esponenzialmente grandi, mentre sono "debolmente" NP-completi quei problemi la cui dimostrazione di intrattabilità dipende pesantemente da riduzioni che usano numeri interi esponenzialmente grandi. Per esempio, le riduzioni illustrate per il DOMINO LIMITATO, la CRICCA, la SODDISFATTIBILITÀ, la PROGRAMMAZIONE LINEARE 0/1, l'INSIEME INDIPENDENTE, il COMMESSO VIAGGIATORE richiedono numeri interi che sono polinomialmente limitati dalla dimensione d dell'input (in alcuni casi sono addirittura limitati da costanti). Pertanto, tutti questi problemi sono fortemente NP-completi. Viceversa, ciò non è vero per la SOMMA DI SOTTOINSIEME, la PARTIZIONE e lo ZAINO, che sono invece debolmente NP-completi.

Formalmente, un algoritmo che risolve un certo problema P, per qualsiasi dato I d'ingresso, in tempo $p(\#, d)$, con p funzione polinomiale in # e d, ha complessità *pseudopolinomiale*. L'algoritmo visto per risolvere il problema SOMMA DI SOTTOINSIEME ha complessità pseudopolinomiale. Vale il seguente risultato:

Teorema 1.1 Nessun problema fortemente NP-completo può essere risolto da un algoritmo pseudopolinomiale, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

Dimostrazione. Se P è fortemente NP-completo, allora il problema ristretto P_p è NP-completo per qualche polinomio p. Se esistesse un algoritmo pseudopolinomiale A per P, allora A risolverebbe P_p in tempo polinomiale, che è assurdo, a meno che non sia $\mathbb{P} = \mathbb{NP}$. ■

Il più famoso problema fortemente NP-completo, in cui numeri interi hanno un ruolo rilevante, è la cosiddetta 3-PARTIZIONE, che generalizza appunto il problema della PARTIZIONE:

3-PARTIZIONE (3-PARTITION). *Dati $3n$ interi $\{a_1,\ldots,a_{3n}\}$, esiste una partizione in n triple $T_1,\ldots,T_n$, tale che la somma dei tre elementi di ogni T_j è la stessa, per $1 \leq j \leq n$?*

Al contrario della PARTIZIONE, della SOMMA DI SOTTOINSIEME e dello ZAINO, che sono tutti risolubili in tempo pseudopolinomiale, la 3-PARTIZIONE non lo è.

Per dimostrare l'NP-completezza forte di un problema P, non è sempre necessario dimostrare l'NP-completezza di un sottoproblema P_p. Infatti, è sufficiente provare l'esistenza di un altro problema Q, di cui si conosce già l'NP-completezza forte, che si può ridurre a P con una trasformazione che richiede tempo pseudopolinomiale. Tale trasformazione deve inoltre garantire che $\#_P$, il più grande intero introdotto nei dati di P, sia limitato superiormente da una funzione $p(d_Q,\#_Q)$ polinomiale nella dimensione d_Q dell'input di Q e nel più grande intero $\#_Q$ che appare in tale input.

1.2 Algoritmi di approssimazione

Molti problemi di ottimizzazione di rilevante interesse pratico sono NP-ardui, poiché gli stessi problemi in forma decisionale sono NP-completi. Per tali problemi, c'è poca speranza di progettare un algoritmo che trovi la soluzione ottima in tempo polinomiale. In molti casi, però, il fatto di non poter ottenere la soluzione ottima non è poi così drammatico, e ci si può accontentare di algoritmi che, in tempo pNolinomiale, trovino una soluzione ammissibile del problema che sia abbastanza "vicina" a quella ottima. Se è possibile dimostrare che la soluzione ammissibile fornita dall'algoritmo si discosta, nel caso pessimo o in quello medio, dalla soluzione ottima per uno scarto (o errore) relativo limitato per tutti i possibili dati di ingresso del problema, allora si ha un algoritmo di "approssimazione".

In questa sezione, vedremo algoritmi di approssimazione assoluta per i problemi BIN PACKING e COMMESSO VIAGGIATORE CON DISUGULIANZE TRIANGOLARI, dove lo scarto relativo è limitato da una costante che dipende solo dall'algoritmo. Vedremo inoltre un algoritmo di ε-approssimazione per la SOMMA DI SOTTOINSIEME, dove lo scarto relativo ε viene fornito in ingresso all'algoritmo stesso e può essere reso piccolo a piacere, al prezzo di un sempre maggiore sforzo computazionale. Vedremo infine come non tutti i problemi NP-ardui ammettano algoritmi di approssimazione.

1.2.1 Approssimazione assoluta

Dato un problema di ottimizzazione P, con funzione costo non negativa c, un algoritmo di approssimazione assoluta fornisce una soluzione ammissibile x' il cui costo si discosta da quello della soluzione ottima x^* per uno scarto relativo garantito e limitato da una costante. Formalmente, occorre che

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq e$$

per una qualche costante e non negativa che dipende dall'algoritmo e per tutti i possibili dati d'ingresso. In altri termini, l'algoritmo garantisce che la percentuale di scostamento dalla soluzione ottima non superi una certa costante. Tale scostamento vale per la medesima costante e per tutti gli input! Ad esempio, se $e = 1$, allora lo scostamento dall'ottimo non supera il 100%, cioè $c(x')$ è al più il doppio di $c(x^*)$, qualora il problema sia di minimizzazione, oppure almeno la metà, nel caso di massimizzazione.

Per iniziare, consideriamo un problema NP-arduo molto noto:

BIN PACKING. *Dato un insieme $A = \{a_1,\ldots,a_n\}$ di interi positivi (i "volumi" di n "oggetti") ed un intero positivo k (la "capacità" di una "scatoletta"), si vuole trovare una partizione di $\{1,\ldots,n\}$ nel minimo numero di sottoinsiemi disgiunti (le "scatolette") tali che $\sum_{i\in S} a_i \leq k$ per ogni insieme S della partizione.*

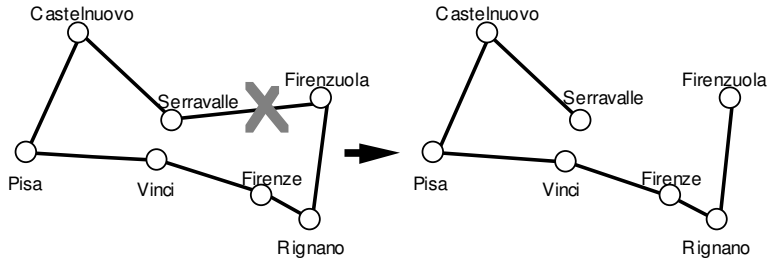

Figura 1.1: Cancellando un arco da un circuito Hamiltoniano si ottiene una catena Hamiltoniana che è anche un albero di copertura.

Si consideri l'algoritmo FF ("First-Fit") in cui gli oggetti sono considerati in un ordine qualsiasi e ciascun oggetto è assegnato alla prima scatoletta che lo può contenere, e dimostriamo che FF è un algoritmo di approssimazione assoluta.

Assumiamo che sia $a_i \leq k$ per ciascun i. Se il numero N di scatolette usate da FF è maggiore di uno, allora tale numero deve essere minore di $\lceil \frac{2}{k} \sum_{i=1}^{n} a_i \rceil$. Questa proprietà deriva dal fatto che non ci possono essere due scatolette riempite al più per metà. Infatti se ci fossero allora il primo oggetto inserito nella scatoletta di indice maggiore avrebbe potuto essere inserito nella scatoletta di indice minore e l'algoritmo FF non l'avrebbe inserito in un'altra scatoletta. D'altro canto, il minimo numero di scatolette N^* non può essere inferiore a $\lceil \frac{1}{k} \sum_{i=1}^{n} a_i \rceil$. Pertanto, $N < 2N^*$ ed FF non utilizza più del doppio del numero ottimo di scatolette. È interessante rimarcare che è possibile dimostrare per un'altra via, molto più complicata, un risultato migliore per FF, e cioè che $N < \frac{17}{10}N^* + 2$. Un'approssimazione ancora migliore si ottiene con una variante di FF, detta FFD ("First-Fit-Decreasing") dove gli oggetti sono considerati secondo l'ordine non crescente $a_1 \geq \cdots \geq a_n$. Per FFD è stato dimostrato che $N < \frac{11}{9}N^* + 4$.

Vediamo ora un esempio più complesso, ovvero un algoritmo di approssimazione assoluta per una variante semplificata del problema del COMMESSO VIAGGIATORE:

COMMESSO VIAGGIATORE CON DISUGUAGLIANZE TRIANGOLARI (ΔTSP). *Date n città e le distanze d_{ij} tra esse, tali che $d_{ij} \leq d_{ik} + d_{kj}$ per $1 \leq i, j, k \leq n$, trovare un percorso che, partendo da una qualsiasi città, attraversi ogni città esattamente una volta e ritorni alla città di partenza, in modo che la distanza complessiva percorsa sia minima.*

La versione decisionale di ΔTSP è NP-completa. Infatti, nella prova di NP-completezza dell'usuale COMMESSO VIAGGIATORE (Teorema 18.6) sono state usate distanze $d_{ij} \in \{1,2\}$, per $1 \leq i, j \leq n$, che ovviamente verificano la proprietà di triangolarità.

