Capitolo 19

Soluzioni per problemi intrattabili

Chi si accontenta, gode

Proverbio

19.1 Algoritmi pseudo-polinomiali

Non si può avere tutto dalla vita; bisogna rinunciare a qualcosa:

- **generalità**: algoritmi pseudo-polinomiali che funzionano per solo alcuni casi particolari dell'input;
- **ottimalità**: algoritmi di approssimazione, che garantiscono di ottenere soluzioni "vicine" alla soluzione ottimale;
- **formalità**: algoritmi euristici, di solito basati su tecniche *greedy* o di ricerca locale, che forniscano sperimentalmente risultati buoni;
- **efficienza**: algoritmi esponenziali branch-&-bound, che limitano lo spazio di ricerca con un'accurata potatura.

Per il rilassamento di ognuno di questi vincoli possiamo definire un tecnica.

19.1.1 Somma di sottoinsiemi SUBSET-SUM

Definizione del problema Dati un insieme $A = \{a_1, a_2, \ldots, a_n\}$ di interi positivi e un intero positivo k, esiste un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} A_i = k$?

Utilizzando *backtracking*, abbiamo risolto la versione di ricerca di questo problema. Quella appena enunciata è la versione decisionale. Per semplificare il confronto, ci concentriamo sulla seconda.

Somma di sottoinsiemi risolto tramite programmazione dinamica

Definiamo una tabella booleana $DP[0 \ldots n][0 \ldots k]$. $DP[i][r]$ è uguale a **true** se esiste un sottoinsieme dei primi i valori memorizzati in A la cui somma è pari a r, **false** altrimenti. Il problema generale è definito da $DP[n][k]$.

Definizione dell'equazione di ricorrenza Analizziamo caso per caso. Prima i casi base:

① se devo ottenere un valore uguale a 0 ($r = 0$), sono sicuro che non prendendo nessun oggetto posso ottenere quel valore, quindi restituiremo **true**;

② nel caso in cui voglia ottenere un valore ($r > 0$), ma non ho più alcun oggetto da sommare ($i = 0$), allora non mi sarà possibile soddisfare la richiesta, restituirò **false**.

Consideriamo ora i casi in cui vogliamo ottenere un valore ($r > 0$) e abbiamo ancora oggetti a nostra disposizione ($i > 0$).

③ il valore dell'oggetto considerato è troppo grande rispetto al valore r che voglio ottenere ($A[i] > r$), quindi escludo quell'oggetto ($i - 1$) e lascio inalterato il valore r;

④ posso prendere anche l'ultimo oggetto in quanto il suo valore non eccede r ($A[i] \leqslant r$), quindi considero la possibilità di non prenderlo e di lasciare inalterato il valore di r ($DP[i-1][r]$), oppure lo prendo e sottraggo il suo valore a quello che voglio ottenere ($DP[i-1][r - A[i]]$).

Siamo quindi riusciti a definire l'equazione di ricorrenza come segue

$$DP[i][r] = \begin{cases} \textbf{true} & r = 0 \\ \textbf{false} & r > 0 \wedge i = 0 \\ DP[i-1][r] & r > 0 \wedge i > 0 \wedge A[i] > r \\ DP[i-1][r] \textbf{ or } DP[i-1][r - A[i]] & r > 0 \wedge i > 0 \wedge A[i] \leqslant r \end{cases}$$

L'algoritmo sfrutta l'equazione di ricorrenza.

Algoritmo 1: Somma di sottoinsiemi risolto tramite programmazione dinami

```
boolean subSetSum(int[] A, int n, int k)
    boolean[][] DP ← new boolean[0...n][0...k] = {false}
    // CASI BASE
    // se il valore da ottenere è 0, non ho bisogno di selezionare nessun ind
    for i ← 0 until n do // primacolonna
        DP[i][0] ← true // r = 0
    // se non ho nessun intero positivo da selezionare non mi è possibile arr
    for r ← 1 until k do // primariga
        DP[0][r] ← false // r > 0 ∧ i = 0
    // CASO RICORSIVO
    from i ← 1 until n do
        from r ← 1 until A[i] − 1 do // commento
            DP[i][r] ← DP[i − 1][r] // A[i] > r
        from r ← A[i] until k do // commento
            DP[i][r] ← DP[i − 1][r] or DP[i − 1][r − A[i]] // A[i] ⩽ r
    // restituisco il valore in posizione k-esima
    return DP[n][k]
```

Complessità Essendo un problema decisionale, è possibile semplificare e utilizzare spazio $\Theta(k)$, invece che $\Theta(nk)$, in quanto avrò n righe e k colonne.

Esempio di esecuzione Ad esempio prendendo l'insieme $A = [5, 9, 10]$ di interi positivi e l'intero positivo $k = 24$ proviamo a riempire la tabella di programmazione dinamica. Dove le righe rappresentano il numero di oggetti presi e le colonne il valore di k desiderato.

	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**	A
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**5**
2	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**9**
3	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	0	0	1	**10**

Analisi della complessità La complessità dell'algoritmo è $\Theta(nk)$, ma la complessità dei dati in ingresso è $\mathcal{O}(n \log k)$, in quanto i valori più grandi del nostro obiettivo possono essere esclusi. Se k è $\mathcal{O}(n^c)$ con c costante, allora subSetSum ha complessità polinomiale $\mathcal{O}(n^{c+1})$. Ma se k è $\mathcal{O}(2^n)$, allora subSetSum ha complessità superpolinomiale $\mathcal{O}(n \cdot 2^n)$.

Osservazione. La complessità di subSetSum dipende quindi dai valori contenuti nell'insieme e non soltanto dalla cardinalità dei dati in ingresso (n).

Somma di sottoinsiemi risolto tramite backtracking

Algoritmo 2: Somma di sottoinsiemi risolto tramite backtracking

```
boolean ssRec(Int[] A, int i, int r)
    if r == 0 then
        return true
    else if i == 0 then
        return false
    else if A[i] > r then
        return ssRec(A, i − 1, r)
    else
        return ssRec(A, i − 1, r) or ssRec(A, i − 1, r − A[i])
    return res
```

Esempio di esecuzione Prendiamo sempre in considerazione l'insieme $A = [5, 9, 10]$ di interi positivi e l'intero positivo $k = 24$. In questo caso non viene effettivamente calcolata una tabella, in quanto sono tutte chiamate ricorsive, ma viene riportata per comodità. Dove le righe rappresentano il numero di oggetti presi i e le colonne il valore di r. Ad esempio per calcolare la chiamata ($i = 3$, $r = 24$) vengono calcolati ricorsivamente il caso precedente ($i - 1 = 3 - 1 = 2, r = 24$) e il caso ($i - 1 = 3 - 1 = 2$, $r - A[i] = 24 - 10 = 14$), ricorsivamente per calcolare ($i = 2, r = 24$) viene effettuata la chiamata ricorsiva per calcolare il caso precedente ($i - 1 = 2 - 1 = 1$, $r = 14$) e il caso ($i - 1 = 2 - 1 = 1$, $r = r - A[i] = 14 - 9 = 5$).

	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**	A
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**5**
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**9**
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**10**

Complessità Ognuno dei valori è stato calcolato tramite chiamate ricorsive. Una chiamata ricorsiva quando $i = 3$, due chiamate ricorsive quando $i = 2$, quattro chiamate ricorsive quando $i = 1$ ed infine otto chiamate quando $i = 0$. Il che è ovvio visto che abbiamo usato un algoritmo esponenziale e quindi raddoppio le chiamate ad ogni passo. Per una complessità totale di $\mathcal{O}(2^n)$.