Per semplicità, ΔTSP può essere reinterpretato come il problema di trovare un circuito Hamiltoniano di costo minimo in un grafo K_n di n nodi, non orientato, pesato e completo, cioè formato da una cricca di n nodi.

Come mostrato nella Figura 1.1, un qualsiasi circuito Hamiltoniano di K_n in cui si cancelli un arco è un particolare albero di copertura per esso e quindi un minimo albero di copertura per K_n non può avere un costo superiore alla soluzione ottima di ΔTSP.

Se si individua un minimo albero di copertura e se ne percorrono gli archi due volte, la prima volta in un senso e l'altra nel senso opposto, si può visitare ciascuna città almeno una volta e la distanza complessiva percorsa non supera il doppio della distanza ottima di ΔTSP. Visitando le città in modo da "saltare" una città già visitata, si ottiene, per la triangolarità delle distanze, un circuito Hamiltoniano x' il cui costo $c(x')$ non supera il doppio del costo $c(x^*)$ del circuito Hamiltoniano minimo x^* per ΔTSP. Un esempio è fornito nella Figura 1.2.

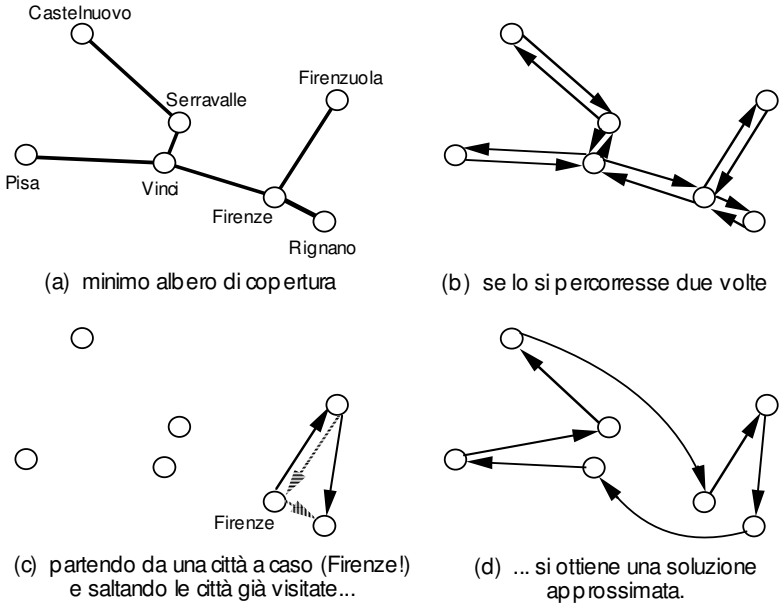

Figura 1.2: Algoritmo di approssimazione assoluta per ΔTSP.

Questo algoritmo di approssimazione richiede $O(n^2 \log n)$ tempo, poiché il minimo albero di copertura può essere trovato con l'algoritmo di Kruskal [cfr. Capitolo 14] e la soluzione x' si può ottenere considerando l'albero ordinato in modo arbitrario e poi visitandone i nodi in ordine anticipato [cfr. Capitolo 5].

L'algoritmo fornisce una soluzione in cui la distanza complessiva percorsa non supera del 100% quella minima. Infatti, si ha:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \frac{|2c(x^*) - c(x^*)|}{|c(x^*)|} = 1.$$

È possibile individuare dati di ingresso particolarmente "perversi" per i quali l'algoritmo produce soluzioni che raggiungono lo scarto pessimo del 100%. Per esempio, si considerino $2n$ città che si trovino sui vertici di due poligoni regolari concentrici, ciascuno di n lati, con distanze pari alle distanze Euclidee tra tali vertici. Un esempio per $n = 6$ è mostrato nella Figura 1.3(a), dove r ed $r + d$ sono i raggi dei cerchi circoscritti, rispettivamente, al poligono interno e a quello esterno. Il minimo albero di copertura è strutturato come mostrato nella Figura 1.3(b) e percorrendo tale albero due volte, si ottiene il percorso illustrato nella Figura 1.3(c). L'algoritmo di approssimazione può trovare, nel caso pessimo, la soluzione x' mostrata nella Figura 1.3(d), che è assai diversa da quella ottima x^* della Figura 1.3(e). Se L ed L' sono i lati, rispettivamente, del poligono interno e di quello esterno, allora:

$$c(x') = (n-1)(L+L') + 2d \quad \wedge \quad c(x^*) = n(L+L')/2 + nd.$$

Ricordiamo che, se R è il raggio del cerchio circoscritto ad un poligono regolare di n lati, allora la lunghezza di ogni lato è $2R\sin(\pi/n)$. Assumendo che il poligono interno abbia raggio $r = 1$ ed il

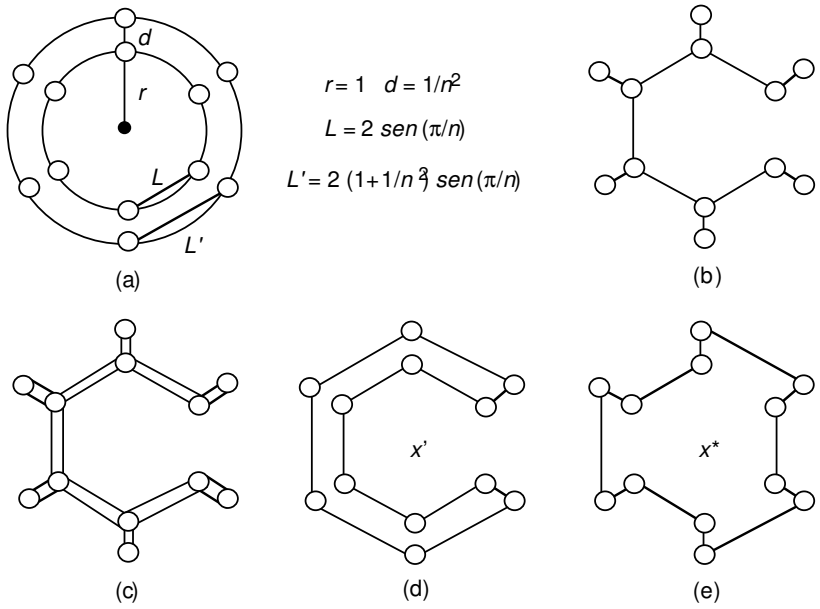

Figura 1.3: Caso pessimo per ΔTSP. (a) Le $2n$ città. (b) Il minimo albero di copertura. (c) L'albero percorso due volte. (d) Soluzione approssimata x', avente costo $c(x') = (n-1)(L+L')+2d$. (e) Soluzione ottima x^*, avente costo $c(x^*) = n(L+L')/2+nd$.

poligono esterno abbia raggio $r+d = 1+1/n^2$, si ottiene:

$$\begin{aligned} L &= 2\sin(\pi/n), \\ L' &= 2(1+1/n^2)\sin(\pi/n), \\ c(x') &= 2(n-1)(2+1/n^2)\sin(\pi/n)+2/n^2, \\ c(x^*) &= n(2+1/n^2)\sin(\pi/n)+1/n. \end{aligned}$$

Essendo:

$$\lim_{n\to\infty} n\sin(\pi/n) = \pi$$

si ottengono:

$$\begin{aligned} \lim_{n\to\infty} c(x') &= 4\pi \\ \lim_{n\to\infty} c(x^*) &= 2\pi \end{aligned}$$

e pertanto, al limite, $c(x')$ tende ad essere il doppio di $c(x^*)$.

I particolari dati di ingresso visti precedentemente, mostrano come lo scarto pessimo $|c(x')-c(x^*)|/|c(x^*)| \leq 1$ sia raggiungibile e quindi non migliorabile per il particolare algoritmo di approssimazione esposto. Questo però non deve fare erroneamente credere che tale algoritmo sia il migliore algoritmo di approssimazione possibile per il ΔTSP, poiché si potrebbe dimostrare uno scarto inferiore ad 1 per un altro algoritmo. In effetti, è noto un algoritmo di approssimazione più astuto, dovuto a Christofides (1976), che fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq 3c(x^*)/2$, cioè che non si scosta più del 50% da quella minima.

1.2.2 Non approssimabilità

Purtroppo, per molti problemi di ottimizzazione, trovare una soluzione approssimata è tanto difficile quanto trovarne una ottima! Questo è vero, per esempio, per le versioni di ottimizzazione di CRICCA, COLORAZIONE e COMMESSO VIAGGIATORE (senza diseguaglianze triangolari). A titolo di esempio, vediamo il seguente risultato.

Teorema 1.2 Non esiste alcun algoritmo di approssimazione assoluta per il COMMESSO VIAGGIATORE tale che $c(x') \leq sc(x^*)$, con s intero positivo, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

Dimostrazione. Per assurdo, si assuma l'esistenza di un algoritmo polinomiale A che fornisca una soluzione x' con $c(x') \leq sc(x^*)$, per un s intero positivo. Si consideri la riduzione CIRCUITO HAMILTONIANO $\propto$ COMMESSO VIAGGIATORE utilizzata nel Teorema 18.6 e si ponga:

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

Per costruzione, la soluzione ottima x^* del COMMESSO VIAGGIATORE è tale che $c(x^*) \leq sn$ se e solo se il grafo G ha un circuito Hamiltoniano.

Si possono presentare due casi. Se G ha un circuito Hamiltoniano, allora $c(x^*) = n$, e quindi A fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq sc(x^*) = sn$. Se invece G non ha alcun circuito Hamiltoniano, allora $c(x^*) > sn$, ed A fornisce, in tempo polinomiale, una soluzione x' per cui $c(x') \geq c(x^*) > sn$.

Pertanto, A restituisce una soluzione x' con $c(x') \leq sn$ se e solo se G ha un circuito Hamiltoniano. Questa è chiaramente una contraddizione, perché A risolverebbe in tempo polinomiale un problema NP-completo, il che è impossibile, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$. ■

1.2.3 ε-approssimazione

In taluni casi, è possibile progettare algoritmi di approssimazione che producono soluzioni approssimate il cui scarto dall'ottimo non è una costante che dipende dall'algoritmo (come nell'approssimazione assoluta), bensì un valore fornito come dato di input che può essere reso arbitrariamente piccolo. In altri termini, comunque si scelga un ε piccolo a piacere, con $0 < \varepsilon \leq 1$, tali algoritmi producono in tempo polinomiale nella dimensione del problema, ma in genere esponenziale in $1/\varepsilon$, una soluzione ammissibile x' tale che:

$$\frac{|c(x') - c(x^*)|}{|c(x^*)|} \leq \varepsilon$$

Algoritmi di questo tipo sono detti algoritmi di ε-approssimazione. Si consideri ad esempio la versione di ottimizzazione della SOMMA DI SOTTOINSIEME:

SOMMA DI SOTTOINSIEME. *Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, trovare un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i \leq k$ e $\sum_{i \in S} a_i$ sia massima.*

Sia $\varepsilon = 1/(h+1)$, con h intero maggiore o uguale ad uno (o, equivalentemente, $h = (1-\varepsilon)/\varepsilon$), e si supponga di memorizzare l'insieme A in un vettore di interi. Il problema può essere risolto con l'algoritmo ε-approssimato apSubsetSum(), basato sulla tecnica *greedy*, che genera dapprima nel ciclo (1) tutti i sottoinsiemi S di A di cardinalità al più h, e successivamente considera nel ciclo (2) ciascun elemento a_j di A, $1 \leq j \leq n$, aggiungendolo ad S qualora non sia già presente in S e non venga superata la soglia k.

Poiché, fissato un $m > 0$, ci sono $n(n-1)\cdots(n-m+1)/m! \leq n^m$ sottoinsiemi di $\{1, \ldots, n\}$ di cardinalità m, il ciclo (1) è ripetuto $\sum_{0 \leq m \leq h} n^m \leq hn^h + 1$ volte. Il ciclo (2), invece, è ripetuto n volte. Pertanto, la complessità dell'algoritmo apSubsetSum() è $O(hn^{h+1})$, che è polinomiale in n, ma esponenziale in h (e quindi in $1/\varepsilon$). Fissato un h, comunque, la complessità diviene polinomiale.

```
apSubsetSum(int[] A, int n, int k, int h)
  { ordina A in modo che A[1] ≥ A[2] ≥ ··· ≥ A[n] }
  SET maxSol = Set()
  int max = 0
(1) foreach S : S ⊆ {1,...,n} and |S| ≤ h do
      int z = Σ_{i∈S} A[i]
      if z ≤ k then
(2)     for j = 1 to n do
          if j ∉ S and z + A[j] ≤ k then
            S.insert(j)
            z = z + A[j]
        if z ≥ max then
          max = z
          maxSol = S
```

apSubsetSum() fornisce una soluzione $x' = maxSol$ di costo $c(x') = max$ tale che $|c(x') - c(x^*)|/|c(x^*)| \leq \varepsilon$. Per dimostrare questo risultato, sia $x^* = B$ il sottoinsieme ottimo di A con

$$|B| = m, B = \{b_1, b_2, \ldots, b_m\}, b_1 \geq b_2 \geq \cdots \geq b_m, \text{ e } c(x^*) = \sum_{1 \leq i \leq m} b_i.$$

Se $m \leq h$, allora l'insieme B è stato considerato nel ciclo (1) e apSubsetSum() ha fornito la soluzione ottima. Se invece $m > h$, essendo $c(x^*) = \sum_{1 \leq i \leq m} b_i$, si ha che $b_i \leq c(x^*)/i$ per $1 \leq i \leq h$, e quindi $b_i \leq c(x^*)/(h+1)$ per $h < i \leq m$. In tal caso, la procedura apSubsetSum() avrà considerato il sottoinsieme $B' = \{b_1, b_2, \ldots, b_h\}$ nel ciclo (1). Sia

$$b_i = \max\{b \in B : \text{apSubsetSum() non ha aggiunto } b \text{ a } B' \text{ nel ciclo (2)}\}.$$

Se tale b_i non esiste, apSubsetSum() ha trovato la soluzione ottima. Altrimenti,

$$c(x^*) - c(x') \leq b_i \leq \frac{c(x^*)}{h+1}$$

e quindi, dividendo per $c(x^*)$, si ottiene

$$\frac{|c(x^*) - c(x')|}{|c(x^*)|} \leq \frac{1}{h+1} = \varepsilon.$$

Esempio (Somma di sottoinsieme approssimato) Siano $n = 8$, $k = 45$, $A = \{23, 19, 13, 12, 11, 8, 7, 5\}$, $h = 1$ (e quindi $\varepsilon = 1/2$). La seguente tabella illustra la computazione di apSubsetSum():

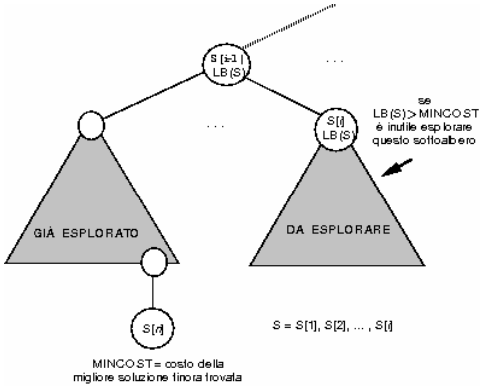

Figura 1.4: Esplorazione "intelligente" dell'albero delle scelte. **TODO**: Rifare figura

$\lvert S\rvert$	insieme costruito in (1)	insieme costruito in (2)	z
0	$\emptyset$	$\{23,19\}$	42
1	$\{23\}$	$\{23,19\}$	42
1	$\{19\}$	$\{23,19\}$	42
1	$\{13\}$	$\{23,13,8\}$	44
1	$\{12\}$	$\{23,12,8\}$	43
1	$\{11\}$	$\{23,11,8\}$	42
1	$\{8\}$	$\{23,13,8\}$	44
1	$\{7\}$	$\{23,13,7\}$	43
1	$\{5\}$	$\{23,13,5\}$	41

L'esecuzione di apSubsetSum() dà come risultato $maxSol = x' = \{23,13,8\}$ con $max = c(x') = 44$. Si noti che la soluzione ottima $B = x^* = \{19,13,8,5\}$, con $c(x^*) = k = 45$, sarebbe stata trovata dall'algoritmo scegliendo $h = 2$. ■

Ovviamente, la situazione ideale sarebbe quella di poter sempre progettare un algoritmo A di ε-approssimazione per un problema P che producesse una soluzione ε-approssimata in tempo polinomiale sia nella dimensione del problema sia in $1/\varepsilon$. Purtroppo, tali algoritmi sono noti soltanto per rari problemi di ottimizzazione che coinvolgono numeri, come ad esempio la SOMMA DI SOTTOINSIEME, per i quali esistono già algoritmi che trovano la soluzione ottima in tempo pseudopolinomiale. Un forte risultato negativo, a questo riguardo, è dato dal seguente teorema, che diamo senza dimostrazione.

Teorema 1.3 Sia P un problema di ottimizzazione tale che:

(1) la versione decisionale di P è fortemente NP-completa;

(2) per ogni insieme I di dati di input per P il valore $c(x^*)$ della soluzione ottima è limitato superiormente da $p(\#)$, dove p è un polinomio e # è il più grande numero che appare in I,

allora, a meno che $\mathbb{P}$ non coincida con $\mathbb{NP}$, non esiste alcun algoritmo di ε-approssimazione per P che richieda tempo polinomiale sia nella dimensione del problema che in $1/\varepsilon$. ■

1.3 Algoritmi branch-&-bound

Per risolvere un problema di ottimizzazione NP-arduo, si può modificare la procedura enumerazione(), vista nel Capitolo 16, in modo da evitare di effettuare certe sequenze di scelte che si rivelino incapaci di generare la soluzione ottima.

Si assuma che il problema da risolvere sia di minimo, che ogni sequenza di scelte abbia costo non negativo, e che ogni scelta, aggiunta alle scelte già effettuate, non faccia diminuire il costo della soluzione parziale così costruita. Si supponga di avere a disposizione una opportuna funzione "lower bound" $\mathsf{lb}(S,i)$, che dipenda dalla sequenza di scelte fatte $S[1,\ldots,i]$ e garantisca che tutte le soluzioni ammissibili generabili aggiungendo nuove scelte a tale sequenza abbiano costo maggiore o uguale a $\mathsf{lb}(S,i)$. Allora, è possibile mantenersi la migliore soluzione ammissibile *minSol* tra quelle finora esaminate, insieme al suo costo *minCost*, e valutare $\mathsf{lb}(S,i)$ ogniqualvolta è effettuata una nuova scelta $S[i]$ in un nodo dell'albero delle scelte. Se $\mathsf{lb}(S,i)$ è maggiore o uguale a *minCost*, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo (si veda la Figura 1.4).