Somma di sottoinsiemi risolto tramite memoization

Algoritmo 3: Somma di sottoinsiemi risolto tramite memoization

```
boolean ssRec(int[] A, int i, int r, DICTIONARY DP)
    // CASI BASE
    if r == 0 then
        return true
    else if r < 0 or i == 0 then
        return false
    else // CASI RICORSIVI
        boolean result ← DP.lookup((i, r)) // uso come chiave la descrizione del problema
        if res == nil then // se il problema non è stato ancora calcolato
            result ← ssRec(A, i − 1, r, DP) // calcolo il caso precedente
            if A[i] < r then // se posso prenderlo prendo in considerazione la possibilità di farlo
                result ← result or ssRec(A, i − 1, r − A[i], DP)
            DP.insert((i, r), res) // inserisco la soluzione nel dizionario
        return result
```

Complessità Il dizionario viene rappresentano come una tabella, ma potrebbe essere una "hashtable".

Se proviamo a riproviamo ad applicare questo algoritmo sull'esempio precedente, le caselle del dizionario che verranno riempite sono solo quelle contenenti i problemi da risolvere e non tutti. Nel caso ci siano molti problemi ripetuti il caso pessimo è rappresentato dalla complessità $\mathcal{O}(nk)$, ossia il caso in cui dobbiamo per forza riempire tutto il dizionario.

	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**	A
0	1					0				0	0				0	0									0	
1						1									0	0									0	**5**
2															1										0	**9**
3																									1	**10**

Se prendiamo invece in considerazione un caso particolare, in cui l'insieme di interi positivi è $A = [1, 1, 1, 1, 1]$ e l'intero positivo è $k = 5$, e proviamo a riempire la tabella di memoization, notiamo che è necessario riempire gran parte del dizionario. Dovendo riempire 2^n caselle.

	0	**1**	**2**	**3**	**4**	**5**	A
0	1	0	0	0	0	0	
1		1	0	0	0	0	**1**
2			1	0	0	0	**1**
3				1	0	0	**1**
4					1	0	**1**
5						1	**1**

Quindi nel caso sfortunato abbiamo una complessità esponenziale di $\mathcal{O}(2^n)$ (come nel caso del backtracking), mentre nel caso riuscissimo a sfruttare la proprietà di memoization otteniamo una complessità di $\mathcal{O}(nk)$. Entrambi i limiti valgono si considera il caso pessimo.

Discussione sulla complessità di somma di sottoinsiemi

Riassumendo abbiamo risolto il problema tramite la programmazione dinamica ottenendo una complessità di $\Theta(nk)$, tramite backtracking ottenendo una complessità di $\mathcal{O}(2^n)$ ed infine tramite memoization ottenendo una complessità di $\mathcal{O}(nk)$ nel caso migliore e di $\mathcal{O}(2^n)$ nel caso pessimo.

Ma $\mathcal{O}(nk)$ è una complessità superpolinomiale? No, non lo è, infatti k è parte dell'input, non una dimensione dell'input. k viene rappresentato da $t = \lceil \log k \rceil$ cifre binarie. Quindi la complessità è $\mathcal{O}(nk) = \mathcal{O}(n \cdot 2^t)$, esponenziale.

Tabella 19.1: Di seguito sono riportate le complessità dei vari algoritmi per la risoluzione del problema di somma di sottoinsiemi. Notiamo che nell'applicazione dell'algoritmo di programmazione dinamica, i valori k contenuti nell'insieme di input influenzano la complessità dell'algoritmo; mentre nell'applicazione dell'algoritmo di memoization i valori fanno parte dell'input, quindi ciò che influenza la complessità è la dimensione della rappresentazione dei dati dell'insieme, non i valori contenuti in esso. Infine nell'applicazione della tecnica di backtracking la complessità non è influenzata dai dati in ingresso.

SUBSET-SUM				
Tecnica	*Algoritmo*	*Input*	*Complessità*	
Programmazione Dinamica	$\Theta(nk)$	$k = \mathcal{O}(n^c)$	$\mathcal{O}(n^{c+1})$	polinomiale
		$k = \mathcal{O}(2^n)$	$\mathcal{O}(n \cdot 2^n)$	superpolinomiale
Backtracking	$\mathcal{O}(2^n)$	-	$\mathcal{O}(2^n)$	esponenziale
Memoization con dizionario	$\mathcal{O}(nk)$	$t = \lceil \log k \rceil$	$\mathcal{O}(n \cdot 2^t)$	esponenziale

SUBSET-SUM fa parte della famiglia dei problemi $\mathbb{NP}$-completi.

19.1.2 Problemi fortemente, debolmente $\mathbb{NP}$-completi

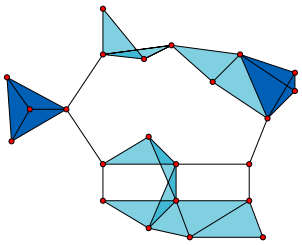
Definizione 19.1.1 (dimensioni del problema)**.** Dato un problema decisionale R e una sua istanza I. La dimensione d di I è la lunghezza della stringa che codifica I. Il valore # è il più grande numero intero che appare in I.

Tabella 19.2: Problemi decisionali e relative grandezze

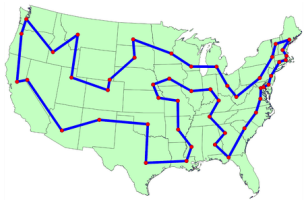
Nome	**Istanza** I	**No. più grande #**	**Dimensione** d
SUBSET-SUM	$\{n, k, A\}$	$\max\{n, k, \max(A)\}$	$\mathcal{O}(n \log \#)$
CLIQUE	$\{n, m, k, G\}$	$\max\{n, m, k\}$	$\mathcal{O}(n + m + \log \#)$
TSP	$\{n, k, d\}$	$\max\{n, k, \max(d)\}$	$\mathcal{O}(n^2 \log \#)$

Un'istanza del problema SUBSET-SUM viene rappresentato dalla grandezza del vettore n, dal numero k che voglio ottenere e dal vettore A; il più grande numero intero che appare nel problema viene determinato dal massimo fra n, k, e il massimo valore presente nel vettore; per memorizzare il problema abbiamo bisogno di n per $\log n$ bit.

Un'istanza del problema CLIQUE viene rappresentato dagli n ed m archi, dal sottoinsieme di k nodi tutti mutualmente adicenti che siamo interessati a trovare e dal grafo G; il più grande numero intero che appare nel problema viene determinato dal massimo fra n, m e k visto che non ci sono pesi associati agli archi; per rappresentare questo problema abbiamo bisogno di memorizzare il grafo, che possiamo rappresentare tramite vettori o liste di adiacenza in spazio $n+m$, e in più abbiamo bisogno di memorizzare il numero più grande presente nel problema.



Un'istanza del problema TSP viene rappresentato dalle n città, da k e dalla matrice simmetrica d di distanze positive; il più grande numero intero che appare nel problema viene determinato dal massimo fra n, k e dal massimo valore contenuto nella matrice; per memorizzare questo problema abbiamo bisogno di rappresentare la matrice in n^2, e in più abbiamo bisogno di memorizzare il numero più grande presente nel problema.