Questo metodo, detto branch-&-bound, non migliora la complessità della procedura enumerazione(), che resta superpolinomiale nel caso pessimo, ma ne abbassa drasticamente il tempo di esecuzione in pratica. Ciò vale in special modo se si riesce ad individuare una "buona" funzione lb, che sia cioè il più vicino possibile al costo della soluzione ottima.

Supponiamo per semplicità che il numero di scelte per individuare una soluzione sia limitato da n, e che al passo i-esimo sia possibile calcolare l'insieme di possibili scelte C a partire dalle scelte passate $S[1,\ldots,i-1]$ tramite la funzione choices(). Sia $c(S,i)$ il costo della sequenza di scelte $S[1,\ldots,i]$. Inoltre, siano *minSol* e *minCost* parametri globali inizializzati a $\emptyset$ e $+\infty$. Allora, la procedura branch&bound() descrive una possibile struttura generale , da chiamarsi con parametro $i = 1$.

```
branch&bound(item[] S, int n, int i, ...)
SET C = choices(S,n,i,...)          % Determina C in funzione di S[1...i-1]
foreach c ∈ C do
    S[i] = c
    int lb = lb(S,i)
    if lb < minCost then
        if i < n then
            branch&bound(S,n,i+1,...)
        else
            if c(S,i) < minCost then
                minSol = S
                minCost = c(S,i)
```

Ovviamente, se $|C| \leq m$, per tutti i passi $1 \leq i \leq n$, ed il calcolo di $\mathsf{lb}()$ richiede $O(f(n))$ tempo, allora la complessità di branch&bound() è $O(m^n f(n))$. In pratica, il tempo di computazione di una procedura branch-&-bound è influenzato da svariati fattori:

- *Soluzione di partenza.* Anziché partire con $minCost = +\infty$, è in genere più conveniente determinare il costo di una soluzione ammissibile del problema ed inizializzare *minCost* al valore di tale costo;
- *Branch.* Prestazioni migliori si possono ottenere variando la cosiddetta regola di branch, ovvero il criterio per determinare le scelte da effettuare e generare i figli di un nodo nell'albero delle scelte;
- *Bound.* La "migliore" regola per calcolare lb() è quella che dà il lower bound più stretto, cioè più vicino possibile al costo della soluzione ottima, ma che sia nel contempo calcolabile velocemente;
- *Ordine di visita.* L'ordine più usato per generare e visitare l'albero delle scelte è senza dubbio quello anticipato (LIFO) precedentemente esposto, perché richiede meno spazio;

infatti, utilizzando una pila, il massimo numero di "record di attivazione" è proporzionale alla lunghezza di un percorso radice-foglia, che è $O(n)$; talvolta, sono usati criteri diversi, in cui l'albero è visitato per livelli (FIFO) o utilizzando una priorità (per esempio considerando prima quel figlio avente il più promettente lower bound);

– *Scarto dall'ottimo*. Anziché raggiungere la soluzione ottima a tutti i costi, l'esecuzione può essere interrotta quando lo scarto dall'ottimo scenda sotto una certa soglia prestabilita, cioè quando $(minCost - \mathsf{lb}^*)/\mathsf{lb}^* \leq e$, con e costante prefissata ed lb^* uguale al più piccolo lower bound calcolato per i nodi generati ma non ancora visitati dell'albero delle scelte.

Come esempio di tecnica branch-&-bound, risolviamo il problema del COMMESSO VIAGGIATORE (senza diseguaglianze triangolari), per il quale sappiamo che non può esistere né un algoritmo ottimo pseudopolinomiale né un algoritmo di approssimazione assoluta polinomiale (a meno che $\mathbb{P} = \mathbb{NP}$). Cerchiamo dunque di specializzare la procedura branch&bound() vista precedentemente.

1.3.1 Commesso viaggiatore

Sia n il numero di città e $d[h,k]$ la distanza, intera non negativa, tra le città h e k, $1 \leq h,k \leq n$. Si assuma che al passo i-esimo siano state fatte le scelte $S[1,\ldots,i]$ prese dall'insieme $\{1,\ldots,n\}$. Ovviamente, un qualsiasi percorso ammissibile del commesso viaggiatore generabile "espandendo" $S[1,\ldots,i]$ deve attraversare le città $S[1],S[2],\ldots,S[i]$ in quest'ordine, passare da $S[i]$ ad una qualsiasi delle rimanenti $n-i$ città, attraversare queste ultime città in un ordine qualsiasi, e da una di queste ritornare ad $S[1]$. Si pongano:

$$\begin{aligned}
A &= \min_{h \notin S}\{d[S[1],h]\},\\
B &= \min_{h \notin S}\{d[S[i],h]\},\\
C[i] &= C[i-1] + d[S[i-1],S[i]],\\
C[1] &= 0,\\
D[h] &= \min_{p,q}\{d[p,h] + d[h,q] : h \neq p \neq q\}, \text{ per ogni } h \notin S.
\end{aligned}$$

Allora, un lower bound $\mathsf{lb}(S)$ è dato dalla seguente espressione:

$$\begin{aligned}
\mathsf{lb}(S) &= C[i] + \left\lceil (A + B + \sum_{h \notin S} D[h])/2 \right\rceil, && \text{se } i < n,\\
\mathsf{lb}(S) &= C[i] + d[S[i],S[1]], && \text{se } i = n.
\end{aligned}$$

Infatti, $C[i]$ è la distanza complessiva per attraversare, nell'ordine, le città $S[1],S[2],\ldots,S[i]$. Invece, A è un lower bound della distanza tra una qualsiasi delle rimanenti $n-i$ città ed $S[1]$, mentre B è un lower bound della distanza tra $S[i]$ ad una qualsiasi di tali $n-i$ città. Infine, $D[h]$ è un lower bound della distanza percorsa per attraversare una qualsiasi di queste ultime $n-i$ città, provenendo da (e dirigendosi verso) un'altra di queste $n-i$ città.

Esempio (Lower bound per il COMMESSO VIAGGIATORE) Si consideri la matrice delle

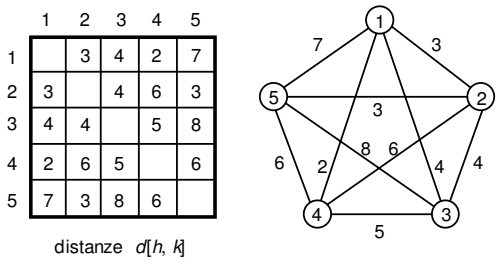

Figura 1.5: Dati di ingresso per il COMMESSO VIAGGIATORE.

distanze fornita nella Figura 1.5 per $n = 5$ città. Siano $i = 2$, $S[1] = 1$ e $S[2] = 3$. Allora

$$\begin{aligned} A &= \min\{\, d[1,2], d[1,4], d[1,5] \,\} = \min\{\, 3,2,7 \,\} = 2; \\ B &= \min\{\, d[3,2], d[3,4], d[3,5] \,\} = \min\{\, 4,5,8 \,\} = 4; \\ C[2] &= C[1] + d[1,3] = 4; \\ D[2] &= 3+3 = 6; \\ D[4] &= 2+5 = 7; \\ D[5] &= 3+6 = 9; \\ \mathsf{lb}(S) &= 4 + \lceil (2+4+6+7+9)/2 \rceil = 18. \quad \blacksquare \end{aligned}$$

Per non generare più volte lo stesso percorso a partire da tutte le città che lo compongono, la prima scelta può essere fatta una volta per tutte nel programma chiamante. Assumendo per esempio $S[1] = 1$, la procedura ricorsiva bbTsp() può essere richiamata direttamente con parametro $i = 2$, dopo aver inizializzato $C[1]$ ed lb a zero e R a $\{2,3,\ldots,n\}$.

```
bbTsp(item[] S, int[] C, SET R, int n, int i)
foreach c ∈ R do
    S[i] = c
    R.remove(c)
    C[i] = C[i−1] + d[S[i−1], S[i]]
    {calcola A, B, e D[h] per ogni h ∈ R}
    int lb = C[i] + iif(i < n, ⌈(A + B + Σ_{h∉S} D[h])/2⌉, d[S[i], S[1]])
    if lb < minCost then
        if i < n then
            bbTsp(S, C, R, n, i+1, ...)
        else
            C[n] = lb
            minSol = S
            minCost = C[n]
    R.insert(c)
```

Per risparmiare tempo, anziché partire con $minCost = +\infty$, si può partire inizializzando $minCost$ al valore di una qualsiasi soluzione ammissibile, cioè di una permutazione di $\{1,\ldots,n\}$. Una tale

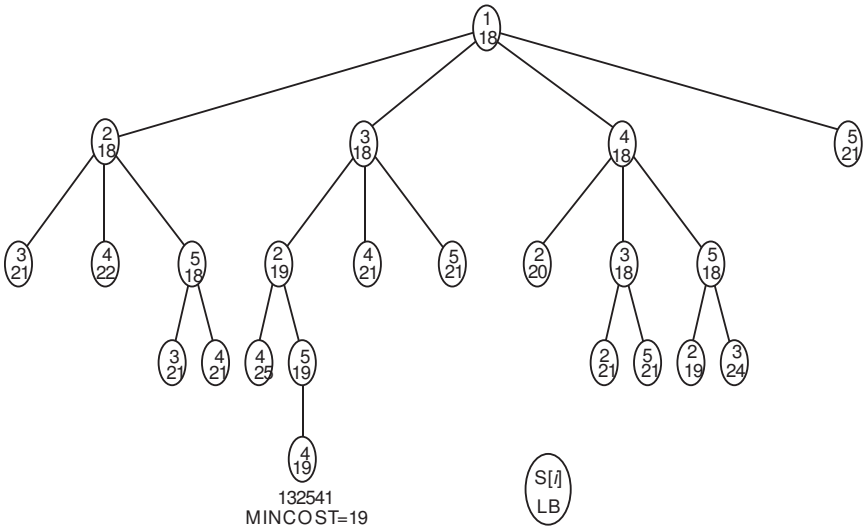

Figura 1.6: Albero delle scelte generato dalla procedura bbTsp() per i dati della Figura 1.5 con *minCost* inizializzato a 21.

permutazione può essere generata casualmente in tempo $O(n)$ avvalendosi della funzione random() introdotta nel Capitolo 17. Infatti, si consideri un vettore $V[1\ldots n]$ inizializzato in modo che $V[i] = i$ per $i = 1,2,\ldots,n$. Allora, una permutazione delle n città può essere posta in un secondo vettore $P[1\ldots n]$ come segue:

for $i = 1$ **to** n **do**
  int $r =$ random(i,n)
  $P[i] = V[r]$
  $V[r] = V[i]$

Esempio (bbTsp()) Sia $4,3,2,5,1$ la permutazione casuale generata per $n = 5$ città. Questa corrisponde al circuito $1,4,3,2,5,1$ che, per i dati della Figura 1.5, ha costo $2+5+4+3+7 = 21$. Si assuma pertanto di aver inizializzato *minCost* a 21. Allora, applicando la procedura bbTsp() ai dati della Figura 1.5, si genera l'albero delle scelte mostrato nella Figura 1.6. Il confronto "lb $<$ *minCost*" permette di "potare" ben 42 nodi sui 65 possibili, con un notevole risparmio del tempo totale di esecuzione. ■

È bene rimarcare come la procedura bbTsp() appena descritta non sia che un semplice esempio (e sicuramente non il migliore) di impiego della tecnica branch-&-bound per il COMMESSO VIAGGIATORE. In effetti, il branch-&-bound è una tecnica così generale che si possono progettare molte procedure diverse per lo stesso problema variandone le regole di branch, le funzioni lower bound, e l'ordine di visita dell'albero delle scelte. Per esempio, ciascun nodo dell'albero può riguardare la scelta di un arco (anziché un nodo) del grafo K_n e avere così due soli figli, corrispondenti all'inclusione oppure all'esclusione dell'arco, mentre può essere considerato l'arco la cui esclusione massimizza il lower bound. Una funzione alternativa per il lower bound può essere ottenuta con un 1-*albero*, dove un 1-albero per K_n è formato da un albero di copertura per i nodi 2, 3, ..., n più due archi incidenti nel nodo 1. È immediato osservare che ogni ciclo Hamiltoniano per K_n è un 1-albero, ma non viceversa, per cui il peso di un 1-albero minimo è un lower bound al costo della soluzione ottima (un 1-albero minimo si può facilmente trovare in tempo polinomiale). In pratica, uno dei migliori lower bound per il COMMESSO VIAGGIATORE è stato ottenuto da Held e Karp (1971)

modificando ulteriormente la nozione di 1-albero. Infine, i nodi dell'albero delle scelte possono essere considerati per priorità, per esempio considerando per primo il nodo col lower bound più piccolo.

1.4 Algoritmi euristici

Quando si è presi dalla disperazione a causa della enorme difficoltà di un problema di ottimizzazione NP-arduo, si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile, non necessariamente ottima né approssimata. Tali euristiche sono di solito basate su tecniche *greedy* [cfr. Capitolo 14] oppure di ricerca locale [cfr. Capitolo 15].

1.4.1 Greedy

Iniziamo progettando un algoritmo euristico di tipo *greedy* per il problema del COMMESSO VIAGGIATORE. Consideriamo l'input come grafo non orientato completo e pesato. Ordiniamone gli archi per pesi non decrescenti e proviamo ad aggiungere archi alla soluzione seguendo questo ordine finché non sono stati aggiunti $n-1$ archi, dove n è il numero di nodi. Per ogni nodo, contiamo quanti archi incidenti in quel nodo sono stati aggiunti alla soluzione. Per poter aggiungere un arco, occorre verificare che per ciascuno dei suoi nodi non siano stati già scelti due archi e che non si formino circuiti. Per verificare quest'ultima proprietà, si utilizza un MFSET, terminando di aggiungere archi quando l'MFSET è formato da una sola componente. A questo punto, si è trovata una catena Hamiltoniana e si chiude il circuito aggiungendo l'arco tra i due nodi estremi della catena, ciascuno dei quali ha un solo arco incidente. Questa euristica richiede tempo $O(n^2 \log n)$ e si può riassumere nella procedura greedyTsp().

```
SET greedyTsp(GRAPH G)
    SET S = Set()
    MFSET M = Mfset(G.n)
    int[] in = new int[1...n]
    for i = 1 to G.n do in[i] = 0
    { ordina gli archi per peso non decrescente }
    foreach [u,v] ∈ G.E do
        if in[u] < 2 and in[v] < 2 and M.find(u) ≠ M.find(v) then
            S.insert([u,v])
            in[u] = in[u] + 1
            in[v] = in[v] + 1
            M.merge(u,v)
    int u = 1; while in[u] ≠ 1 do u = u + 1
    int v = u + 1; while in[v] ≠ 1 do v = v + 1
    S.insert([u,v])
    return S
```

Esempio (Commesso viaggiatore "ingordo") Applicando l'algoritmo euristico greedyTsp() ai dati di ingresso della Figura 1.5, si inseriscono in S, nell'ordine, gli archi $[1,4]$, $[1,2]$, $[2,5]$, $[3,4]$, $[3,5]$ che producono il circuito 1, 4, 3, 5, 2, 1 il cui costo è 21. Tale soluzione può essere utilizzata come soluzione di partenza per poi applicare o un algoritmo branch-&-bound come bbTsp() (dopo aver inizializzato *minCost* a 21) o un'ulteriore euristica di ricerca locale

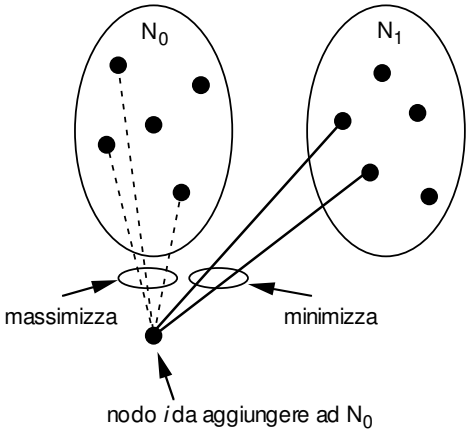

Figura 1.7: Euristica *min-max-greedy* per la BISEZIONE: situazione prima di aggiungere un nuovo nodo a V_0. TODO: "N_0" $\Rightarrow$ "V_0", "N_1" $\Rightarrow$ "V_1"

> [cfr. Capitolo 1.4.2], che di solito per il COMMESSO VIAGGIATORE si comporta meglio di *greedy*. ■

Come secondo esempio, consideriamo ora il problema della BISEZIONE [cfr. Capitolo 18] in forma di ottimizzazione:

BISEZIONE. *Dato un grafo non orientato $G = (V,E)$ di n nodi, trovare una partizione di V in due sottoinsiemi V_0 ed V_1 di $n/2$ nodi ciascuno tale che il numero di archi con un estremo in V_0 e l'altro estremo in V_1 sia minimo.*

Dato G con n pari, una semplice euristica *greedy* può essere basata sulle considerazioni seguenti. Si inizia selezionando casualmente un nodo che è posto in V_0 ed un altro nodo che è posto in V_1. Si prosegue aggiungendo alternatamente un nodo a V_0 e il nodo successivo a V_1, fino all'esaurimento dei nodi. Come mostrato nella Figura 1.7, il nodo u da aggiungere in ciascun passo a V_k è scelto in modo da minimizzare il numero di archi $[u,v]$ con $v \in V_{1-k}$. Nel caso ci siano più nodi che soddisfano tale criterio, però, si sceglie tra questi il nodo u che massimizza il numero di archi $[u,v]$ con $v \in V_k$. Se ci sono più nodi che soddisfano anche il secondo criterio, allora il nodo u è scelto casualmente tra questi. Tale euristica, detta *min-max-greedy*, si comporta molto bene, proprio grazie al secondo criterio di scelta. Infatti, gli archi $[u,v]$ con $v \in V_k$ non andranno ad aumentare il costo della partizione, poiché u è aggiunto proprio ad V_k. Pertanto, più archi di questo tipo sono aggiunti subito, minore diviene la possibilità che nei passi successivi si sia costretti a scegliere un nodo che, introducendo parecchi archi tra V_k ed V_{1-k}, faccia aumentare di molto il costo della partizione.