Definizione 19.1.2 (problema fortemente $\mathbb{NP}$-completo)**.** Sia R_{pol} il problema R ristretto a quei dati d'ingresso per i quali il più grande valore da rappresentare è limitato superiormente da $p(d)$, con p funzione polinomiale in d (d è la dimensione del mio input). R è fortemente $\mathbb{NP}$-completo se R_{pol} è $\mathbb{NP}$-completo.

In altre parole, dato un problema che sappiamo essere $\mathbb{NP}$-completo, prendi il problema ristretto in cui metto dei limiti ai valori dei dati in input, questi limiti devono essere polinomiali nella dimensione del problema. Se comunque il problema resto $\mathbb{NP}$-completo allora diremo che il problema è più che $\mathbb{NP}$-completo, è fortemente $\mathbb{NP}$-completo.

Se invece il problema non risulta più $\mathbb{NP}$-completo, e quindi finisce nello spazio $\mathbb{P}$, allora si dice debolmente $\mathbb{NP}$-completo.

Definizione 19.1.3 (problema debolmente $\mathbb{NP}$-completo)**.** Se un problema $\mathbb{NP}$-completo non è fortemente $\mathbb{NP}$-completo, allora è debolmente $\mathbb{NP}$-completo.

Osservazione. Il problema cricca (CLIQUE), ad esempio, è fortemente $\mathbb{NP}$-completo, mentre quello della somma di sottoinsieme SUBSET-SUM non lo è.

19.1.3 Esempio di problema debolmente $\mathbb{NP}$-completo

Sappiamo che il problema SUBSET-SUM è $\mathbb{NP}$-completo.

Somma di sottoinsiemi (SUBSET-SUM)

Definizione del problema Dati un vettore A contenente n interi positivi ed un intero positivo k, esiste un sottoinsieme $S \subseteq \{1 \dots n\}$ tale che $\sum_{i \in S} a[i] = k$?

Dimostrazione. Somma di sottoinsieme è debolmente $\mathbb{NP}$-completo. Possiamo imporre come limite che $\forall A[i] \leqslant k$ (valori più grandi di k vanno esclusi). Se $k = \mathcal{O}(n^c)$, allora $\# = \max\{n, k, a_1, \dots, a_n\} = \mathcal{O}(n^c)$. La soluzione basata su programmazione dinamica ha complessità $\mathcal{O}(nk) = \mathcal{O}(n^{c+1})$, quindi in $\mathbb{P}$. Possiamo dedurne che SUBSET-SUM non è fortemente $\mathbb{NP}$-completo. □

Nota. Il problema di SUBSET-SUM è debolmente $\mathbb{NP}$-completo: limitando le dimensioni dei dati in ingresso il problema diventa polinomiale.

19.1.4 Algoritmi pseudo-polinomiali

Definizione (Algoritmo pseudo-polinomiale)**.** Un algoritmo che risolve un certo problema R, per qualsiasi dato I d'ingresso, in tempo $p(\#, d)$, con p funzione polinomiale in $\#$ e d, ha complessità pseudo-polinomiale.

Osservazione. Gli algoritmi per SUBSET-SUM basati su programmazione dinamica e memoization sono pseudo-polinomiali.

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

19.1.5 Esempi di problemi fortemente $\mathbb{NP}$-completi

Cricca (CLIQUE)

Definizione del problema Dati un grafo non orientato ed un intero k, esiste un sottoinsieme di almeno k nodi tutti mutuamente adiacenti?

Dimostrazione. CLIQUE è fortemente $\mathbb{NP}$-completo. Possiamo porre come limite $k \leqslant n$, in quanto non ha senso chiedersi se esiste in un grafo con k nodi se esiste un sottografo con n nodi con $n > k$, la risposta in quel caso è **false**. Quindi il valore massimo dei dati in ingresso $\# = \max\{n, m, k\}$, in quanto $k \leqslant n$ possiamo semplificare $\# = \max\{n, m\}$. La dimensione del problema è data da $d = \mathcal{O}(n + m + \log \#)$, ma visto che sappiamo che $\# = \max\{n, m\}$, allora possiamo semplificare e otteniamo $d = \mathcal{O}(n + m)$. Quindi $\# = \max\{n, m\}$ è limitato superiormente da $\mathcal{O}(n + m)$ qualunque siano i dati di ingresso. Possiamo dedurne che il problema ristretto è identico a CLIQUE, che è $\mathbb{NP}$-completo. □

Nota. Il problema CLIQUE è fortemente $\mathbb{NP}$-completo.

Commesso viaggiatore (TSP)

Definizione del problema Date n città e una matrice simmetrica d di distanze positive, dove $d[i][j]$ è la distanza fra i e j, 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 totale percorsa sia minima.

Dimostrazione per assurdo che TSP è fortemente $\mathbb{NP}$-completo. Per assurdo, supponiamo che TSP sia debolmente $\mathbb{NP}$-completo. Allora esiste una soluzione pseudo-polinomiale. Usiamo questa soluzione per risolvere un problema $\mathbb{NP}$-completo in tempo polinomiale, il che è assurdo a meno che $\mathbb{P} = \mathbb{NP}$. □

Un problema se non ha valori numerici molto probabilmente risulta $\mathbb{NP}$-completo, mentre se li ha potrebbe essere o non essere debolmente $\mathbb{NP}$-completo. Questo è un esempio di un problema che non lo è.

Nota. Il problema TSP è fortemente $\mathbb{NP}$-completo nonostante abbia valori numerici.

Circuito hamiltoniano (HAMILTONIAN-CIRCUIT)

Definizione del problema Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?

Questo problema è fortemente collegato al problema del commesso viaggiatore, in quanto un tour del commesso viaggiatore è un circuito hamiltoniano.

Complessità Il problema HAMILTONIAN-CIRCUIT è $\mathbb{NP}$-completo. É uno dei 21 problemi elencati nell'articolo di Karp.

Dimostriamo che TSP è fortemente $\mathbb{NP}$-completo. Sia $G = (V, E)$ un grafo non orientato. Definiamo una matrice di distanze a partire da G.

$$d[i][j] = \begin{cases} 1 & (i,j) \in E \\ 2 & (i,j) \notin E \end{cases}$$

Il grafo G ha un circuito hamiltoniano se e solo se è possibile trovare un percorso da commesso viaggiatore di costo n.

Simmetria fra TSP e HAMILTONIAN-CIRCUIT Se esistesse un algoritmo pseudopolinomiale A per TSP, HAMILTONIAN-CIRCUIT potrebbe essere risolto da A in tempo polinomiale. La riduzione consiste nel prendere il circuito hamiltoniano dove il k cercato nel circuito hamiltoniano è pari alle n città. □

Nota. Non sempre è semplice identificare immediatamente se un problema numerico risulti fortemente o debolmente $\mathbb{NP}$-completo.

Partizione (PARTITION)

Definizione del problema Dato un vettore A contenente n interi positivi, esiste un sottoinsieme $S \subseteq \{1 \dots n\}$ tale che $\sum_{i \in S} A[i] = \sum_{i \in S} A[i]$?

0	1	2	3	4	5
14	6	12	3	7	2

Ad esempio prendendo in considerazione questo vettore possiamo prendere gli insiemi $\{14, 6, 2\}$ e $\{12, 3, 7\}$.

Conclusioni É possibile ridurre il problema PARTITION a SUBSET-SUM scegliendo come valore k la metà di tutti i valori presenti:

$$k = \frac{\sum_{i=1}^{n} A[i]}{2} = \frac{44}{2} = 22$$

Nota. Il problema PARTITION è debolmente $\mathbb{NP}$-completo.