L'euristica *min-max-greedy* può essere realizzata in modo da avere complessità $O(n^2)$ nel caso pessimo, utilizzando una apposita struttura di dati detta "a cestini" (cfr. Esercizio 1.17). Prove sperimentali hanno mostrato che in pratica *min-max-greedy* fornisce "buone" soluzioni. In questo caso, è opportuno ripetere l'esecuzione dell'euristica un numero prefissato di volte, per esempio 100, scegliendo ogni volta, in modo casuale, una nuova coppia di nodi di partenza da inserire in V_0 ed V_1, e selezionando alla fine la migliore delle 100 soluzioni individuate.

1.4.2 Ricerca locale

Per introdurre le tecniche euristiche basate su ricerca locale, si consideri ancora il problema del COMMESSO VIAGGIATORE e sia π un circuito Hamiltoniano del grafo completo non orientato K_n.

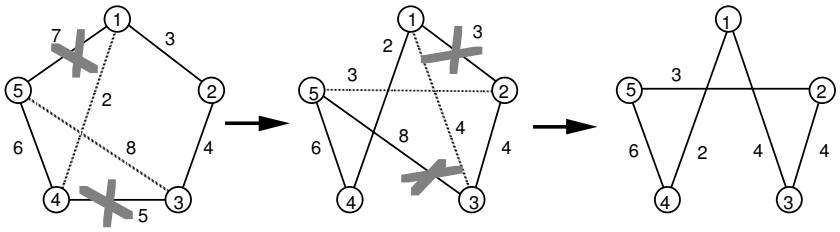

Figura 1.8: Euristica di ricerca locale con intorno I_2 per il COMMESSO VIAGGIATORE.

Un'euristica ragionevole di ricerca locale può essere progettata sulla base del seguente intorno:

$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito}\}.$$

Si noti che $|I_2(\pi)| = n(n-1)/2 - n$. Infatti, cancellati due archi non consecutivi, c'è un unico modo per riconnettere i due spezzoni in un circuito con archi che non fanno parte di π, mentre il numero di tutte le possibili coppie di archi non consecutivi di π è dato da quello di tutte le $n(n-1)/2$ coppie meno le n coppie di archi consecutivi. Pertanto, in un'euristica di ricerca locale, il passaggio da una soluzione π ad una $\pi' \in I_2(\pi)$ richiede $O(n^2)$ tempo.

Esempio (Ricerca locale per COMMESSO VIAGGIATORE) Si consideri un'euristica di ricerca locale con intorno I_2 e la si esegua sui dati della Figura 1.5. Come illustrato nella Figura 1.8, partendo dal circuito iniziale $\pi = 1,2,3,4,5,1$ con costo $c(\pi) = 25$, e scambiando gli archi $[1,5]$ e $[3,4]$ con gli archi $[1,4]$ e $[3,5]$, si ottiene la nuova soluzione $\pi' = 1,2,3,5,4,1$ di costo $c(\pi') = 23$. Scambiando ulteriormente gli archi $[1,2]$ e $[3,5]$ con $[1,3]$ e $[2,5]$, si ottiene la soluzione ottima $\pi'' = 1,3,2,5,4,1$ di costo $c(\pi'') = 19$. ■

Ovviamente, una soluzione iniziale può essere generata casualmente oppure costruita tramite l'euristica greedyTsp(). Purtroppo, il numero complessivo di soluzioni π esaminate dall'euristica di ricerca locale può essere esponenziale. Inoltre, non vi è alcuna garanzia che la soluzione trovata sia ottima. Infatti, come già è stato osservato nel Capitolo 15, un algoritmo di ricerca locale si arresta su un ottimo locale, cioè su una soluzione π il cui costo $c(\pi)$ è minore di $c(\pi')$ per ogni $\pi' \in I_2(\pi)$. In generale, niente assicura che tale soluzione sia anche un ottimo globale, poiché può esistere una soluzione $\pi^* \notin I_2(\pi)$ con costo $c(\pi^*)$ minore di $c(\pi)$, per esempio una ottenibile scambiando 3 o 4 archi anziché due soli archi.

Generalizzando, si può definire, per $2 \leq k \leq n$, un intorno $I_k(\pi)$ dato da tutte le soluzioni π' ottenibili da π cancellando k archi del circuito e sostituendoli con k archi esterni al circuito. Purtroppo, è possibile dimostrare che l'unico intorno che garantisce sempre che l'ottimo locale trovato sia anche l'ottimo globale si ottiene solo dall'unione di $I_2(\pi), I_3(\pi), \ldots, I_n(\pi)$. Comunque, prove sperimentali hanno mostrato che per l'intorno dato dall'unione di $I_2(\pi)$ e $I_3(\pi)$ si ottiene una euristica di ricerca locale che fornisce in pratica "buone" soluzioni. Ovviamente, tale euristica ha complessità esponenziale nel caso pessimo e non vi è alcuna garanzia che la soluzione prodotta sia vicina a quella ottima! In tal caso, è pratica comune ripetere l'esecuzione dell'euristica di ricerca locale un numero prefissato di volte, per esempio 100, scegliendo ogni volta, in modo casuale, una soluzione ammissibile di partenza, e selezionando alla fine la migliore delle 100 soluzioni (ottimi locali) così individuate.

Anche per il problema della BISEZIONE si possono definire euristiche di ricerca locale, che magari iniziano proprio a partire dalla soluzione prodotta da *min-max-greedy* e poi continuano

(a)

2	3		4
1	5	7	8
9	6	10	12
13	14	11	15

(b)

2	3	7	4
1	5		8
9	6	10	12
13	14	11	15

(c)

2	3	7	4
1	5	8	
9	6	10	12
13	14	11	15

(d)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figura 1.9: Il gioco del quindici. (a) Una configurazione iniziale; (b) una mossa a partire da (a); (c) una mossa a partire da (b); (d) la configurazione finale.

scambiando ripetutamente tra loro un nodo di V_0 con uno di V_1. Purtroppo, mentre per il problema del COMMESSO VIAGGIATORE le euristiche di ricerca locale risultano nettamente superiori a quelle *greedy*, per la BISEZIONE invece accade esattamente il contrario, e per ottenere soluzioni migliori occorre considerare tecniche euristiche più raffinate della ricerca locale.

La ricerca locale, infatti, tentando sempre di migliorare il costo della soluzione, presenta l'inconveniente di bloccarsi su un minimo locale. Per superare tale inconveniente, sono state introdotte tecniche più sofisticate di ricerca che permettono anche di peggiorare il costo della soluzione, nella speranza di sfuggire da un minimo locale. Tali tecniche devono però "evitare" la possibilità di effettuare troppo presto "mosse" inverse di "miglioramento", per non ricadere subito sul medesimo minimo locale. A tal fine, vengono introdotte delle "proibizioni" delle mosse che devono valere per un certo lasso di tempo, e per questo tali tecniche sono dette di "ricerca tabù".

1.5 Reality Check

Il GIOCO DEL QUINDICI è un rompicapo inventato da Samuel Loyd nel 1878. Il gioco consiste in un griglia quadrata di $n \times n$ posizioni (nella forma classica, $n = 4$), con $n^2 - 1$ tessere quadrate di lato unitario (nel caso di $n = 4$, il numero di tessere è 15, da cui il nome), numerate da 1 a $n^2 - 1$, che coprono tutte le caselle della griglia, tranne una casella che rimane vuota. Una "mossa" consiste nello spostare orizzontalmente o verticalmente una tessera adiacente alla casella vuota, coprendo la casella vuota ma scoprendo la casella dove si trovava la tessera. Lo scopo del gioco è riordinare le tessere in modo crescente dopo averle "mescolate" in modo casuale.

Esempio (Gioco del quindici) La quattro tabelle in Figura 1.9 corrispondono ad (a) una configurazione iniziale; (b) una mossa a partire da (a), spostando il 7 nella casella vuota; (c) una mossa a partire da (b), spostando l'8 nella casella vuota; (d) la configurazione finale.

È possibile risolvere questo problema attraverso un algoritmo branch-&-bound. Per ogni configurazione delle tessere, possiamo definire il lower bound come la somma delle "distanze" di una tessera dal suo posto, ovvero il numero di caselle attraverso le quali la tessera deve comunque transitare per portarsi al proprio posto. Per esempio, nella configurazione (a) mostrata nella Figura 1.9 le tessere 4,7,8,9,12,13,14 sono al loro posto, mentre le tessere 2,3,1,5,6,10,11,15 si trovano in caselle adiacenti ai rispettivi posti di competenza e quindi ognuna di esse contribuisce con distanza 1; pertanto il lower bound per la configurazione (a) è pari a 8. Si noti che per la configurazione (b) il lower bound è 9, mentre per la configurazione (c) il lower bound è 10, quindi non sono certo mosse da seguire.

La radice dell'albero delle scelte corrisponde alla configurazione iniziale (la (a) nel nostro esempio), mentre la regola di branch prevede di generare tutti i figli di un nodo che corrispondono alle configurazioni ottenibili con una mossa a partire dalla configurazione di tale nodo. Per esempio, il nodo (a) avrà 3 figli: uno è (b), gli altri si ottengono spostando le tessere 4 e 3. Ogni nodo dell'albero delle scelte viene caratterizzato dal suo livello (distanza dalla radice) che ci dà il

numero di mosse effettuate (costo) per ottenere la corrispondente configurazione. L'ordine di visita dell'albero delle scelte si svolge per priorità, utilizzando una coda con priorità: si visita per primo quel nodo non ancora visitato per il quale è minima la somma del suo livello e del suo lower bound.