3-Partizione (3-PARTITION)

Dati $3n$ interi $\{a_1, \dots, a_{3n}\}$ esiste una partizione in n triple $T_1, \dots, T_n$, tale che la somma dei tre elementi di ogni T_j sia la stessa, per $1 \leqslant j \leqslant n$?

Nota. Il problema 3-PARTITION è fortemente $\mathbb{NP}$-completo: non esiste un algoritmo pseudopolinomiale per risolverlo.

19.2 Algoritmi di approssimazione

Facciamo una piccola premessa. I problemi più interessanti sono in forma di ottimizzazione. Se il problema di decisione è $\mathbb{NP}$-completo, non sono noti algoritmi polinomiali per il problema di ottimizzazione. Esistono però algoritmi polinomiali che trovano soluzioni ammissibili più o meno vicine a quella ottima.

Definizione (Algoritmi di approssimazione)**.** Se è possibile dimostrare un limite superiore/inferiore al rapporto fra la soluzione trovata e la soluzione ottima, allora tali algoritmi vengono detti algoritmi di approssimazione.

Approssimazione

Definizione ($\alpha(n)-$*approssimazione*)**.** Dato un problema di ottimizzazione con funzione costo non negativa c, un algoritmo si dice di $\alpha(n)$-approssimazione se fornisce una soluzione ammissibile x il cui costo $c(x)$ non si discosta dal costo $c(x^*)$ della soluzione ottima x^* per più di un fattore $\alpha(n)$, per qualunque input di dimensione n:

$$\begin{aligned} c(x^*) \leqslant c(x) \leqslant \alpha(n)c(x^*) \quad & \alpha(n) > 1 \quad \text{(Minimizzazione)} \\ \alpha(n)c(x^*) \leqslant c(x) \leqslant c(x^*) \quad & \alpha(n) < 1 \quad \text{(Massimizzazione)} \end{aligned}$$

In altre parole se si sta cercando di minimizzare

Osservazione. $\alpha(n)$ può essere una costante, valida per tutti gli n.

Nota. Identificare un valore $\alpha(n)$ e dimostrare che l'algoritmo lo rispetta è ciò che rende un buon algoritmo un algoritmo di approssimazione.

19.2.1 Bin packing

Definizione del problema Dati un vettore A contenente n interi positivi (i volumi degli oggetti) e un intero positivo k (la capacità di una scatola, tale che $\forall i : A[i] \leqslant k$), si vuole trovare una partizione di $\{1, \ldots, n\}$ nel minimo numero di sottoinsiemi disgiunti ("scatole") tali che $\sum_{i \in S} A[i] \leqslant k$ per ogni insieme S della partizione.

Dato il seguente vettore e un intero $k = 8$ come risolvereste il problema?

0	1	2	3	4	5	6
3	7	2	5	4	3	5

Appoccio ingordo per bin packing

Algoritmo first-fit Gli oggetti sono considerati in un ordine qualsiasi e ciascun oggetto è assegnato alla prima scatola che lo può contenere, tenuto conto di quanto spazio è stato occupato della stessa.

3,2,3	7	5	4	5

In questo particolare caso si è comportato in maniera ottimale.

Sia $N > 1$ il numero di scatole usare da FIRST-FIT (se $N = 1$, l'algoritmo è ottimale). Il numero minimo di scatole N^* è limitato da:

$$N^* \geqslant \left\lceil \frac{\sum_{i=1}^{n} A[i]}{k} \right\rceil = \left\lceil \frac{29}{8} \right\rceil = \lceil 3.625 \rceil = 4$$

non possono esserci due scatole riempite meno della metà:

$$N < \frac{\sum_{i=1}^{n} A[i]}{k/2} = \frac{29}{8/2} = 7.250$$

abbiamo quindi:

$$N < \frac{\sum_{i=1}^{n} A[i]}{k/2} = 2\frac{\sum_{i=1}^{n} A[i]}{k} \leqslant 2N^* = \alpha(n)N^*$$

che implica $\alpha(n) = 2$.

Nota. Nel peggiore dei casi l'algoritmo First-Fit utilizza il doppio delle scatole.

Algoritmo first-fit decreasing Se consideriamo gli oggetti in ordine non descrescente è possibile dimostrare un risultato migliore per l'algoritmo First-Fit.

$$N < \frac{17}{10}N^* + 2$$

Nella variante FFD (First-fit decreasing): gli oggetti sono considerati in ordine non decrescente

$$N < \frac{11}{9}N^* + 4$$

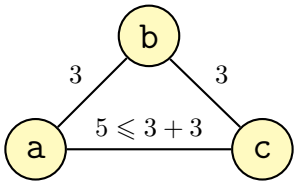
Nota. Queste sono dimostrazioni di limiti superiori per il fattore $\alpha(n)$, per casi particolari l'approssimazione può essere migliore.

19.2.2 Commesso viaggiatore con disuguaglianze triangolari (Δ-TSP)

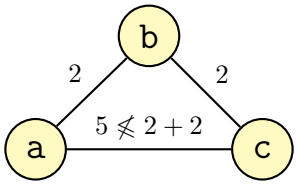
Definizione del problema Siano date n città e le distanze (positive) $d[i][j]$ tra esse, tale per cui vale la regola delle diseguaglianze triangolari:

$$d[i][j] \leqslant d[i][k] + d[k][j] \quad \forall i, j, k : \quad 1 \leqslant i, j, k \leqslant n$$

Trovare un percorso che, partendo da una qualsiasi città, attraversi ogni città esattamente una volte e ritorni alla città di partenza, in modo che la distanza complessiva percorsa sia minima.



(a) con diseguaglianza triangolare



(b) senza diseguaglianza triangolare

Nota. Δ-TSP è $\mathbb{NP}$-completo.

Dimostriamo che HAMILTONIAN-CIRCUIT $\leqslant_p$ Δ-TSP. Sia $G = (V, E)$ un grafo non orientato. Definiamo una matrice delle distanze a partire da G

$$d[i][j] = \begin{cases} 1 & (i,j) \in E \\ 2 & (i,j) \notin E \end{cases}$$

Abbiamo applicato la stessa riduzione che avevamo applicato per il problema TSP. Il grafo G ha un circuito hamiltoniano se e solo se è possibile trovare un percorso da commesso viaggiatore lungo n. Valgono le diseguaglianze triangolari:

$$d[i][j] \leqslant 2 \leqslant d[i][k] + d[k][j]$$ □

Questo particolare grafo gode di disuguaglianza triangolare perché $d[i][j]$ può avere valore massimo 2, mentre $d[i][k]$ e $d[k][j]$ hanno come valori minimi 1.

Nota. Se esistesse una soluzione polinomiale (in $\mathbb{P}$) per Δ-TSP, allora esisterebbe anche una soluzione per HAMILTONIAN-CIRCUIT cosa che non riteniamo possibile.

Commesso viaggiatore come circuito hamiltoniano pesato

Interpretiamo Δ-TSP come il problema di trovare un circuito hamiltoniano di peso minimo su un grafo completo.