1.6 Esercizi

Esercizio 1.1 (Partizione)**.** Si progetti un algoritmo di programmazione dinamica con complessità pseudopolinomiale per la PARTIZIONE.

Esercizio 1.2 (Scheduling multiprocessore)**.** Fornire un algoritmo di programmazione dinamica con complessità pseudopolinomiale per il problema dello SCHEDULING MULTIPROCESSORE definito nel Capitolo 18. Si noti che non è possibile frazionare l'esecuzione di un programma in più intervalli di tempo, a differenza del problema REAL-TIME SCHEDULING.

Esercizio 1.3 (Algoritmo di McNaughton)**.** Si dimostri che SCHEDULING MULTIPROCESSORE in forma di ottimizzazione è risolubile in tempo $O(n)$ se è possibile frazionare l'esecuzione di un programma in più intervalli (aventi lunghezze reali), evitando ovviamente che due intervalli dello stesso programma siano eseguiti nello stesso istante da più di un processore (Suggerimento: si consideri $d = \max\{\max_i\{t_i\}, (1/m)\sum_i t_i\}$).

Esercizio 1.4 (Real-time scheduling)**.** Si dimostri che REAL-TIME SCHEDULING, in forma decisionale, è risolubile in tempo polinomiale anche per $m \geq 1$ processori, qualora non sia richiesto di usare un algoritmo guidato da priorità (Suggerimento: ci si riconduca all'Esercizio 1.3, ponendo $d = 1$ e $t_i = C_i/T_i$).

Esercizio 1.5 (I "soliti ignoti")**.** La scorsa notte, i "soliti ignoti" hanno svuotato l'oreficeria "Bella Gioia", usando la tecnica del "buco". Dopo il fattaccio, i tre componenti della banda si sono riuniti per spartire il bottino in parti uguali. Il valore del bottino è stimato in m Euro, con m multiplo di 3, sommando i prezzi riportati sui cartellini degli n monili. Dopo lunghe discussioni, il capo della banda vi ha incaricato di scrivere una procedura per effettuare la spartizione usando la tecnica della programmazione dinamica.

Esercizio 1.6 (Minima copertura con nodi)**.** Dato un grafo non orientato G, il problema della MINIMA COPERTURA CON NODI chiede di trovare un sottoinsieme S di nodi avente minima cardinalità tale che ciascun arco abbia almeno un estremo in S. Si progetti un algoritmo di approssimazione assoluta.

Esercizio 1.7 (ΔTSP euclideo)**.** Si assuma che le n città del ΔTSP siano punti nel piano e che le distanze tra le città siano le distanze Euclidee tra i punti. Si dimostri che un percorso ottimo del commesso viaggiatore non incrocia mai se stesso.

Esercizio 1.8 (Zaino, ε-approssimato)**.** Si fornisca un algoritmo ε-approssimato per il problema dello ZAINO in forma di ottimizzazione.

Esercizio 1.9 (Cricca di peso massimo)**.** Si progetti, usando la tecnica del branch-&-bound, un algoritmo per il problema della CRICCA DI PESO MASSIMO: Dato un grafo non orientato $G = (V, E)$ con pesi interi non negativi sui nodi, trovare un sottoinsieme S dei nodi tale che esista un arco tra ogni coppia di nodi in S e la somma dei pesi dei nodi in S sia la più grande possibile.

Esercizio 1.10 (1-albero minimo)**.** Si progetti un algoritmo di complessità polinomiale per trovare un 1-albero minimo di un grafo non orientato e pesato. Si riesegua poi l'algoritmo bbTsp(), utilizzando un 1-albero minimo per calcolare il lower bound, sui dati dell'Esempio 1.3.1.

Esercizio 1.11 (Riduzione matrice delle distanze)**.** Si progetti, usando la tecnica del branch-&-bound, un algoritmo per il COMMESSO VIAGGIATORE utilizzando i seguenti criteri: (1) Scelta: inclusione/esclusione di un arco; (2) Regola di branch: considerare l'arco la cui esclusione massimizza il lower bound; (3) Visita: per priorità (cioè per lower bound più piccolo); (4) Lower bound: metodo di "riduzione" sulla matrice delle distanze (con lb inizializzato a 0):

(1) calcolare $r_i = \min_j\{d_{ij}\}$ per ogni riga i della matrice;
(2) se $r_i > 0$, sottrarre r_i a tutti gli elementi d_{ij} della riga i e sommare r_i ad lb;
(3) calcolare $c_j = \min_i\{d_{ij}\}$ per ogni colonna j della matrice;
(4) se $c_j > 0$, sottrarre c_j a tutti gli elementi d_{ij} della colonna j e sommare c_j ad lb;

Si esegua l'algoritmo sui dati dell'Esempio 1.3.1.

Esercizio 1.12 (Zaino, B&B)**.** Si progetti un algoritmo branch-&-bound per il problema dello ZAINO in forma di ottimizzazione.

Esercizio 1.13 (Minima superstringa comune)**.** Date n stringhe $P_1,\ldots,P_n$, il problema della MINIMA SUPERSTRINGA COMUNE chiede di trovare la più corta stringa T che è una superstringa di ciascuna P_i. Dato che questo problema è NP-arduo, si progetti una euristica di tipo *greedy* modificando quella per il COMMESSO VIAGGIATORE vista in questo capitolo.

Esercizio 1.14 (Ottimo locale per COMMESSO VIAGGIATORE)**.** Si costruisca un grafo K_6 per il COMMESSO VIAGGIATORE per il quale esistano due soluzioni π e π^* tali che π sia un ottimo locale per l'intorno $I_2(\pi)$, mentre π^* sia un ottimo globale che non appartiene ad $I_2(\pi)$.

Esercizio 1.15 (Cardinalità di I_k)**.** Si dimostri che la cardinalità di $I_k(\pi)$, per il COMMESSO VIAGGIATORE è $O(n^k)$. Per $k > 2$, esiste un unico modo di ricomporre il circuito dopo aver tolto k archi?

Esercizio 1.16 (Partizione)**.** Si progetti un algoritmo euristico di ricerca locale per il problema della PARTIZIONE in forma di ottimizzazione, dove si vuole minimizzare lo scarto tra la somma degli elementi della prima parte e la somma di quelli della seconda parte.

Esercizio 1.17 (*min-max-greedy*)**.** Le operazioni richieste dall'euristica *min-max-greedy* possono essere realizzate usando una struttura di dati detta "a cestini". Vengono utilizzate due matrici $cestino[0\ldots1,0\ldots n,0\ldots n]$ e $dim[0\ldots1,0\ldots n]$. In generale, $cestino[k,h,d]$ contiene un nodo u tale che il numero di archi $[u,v]$ con $v \in V_k$ è uguale ad h, mentre $dim[k,h]$ contiene il numero di tali nodi. Inoltre, sono mantenuti in $min[k]$ e $max[k]$ i valori del minimo e massimo "cestino" non vuoto (si veda la Figura 1.10, notando come ogni nodo compaia in entrambi gli insiemi). Si considerino le seguenti operazioni:

– insert(u), che inserisce il nodo u nel cestino;
– remove(u), che cancella il nodo u dal cestino;
– inc(k,u), che che aumenta di 1 il numero di archi $[u,v]$ con $v \in V_k$.

Si scrivano procedure efficienti per realizzare tali operazioni, aggiornando opportunamente i valori $min[k]$ e $max[k]$. Si mostri poi come realizzare efficientemente la scelta del nodo i da aggiungere alla partizione.

Esercizio 1.18 (Allineamento globale multiplo)**.** Generalizzando il problema dell'allineamento globale visto nell'Esercizio 13.7 ad un numero arbitrario k di stringhe $P_1,\ldots,P_k$, si ottiene il cosiddetto problema dell'ALLINEAMENTO GLOBALE MULTIPLO che però è NP-arduo. Si progetti un algoritmo euristico per l'allineamento globale multiplo basato sull'algoritmo per l'allineamento globale tra 2 stringhe.

Esercizio 1.19 (Colorazione)**.** Si progetti un algoritmo euristico per il problema della COLORAZIONE in forma di ottimizzazione.

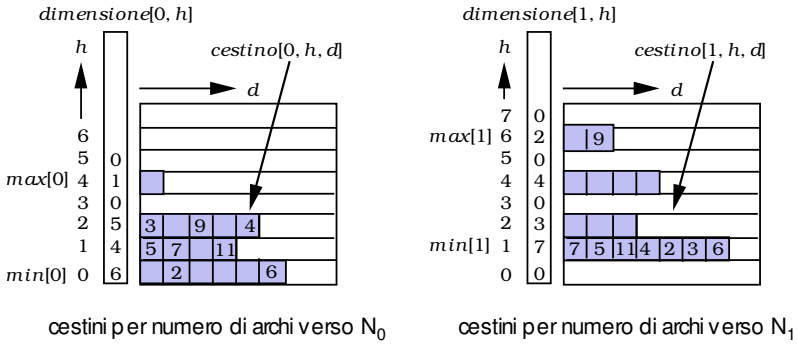

Figura 1.10: Struttura di dati “a cestini”.

1.7 Soluzioni

Soluzione Esercizio 1.5

La procedura bottino() utilizza una matrice booleana a tre dimensioni $S[1\ldots n,-m\ldots m/3,-m\ldots m/3]$. $S[i,j,h]=$ **true** se i primi i monili sono spartiti dando valore j al primo ladro, h al secondo, ed il rimanente al terzo. La spartizione è possibile se e solo se $S[n,m/3,m/3]=$ **true**, e la soluzione si può ricavare a ritroso a partire da $S[n,m/3,m/3]$. La procedura ha complessità pseudopolinomiale $O(nm^2)$.

```
bottino(int[] P, int n, int m, int[][][] S)
for i = 1 to n do
    for j = −m to m/3 do
        for h = −m to m/3 do
            S[i, j, h] = false
S[1, P[1], 0] = true
S[1, 0, P[1]] = true
S[1, 0, 0] = true
for i = 2 to n do
    for j = −m to m/3 do
        for h = −m to m/3 do
            S[i, j, h] = S[i − 1, j − P[i], h] or S[i − 1, j, h − P[i]] or S[i − 1, j, h]
```

Soluzione Esercizio 1.6

Si parte con una soluzione S vuota e si scandiscono uno dopo l’altro tutti gli archi di G. Se si incontra un arco $e=[u,v]$ per il quale entrambi i nodi estremi u e v non sono stati ancora selezionati, allora si aggiungono i due nodi u e v alla soluzione S. Tale algoritmo adotta una strategia *greedy*, poiché la scelta permette di coprire non solo l’arco e, ma anche tutti gli archi incidenti nei nodi u e v. Inoltre garantisce di coprire tutti gli archi poiché l’arco e non è selezionato se almeno uno dei suoi nodi estremi è già stato incluso in S. La complessità è $O(n+m)$ qualora si memorizzi il grafo con liste di adiacenza. Si osservi che il sottografo indotto dall’insieme S consiste di $|S|/2$ archi nessuna coppia dei quali ha un nodo estremo in comune. D’altro canto, per coprire tale sottografo occorrono almeno $|S|/2$ nodi e quindi una copertura ottima S^* di G ha cardinalità non inferiore ad $|S|/2$. Ma allora $|S|$ non supera il doppio della cardinalità minima $|S^*|$.

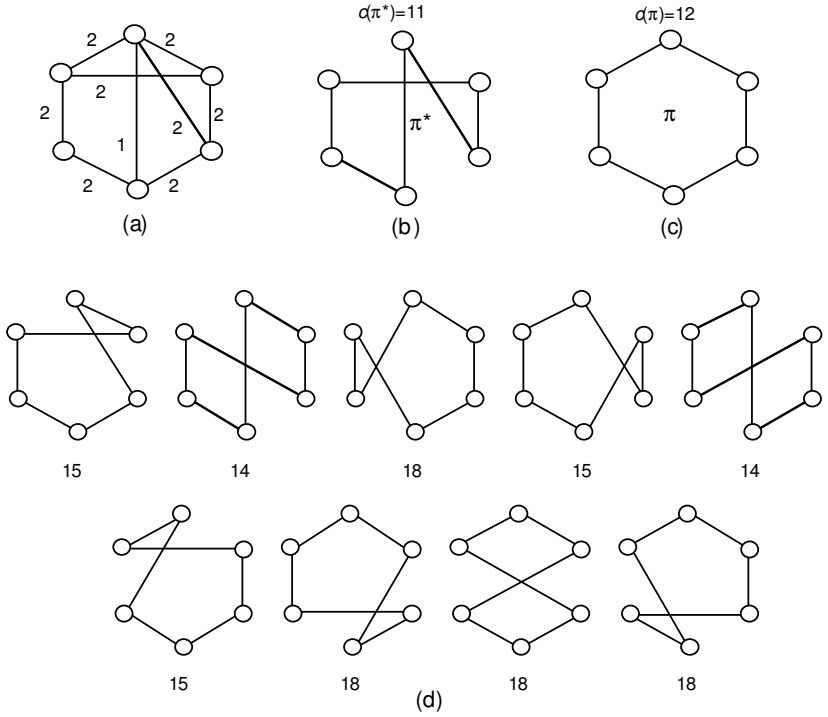

Figura 1.11: (a) Dati per il COMMESSO VIAGGIATORE (le distanze non indicate valgono 5). (b) Ottimo globale π^*. (c) Un ottimo locale π. (d) Tutte le soluzioni π' che stanno in $I_2(\pi)$.

Soluzione Esercizio 1.13

Si assuma per semplicità e senza perdere in generalità che nessuna stringa sia una sottostringa di un'altra. Costruiamo un grafo completo orientato e pesato G (detto grafo delle *sovrapposizioni*) i cui nodi corrispondono alle stringhe e i cui archi orientati corrispondono a coppie ordinate di stringhe. Il peso c_{ij} dell'arco (i, j) è uguale al numero massimo di caratteri identici tra un suffisso di P_i ed un prefisso di P_j, ma cambiato di segno (per esempio, se $P_i = \texttt{ACGA}$ e $P_j = \texttt{GACA}$, allora $c_{ij} = -2$, mentre $c_{ji} = -1$). Trovare la minima superstringa comune corrisponde a trovare il cammino Hamiltoniano di peso minimo nel grafo G. L'euristica *greedy* è simile alla procedura greedyTsp(), con la differenza che per ogni nodo i occorrono due contatori $in[i]$ e $out[i]$ inizializzati a 0, che un arco orientato (i, j) è aggiunto alla soluzione se $out[i] < 1$ **and** $in[j] < 1$ **and** $S.\mathsf{find}(i) \neq S.\mathsf{find}(j)$, e che ovviamente si evita di richiudere il ciclo non aggiungendo alcun arco nella conclusione.

Soluzione Esercizio 1.14

Si considerino i dati mostrati nella Figura 1.11(a), dove si assume che le distanze non esplicitate siano tutte uguali a 5. La soluzione π^* illustrata nella Figura 1.11(b) ha costo $c(\pi*) = 11$ ed è un ottimo globale. La soluzione π della Figura 1.11(c) ha costo $c(\pi) = 12$ ed è un ottimo locale, come si può verificare considerando i costi di tutte le soluzioni $\pi' \in I_2(\pi)$ mostrate nella Figura 1.11(d). L'ottimo globale π^* non appartiene a $I_2(\pi)$, poiché non è ottenibile da π scambiando due archi, ma scambiandone tre.

Soluzione Esercizio 1.16

Dato che una soluzione ammissibile S per il problema consiste in una partizione dell'insieme A in due parti, un intorno $I(S)$ si può definire come l'insieme di tutte le soluzioni S' ottenibili da S o spostando un elemento da una parte all'altra o scambiando tra loro due elementi tra le due

parti. La nuova soluzione S' che sostituisce S si sceglie tra quelle che diminuiscono lo scarto tra la somma degli elementi che stanno in una parte e la somma di quelli che stanno nell'altra parte. La cardinalità di $I(S)$ è $O(n^2)$. Per ciascuna mossa possibile (spostamento o scambio), la differenza tra gli scarti delle soluzioni S ed S' si può calcolare in tempo costante. Pertanto ogni iterazione di ricerca locale costa tempo $O(n^2)$.

Soluzione Esercizio 1.18

La semplice euristica seguente, nota come *star alignment*, è molto usata in biologia. Si risolvono tutti i problemi di allineamento globale tra tutte le coppie di stringhe usando $\frac{k(k-1)}{2}$ volte l'algoritmo di programmazione dinamica visto nella soluzione dell'Esercizio 13.7. Sia c_{ij} il punteggio ottenuto per la coppia di stringhe P_i e P_j. Si calcola $s_i = \sum_{j=1}^{k} c_{ij}$, per ogni $1 \leq i \leq k$ e si seleziona $s_h = \max_{1 \leq i \leq k}\{s_i\}$. Si sceglie come soluzione dell'allineamento globale multiplo quella data dai $k-1$ allineamenti globali ottenuti tra P_h e ciascuna delle altre stringhe. Tale euristica ha complessità $O(k^2n^2)$, dove n è la lunghezza della stringa più lunga tra $P_1, \ldots, P_k$.

Soluzione Esercizio 1.19

Descriviamo una euristica molto nota, chiamata *saturation degree*. Si assuma che i colori siano indicati con interi positivi. L'euristica colora un nodo u alla volta, assegnandogli il primo colore disponibile (cioè l'intero più piccolo) che non è stato ancora assegnato ad alcun nodo del suo insieme di adiacenza $\mathsf{adj}(u)$. Ad ogni passo, si colora il nodo u per il quale è massimo il numero di colori distinti già assegnati ai nodi in $\mathsf{adj}(u)$. In caso di parità tra più nodi, si colora il nodo u per il quale è massimo il numero di nodi già colorati in $\mathsf{adj}(u)$. L'idea di base è che questi nodi hanno una minore possibilità di scelta del colore e quindi un maggior rischio che tutti i colori disponibili siano assegnati ai nodi adiacenti nei passi successivi. In caso di ulteriore parità tra più nodi, se ne colora uno di questi scelto in modo casuale. Di solito, l'euristica è rieseguita un prefissato numero di volte. Infatti ad ogni esecuzione le scelte casuali che risolvono le situazioni di parità saranno presumibilmente diverse, e quindi in genere si otterranno colorazioni diverse, tra le quali viene selezionata quella che usa il minor numero di colori.