	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	

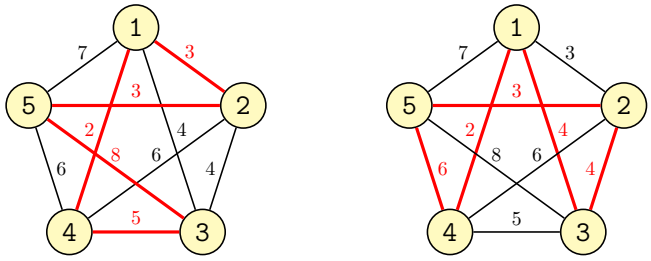
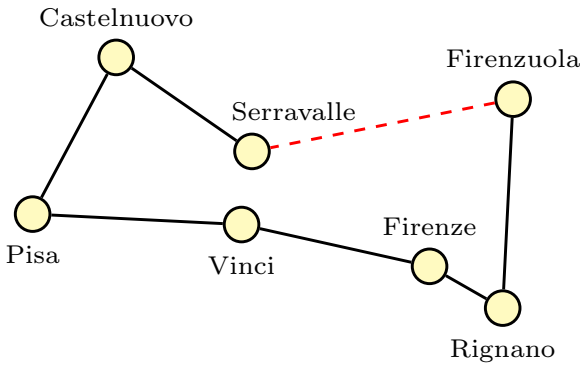


Figura 19.2: Il costo totale risulta 21. Il costo totale risulta 19.

Algoritmo di approssimazione per Δ-TSP

Se si considera un circuito hamiltoniano e si cancella un suo arco, si ottiene un albero di copertura.



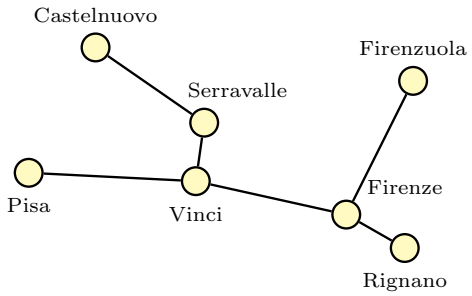
Che relazione hanno gli alberi di copertura con i circuiti hamiltoniani?

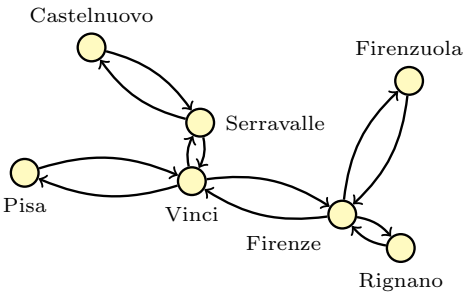
Teorema. *Qualunque circuito hamiltoniano π, ha costo $c(\pi)$ superiore al costo mst di albero di copertura di peso minimo, ovvero $mst < c(\pi)$.*

Dimostrazione per assurdo. Supponiamo che esista un circuito hamiltoniano π di costo $c(\pi) \leqslant mst$. Togliamo un arco, otteniamo un albero di copertura con peso inferiore $mst' < c(\pi) \leqslant mst$. Contraddizione, visto che mst è il costo minimo fra tutti gli alberi di copertura. □

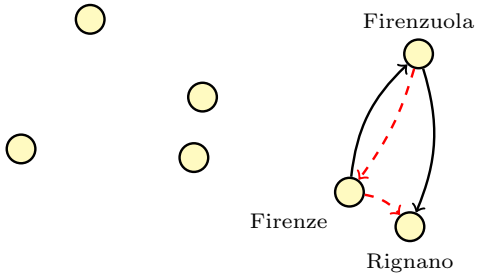
Algoritmo per Δ-TSP

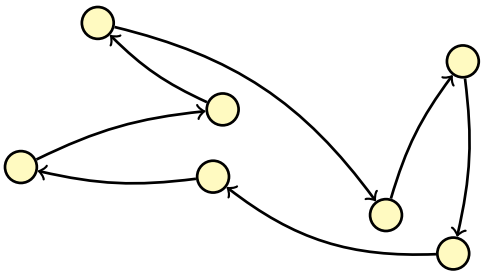
Si individua un minimo albero di copertura di peso mst e se ne percorrono gli archi due volte, prima in un senso e poi nell'altro. In questo modo, si visita ogni città almeno una volta. La distanza complessiva di tale circuito è uguale a $2 \cdot mst$. Ma non è un circuito hamiltoniano!





Si evita di passare per città già visitate, saltandole. Per la diseguaglianza triangolare, il costo $c(\pi)$ del circuito così ottenuto è inferiore o uguale a $2 \cdot mst$. Concludendo, $c(\pi) \leqslant 2 \cdot mst < 2 \cdot 2 \cdot c(\pi^*)$ implica che $\alpha(n) = 2$, dove $c(\pi^*)$ è il costo del circuito hamiltoniano ottimo.





Discussione sulla complessità dell'algoritmo per Δ-TSP

La complessità dell'algoritmo è pari a $\mathcal{O}(n^2 \log n)$. I fattori che contribuiscono sono:

- $\mathcal{O}(n^2 \log n)$ per l'algoritmo di Kruskal;
- $\mathcal{O}(n)$ per la visita in profondità del *mst* con $2n$ archi.

Tuttavia esistono grafi "perversi" per cui il fattore di approssimazione tende al valore 2. L'algoritmo di Christofides (1976) ha un fattore di approssimazione di 3/2, il migliore risultato al momento.

Non approssimabilità di TSP

Abbiamo visto che Δ-TSP può essere approssimato. Non esiste alcun algoritmo di $\alpha(n)$-approssimazione per TSP tale che $c(x') \leqslant sc(x^*)$, con $s \geqslant 2$ intero positivo, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

Nota. $\Delta - tsp$ è un problema approssimabile, ma il problema generale non lo è.

19.3 Algoritmi euristici

Quando si è presi dalla disperazione a causa della enorme difficoltà di un problema di ottimizzazione $\mathbb{NP}$-hard, si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile. La soluzione fornita non è necessariamente ottima, né necessariamente approssimata. Talvolta non si riesce a dimostrare che non sia possibile fare meglio di così.

Le tecniche che è possibile utilizzare in questi casi sono l'approccio ingordo e la ricerca locale.

19.3.1 Approccio ingordo per TSP

Algoritmo "Shortest edges first"

Ordiniamo gli archi per pesi non decrescenti e aggiungiamo archi alla soluzione seguendo questo ordine finché non sono stati aggiunti $n-1$ archi, dove n è il numero di nodi.

Per poter aggiungere un arco, occorre verificare che

- per ciascuno dei suoi nodi non siano stati già scelti due archi;
- che non si formino circuiti (questo lo si fa tramite l'utilizzo della struttura dati MFSET).

A questo punto, si è trovata una catena Hamiltoniana (un percorso che tocca tutti i nodi ma che non tocca il nodo di partenza). Si chiude il circuito aggiungendo l'arco tra i due nodi estremi della catena.

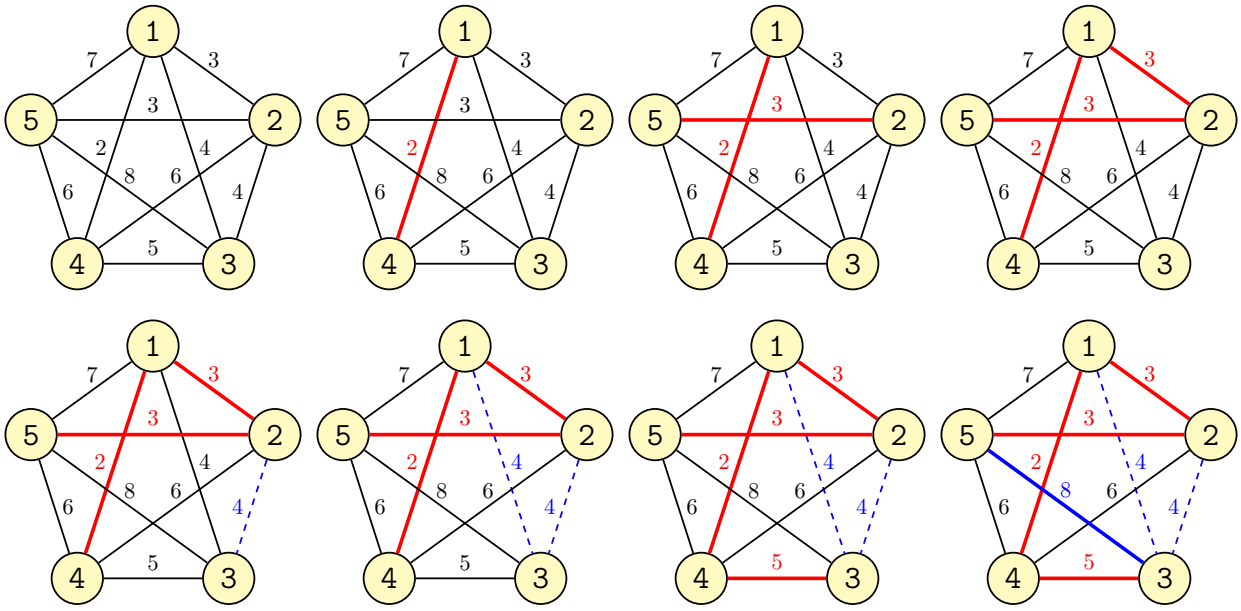


Figura 19.5: Applicazione dell'algoritmo "Shortest edges first". Questo algoritmo parte dagli archi, scegliendo sempre l'arco con peso minore. Non è possibile aggiungere i nodi con peso 4 perché i nodi che toccano hanno già due archi selezionati. Il costo totale del percorso con questo algoritmo risulta 21.

Algoritmo 4: Risoluzione del commesso viaggiatore con algoritmo ingordo

```
SET greedyTsp(GRAPH G)
    // creazione strutture dati
    SET result ← Set
    MFSET M ← Mfset(G.size)
    // archi in ingresso ad un nodo
    int[] edges ← new int[1...G.size] = { 0 } // no. archi della catena
    A ← { ordina gli archi per peso decrescente }
    // per ogni arco che appartiene all'insieme degli archi
    foreach (u, v) ∈ A do
        // se gli archi entranti in entrambi i nodi sono minori di 2 e non si è formato un circuito
        if edges[u] < 2 and edges[v] < 2 and M.find(u) ≠ M.find(v) then
            // prendo nota dell'arco inserito
            S.insert((u, v))
            // aggiorno il no. di lati entranti nei due nodi
            edges[u]++
            edges[v]++
            // li considero come un unico nodo
            M.merge(u,v)
    // chiudo il circuito
    int u ← 1
    while edges[1] ≠ 1 do u++
    int v ← u + 1
    while not edges[v] ≠ 1 do v++
    // chiusura del circuito hamiltoniano
    S.insert((u, v))
    // restituisco l'insieme che archi che costituisce il percorso
    return S
```

Algoritmo "Nearest Neighbor"

Possiamo usare un approccio diverso. Si parte da una città, si seleziona come prossima città quella più vicina (con distanza più bassa) e si va avanti così, evitando città già visitate. Quando si sono visitate tutte le città, si torna alla città di partenza.

Così facendo possiamo lavorare direttamente sulla matrice, senza usare strutture di appoggio (come il grafo dell'esempio precedente).

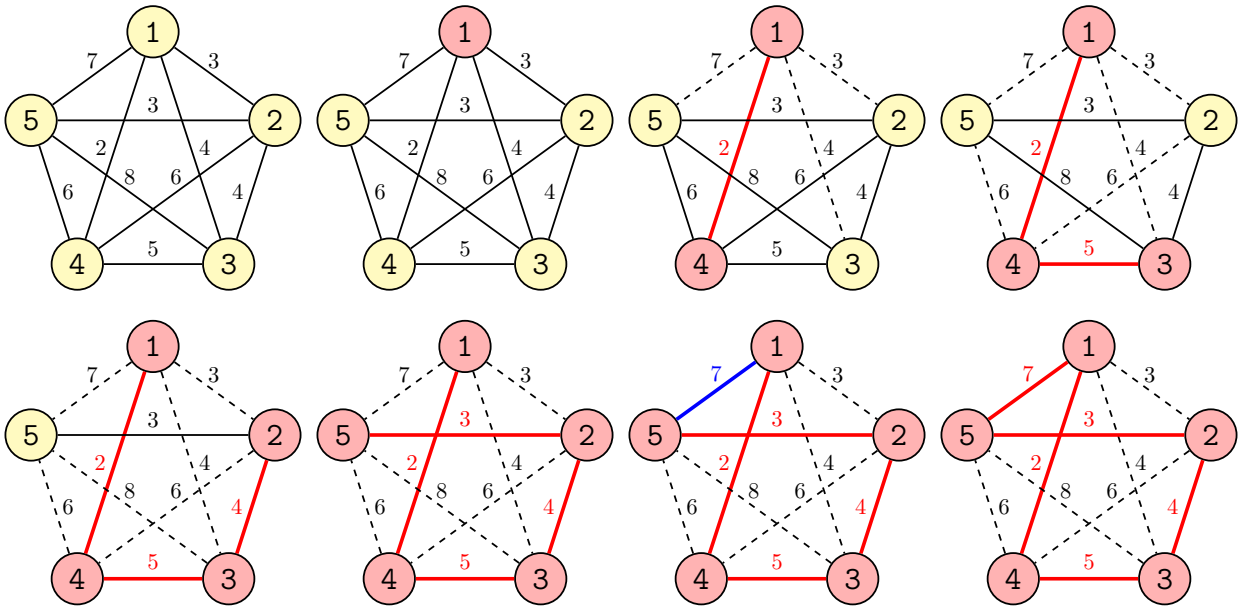


Figura 19.6: Applicazione dell'algoritmo "Nearest Neighbor". Il costo totale del percorso con questo algoritmo risulta 21, come nel caso precedente.

Discussione sulla complessità dell'approccio ingordo per TSP

Il costo computazionale dell'algoritmo "Shortest edges first" ha un costo computazionale di $\mathcal{O}(n^2 \log n)$, dove $\log n$ è dovuto all'ordinamento degli archi; mentre l'algoritmo "Nearest Neighbor" ha un costo di $\mathcal{O}(n^2)$.

La soluzione così ottenuta si può utilizzare può essere migliorata ancora tramite ricerca locale e può essere usata come base di partenza per un algoritmo branch-&-bound.

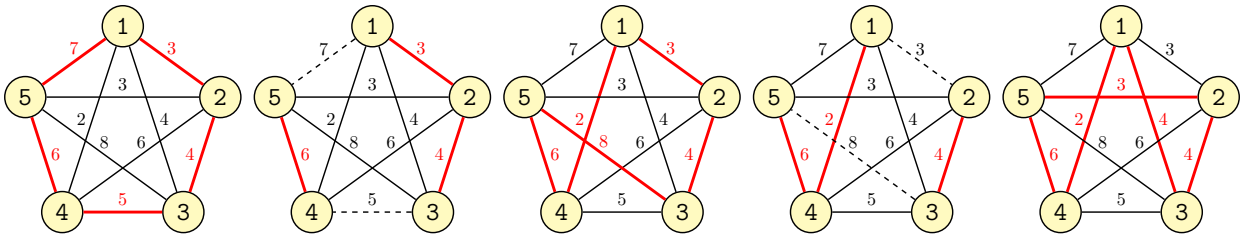
19.3.2 Approccio ricerca locale per TSP

Algoritmo di ricerca locale

Sia π un circuito Hamiltoniano del grafo completo derivante dal problema TSP. Si consideri il seguente intorno: $I_2(\pi) = \{\pi' : \pi'$ è ottenuto da π cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito$\}$

Nota. La cardinalità di $I_(\pi)$ è $|I_(\pi)| = \frac{n(n-1)}{2} - n$, in quanto ci sono $\frac{n(n-1)}{2}$ coppie di archi del circuito e n di esse sono consecutive.

Una volta spezzato il circuito, esiste un solo modo per riconnetterlo.



(a) Costo: 25. **(b)** Costo: 23. **(c)** Costo: 19.

Figura 19.7: Il costo totale del percorso con questo algoritmo risulta 19, che risulta ottimo.

Discussione sulla complessità dell'approccio di ricerca locale per TSP

Il costo per esaminare $I_2(\pi)$ è $\mathcal{O}(n^2)$.

19.4 Algoritmi branch&bound

A differenza delle tecniche di approssimazione che rinunciavano alla formalità nella risoluzione di un problema pur di ottenere un algoritmo polinomiale, per risolvere un problema di ottimizzazione $\mathbb{NP}$-arduo, si può modificare la procedura enumerazione, vista nel capitolo sul *backtracking*, in modo da "potare" certe sequenze di scelte che di rivelino incapaci di generare la soluzione ottima.

Facciamo una serie di assunzioni senza perdere troppa generalità:

- problema di ottimizzazione;
- ogni sequenza di scelte abbia costo non negativo;
- ogni scelta, aggiunta alle scelte già effettuate, non faccia diminuire il costo della soluzione parziale così ottenuta.

Algoritmo 5: Ripasso della procedura di enumerazione delle soluzioni

```
boolean enumerazione(⟨dati problema⟩, ITEM[] S, int i, ⟨dati parziali⟩)
    // S: vettore contenente le soluzioni parziali S[1][i]
    // i: indice della scelta corrente

    if isAdmissible(⟨dati problema⟩, S, i, ⟨dati parziali⟩) then
        // è una soluzione ammissibile
        if processSolution(⟨dati problema⟩, S, i, ⟨dati parziali⟩) then
            // vogliamo bloccare l'esecuzione alla prima soluzione trovata
            return true // trovata soluzione, restituisco true
    else if reject(⟨dati problema⟩, S, i, ⟨dati parziali⟩) then
        return false // impossibile trovare soluzioni, restituisco false
    else
        SET C ← scelte(⟨dati problema⟩, S, i, ⟨dati parziali⟩) // l'insieme delle scelte possibili
        foreach c ∈ C do // per ogni possibile scelta fra quelle possibili
            S[i] ← c // memorizzo la scelta parziale

            // richiamo ricorsivamente l'algoritmo effettuando la scelta successiva
            if enumerazione(⟨dati problema⟩, S, i + 1, ⟨dati parziali⟩) then
                return true // trovata soluzione, restituisco true
    return false // nessuna soluzione, restituisco false
```

Cerchiamo i limiti (superiore ed inferiore) della soluzione minima.

Limite superiore

Durante l'enumerazione, si mantengono informazioni sulla miglior soluzione ammissibile (*minSol*) ed il suo costo (*minCost*). *minCost* costituisce un limite superiore (*upper bound*) per il costo della soluzione minima.

Limite inferiore

Si supponga di avere a disposizione una opportuna funziona "lower bound" lb($\langle$*dati problema*$\rangle$, *S*, *i*, $\langle$*dati parziali*$\rangle$), che dipenda dalla sequenza di scelte fatte in precedenza $S[1 \ldots i]$, e che garantisca che tutte le soluzioni ammissibili generabili facendo nuove scelte abbiano costo $\geqslant$ lb.

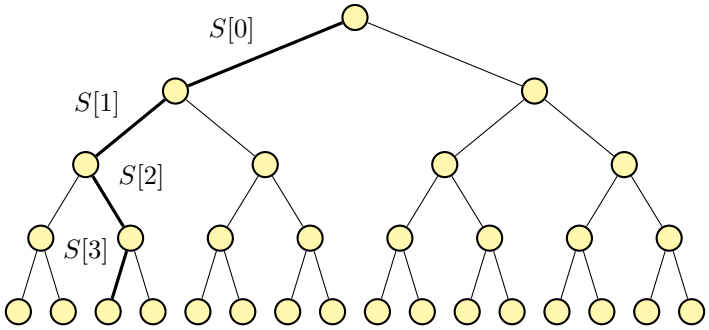


Figura 19.8: Se lb(S,i) è maggiore o uguale a *minCost*, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo, effettuando così una **potatura**.

Questo metodo non migliora la complessità (superpolinomiale) della procedura enumerazione, ma nella pratica ne abbassa di molto il tempo di esecuzione. Tutto dipende dalla funzione lb (che dobbiamo definire noi), che deve approssimare il più possibile il costo della soluzione ottima. Il limite superiore è dato dal *minCost*.

Algoritmo 6: Somma di sottoinsiemi

```
branch&bound(⟨dati problema⟩, ITEM[] S, int i, ⟨dati parziali⟩)
    SET C ← scelte(⟨dati problema⟩, n, i, ⟨dati parziali⟩)  // determina l'insieme in funzione delle
     scelte precedenti S[1...i-1]
    // esamino ogni scelta
    foreach c ∈ C do
        S[i] ← c // effettuo la scelta i-esima
        // calcolo il lower bound in base alle scelte fasse in precedenza
        int lb ← lb(⟨dati problema⟩, S, i, ⟨dati parziali⟩)
        if lb < minCost then // se il limite inferiore non eccede il costo minimo
            if i < n then // posso ancora effettuare delle scelte
                // faccio ricorsivamente le scelte successive
                branch&bound(⟨dati problema⟩, S, i+1, ⟨dati parziali⟩)
            else
                // ho effettuato tutte le scelte possibili (i==n)
                // se il costo della soluzione trovata è minore del costo della soluzione parziale
                if cost(S,i) < minCost then
                    // allora la soluzione trovata è migliore del minimo parziale
                    minSol ← S  // aggiorno la soluzione minima parziale
                    minCost ← cost(S,i)  // aggiorno il costo minimo parziale
```

19.4.1 Approccio branch-&-Bound per TSP

Sia n il numero delle città, e $d[h][k]$ la distanza, intera e non negativa, fra le città h e k. Al passo i-esimo sono state fatte le scelte $S[1 \dots i]$ prese dall'insieme $\{1, \dots, n\}$. Un percorso ammissible che "espande" $S[1 \dots i]$ deve

1) attraversare le città $S[1 \dots i]$;
2) passare da $S[i]$ ad una qualsiasi delle rimanenti $n - 1$ città;
3) attraversare queste ultime città in un ordine qualsiasi;
4) da una di queste ritornare a $S[1]$.

La distanza percorsa finora è $cost[i]$, ossia il costo che ho sostenuto per fare i primi i passi.

$$cost[i] = \begin{cases} 0 & i = 1 \\ cost[i-1] + d[S[i-1]][S[i]] & i > 1 \end{cases}$$

Viene calcolata ricorsivamente dalla sommando il costo sostenuto per arrivare alla città precedente ($cost[i-1]$) e la distanza intercorsa fra la città precedente e quella attuale.

Il limite inferiore (*lower bound*) della distanza per "uscire" da $S[i]$ (ha un costo di $O(n)$):

$$out = \min_{h \notin S} \{d[S[i], h]\}$$

Il limite inferiore (*lower bound*) della distanza per tornare a $S[1]$ (ha un costo di $O(n)$):

$$back = \min_{h \notin S} \{d[h, S[1]]\}$$

Il limite inferiore della distanza percorsa per attraversare una qualsiasi di queste ultime città h delle $n - i$ città, provenendo da (e dirigendosi verso) una città non compresa in $S[2 \dots n]$ (ha un costo di $\mathcal{O}(n^3)$).

$$\forall h \notin S : transfer = \min_{p,q \notin S[2 \dots i-1]} \{d[p, h] + d[h, q] : h \neq p \neq q\}$$

Se posso effettuare ancora delle scelte ($i < n$), è possibile calcolare un possibile limite inferiore $\mathsf{lb}(d, S, i)$ calcolando la somma:

- del costo $cost[i]$ per arrivare al nodo $S[i]$, già speso;
- metà del costo ottenuto sommando:
  - il limite inferiore *out* del costo per andare dal nodo $S[i]$ ad un qualunque altro nodo;
  - il limite inferiore per attraversare i nodi non contenuti in S;
  - il limite inferiore *back* del costo per tornare al nodo $S[i]$ da un qualunque altro nodo.

$$\mathsf{lb}(d, S, i) = cost[i] + \left\lceil \frac{out + \sum_{h \notin S} transfer[h] + back}{2} \right\rceil$$

Algoritmo 7: Appoccio branch&bound al problema del commesso viaggiatore

```
bbTsp([] S, int cost, SET R, int n, int i)
    SET choices ← copy(R)
    foreach c ∈ choices do
        S[i] ← c // memorizza la scelta
        // MECCANISMO PER GENERARE LE PERMUTAZIONI
        R.remove(c) // rimuovi la scelta nell'insieme delle scelte possibili
        // MECCANISMO DI VALUTAZIONE
        if i < n then
            { calcola out, back, transfer[h] per ogni h ∈ R ⇔ h ∉ S[1...i] }
            int lb ← ⌈(out+∑_{h∉S} transfer[h]+back)/2⌉
            if lb < minCost then
                bbTsp(S, cost + d[S[i − 1]][S[i]], R, i + 1)
        else
            lb ← cost + d[S[i]][S[1]] // aggiorna il limite inferiore
            if lb < minCost then // se abbiamo trovato una soluzione migliore
                minSol ← copy(S) // aggiorna la soluzione ottimale
                minCost ← lb // aggiorna il costo minimo
        R.insert(c) // reinserisco la scelta nell'insieme delle scelte possibili
```

Precisazione sull'algoritmo

minCost è una variabile globale. Invece di inizializzarla a $+\infty$, possiamo scegliere una permutazione a caso (un cammino qualunque), in questo modo abbiamo già una stima iniziale. Ad esempio la permutazione 1-2-3-4-5 ha un costo pari a 21. Per evitare che lo stesso circuito sia generato più volte, si parte da un nodo fissato (ad esempio 1).

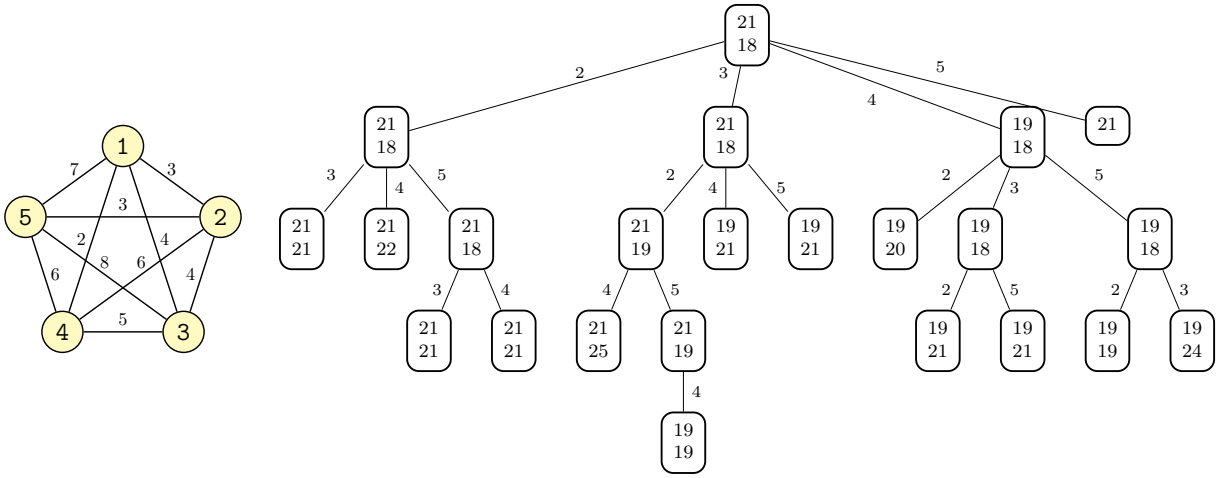


Figura 19.9: Sono indicati i limiti superiori ed inferiori.

In questo semplice esempio è stato possibile "potare" 42 nodi su 65.

Possibili miglioramenti

Sono possibili diversi miglioramenti:

- è possibile variare l'ordine di visita dell'albero delle scelte, invece di utilizzare una visita in profondità è possibile utilizzare un meccanismo "Best-first";
- è possibile variare il meccaniscmo di ramificazione (*brancing*) sui nodi, sugli archi...;
- è possibile cercare dei limiti inferiori più stretti.

19.5 Riassumendo

Abbiamo prima trattato di algoritmi pseudopolinomiali vedendo più soluzioni per il problema di somma di sottoinsiemi (SUBSET-SUM). Abbiamo visto che è possibile risolverlo utilizzando più tecniche. Tramite la programmazione dinamica abbiamo ottenuto una complessità di $\Theta(nk)$, tramite backtracking una complessità di $\mathcal{O}(2^n)$ ed infine tramite memoization una complessità di $\mathcal{O}(nk)$. Sia tramite la programmazione dinamica che la memoization, che abbiamo visto essere un miglioramento della stessa, la complessità dell'algoritmo dipende dalla dimensione dell'input.

Abbiamo poi definito i problemi fortemente e debolmente $\mathbb{NP}$-completi. Abbiamo illustrato vari problemi e abbiamo dimostrato le varie equivalenze fra di essi. Per illustrare i capitoli successivi abbiamo preso come modello il problema del commesso viaggiatore TSP.

Nell'affrontare l'argomento degli algoritmi di approssimazione abbiamo dovuto definire un problema ristretto rispetto a TSP, Δ-TSP, in quanto esso risulta non approssimabile.

Abbiamo poi affrontato il capitolo degli algoritmi euristici vedendo due approcci ingordi per TSP. L'algoritmo "Shortest edges first" ha una complessità di $\mathcal{O}(n^2 \log n)$, dove $\log n$ è dovuto all'ordinamento degli archi; mentre "Nearest Neighbor" ha un costo di $\mathcal{O}(n^2)$. Abbiamo poi migliorato ulteriolmente l'algoritmo utilizzando l'approccio di ricerca locale. Tuttavia non siamo riusciti ad ottenere un miglioramento nella complessità che è risultata comunque di $\mathcal{O}(n^2)$.

Infine abbiamo visto un'ultima tipologia di algoritmi, quelli del tipo "branch-&-bound".