19 Soluzioni per problemi intrattabili

19.1 Algoritmi pseudo-polinomiali

19.1.1 Somma di sottoinsieme

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 $\sum_{i \in S}^{a_i} = k$?

Definizione equazione di ricorrenza Definiamo una tabella booleana $DP[0 \ldots n][0 \ldots k]$. $DP[i][r]$ è uguale a **vero** se e solo se è possibile ottenere r dai primi i valori memorizzati nel vettore di input.

$$DP[i][r] = \begin{cases} \textbf{falso} & r < 0 \\ \textbf{vero} & r = 0 \\ \textbf{falso} & r > 0 \wedge i = 0 \\ DP[i-1][r] \textbf{ or } DP[i-1][r - A[i]] & r > 0 \wedge i > 0 \end{cases}$$

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

Algoritmo L'algoritmo sfrutta l'equazione di ricorrenza

```
boolean subSetSum(int[ ] A, int n, int k)
    boolean[ ] DP = new boolean[0...k]
    // casi base
    // dati nessun oggetto posso ottenere il valore 0
    DP[0][0] = vero
    per r = 1 fino a k fai
        // dati 0 oggetti posso ottenere un valore di k > 0
        DP[0][r] = falso
    // riempi la tabella
    da i = 1 fino a n fai
        da r = A[i] fino a k fai % A[i] perché
            // A[i]: semplifico
            // logica
            DP[i][r] = DP[i-1][k] or DP[i-1][r - A[i]]
    ritorna DP[n][k]
```

Analisi della complessità La complessità dell'algoritmo è $\mathcal{O}(nk)$, 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 *super*polinomiale $\mathcal{O}(n \cdot 2^n)$.

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

Definizione 19.1 (problema fortemente $\mathbb{NP}$-completo)**.** *Sia R_p 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. R è fortemente $\mathbb{NP}$-completo se R_p è $\mathbb{NP}$-completo.*

Il problema clique, ad esempio, è fortemente $\mathbb{NP}$-completo, mentre subSetSum non lo è.

Definizione 19.2 (problema debolmente $\mathbb{NP}$-completo)**.** *Se un problema $\mathbb{NP}$-completo non è fortemente $\mathbb{NP}$-completo, allora è debolmente $\mathbb{NP}$-completo.*

Nota. *Il problema di* subSetSum *è debolmente $\mathbb{NP}$-completo, in quanto limitando le dimensioni dei dati in ingresso il problema diventa polinomiale.*

19.1.2 Partizione

Definizione del problema Dato un insieme $A = \{a_1, a_2, \ldots, a_n\}$ di interi positivi, trovare un sottoinsieme S di $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$.

Considerazioni partizione è debolmente $\mathbb{NP}$-completo perché è possibile ridurlo a subSetSum scegliendo come valore k la metà di tutti i valori scelti.

19.1.3 Partizione di terne

Definizione del problema Dati $3n$ interi $\{a_1, a_2, \ldots, a_n\}$, trovare una partizione di n triple $T_1, \ldots, T_n$ tale che la somma dei tre elementi di ogni T_j sia la stessa, per $1 \leqslant j \leqslant n$.

Considerazioni partizioneTerne è fortemente $\mathbb{NP}$-completo, in quanto non esiste un algoritmo pseudo-polinomiale per risolvere questo problema.

19.2 Algoritmi di approssimazione

Nota.

19.2.1 Bin packing

Definizione del problema Utilizziamo un approccio ingordo.
Se consideriamo gli oggetti in ordine non descrescente.

19.2.2 Commesso viaggiatore

Definizione del problema

Restringiamo il problema

Teorema 1 (tsp non approssimabile)**.**

Nota. $\Delta - tsp$ *è un problema approssimabile, ma il problema generale no.*

19.3 Algoritmi branch&bound

Variante della tecnica di programmazione backtrack. Cerchiamo di analizare tutto lo spazio delle soluzioni, evitando certe sequence di scelte che facciano diminuire il costo della soluzione parziale costruita.
Cerchiamo i limiti (superiore ed inferiore) della soluzione minima.

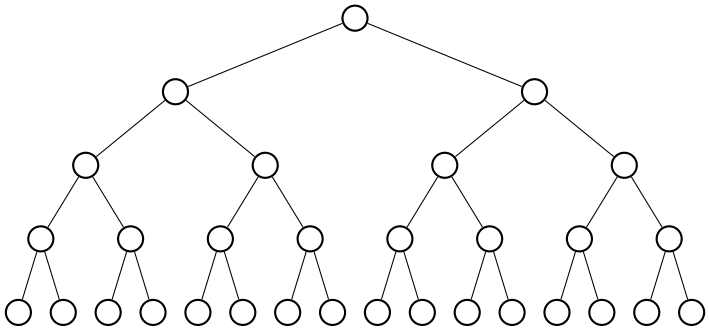


Figura 1: Se lb(S,i) è maggiore o uguale a *minCost*, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo.

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 deve essere il più possibile vicino alla soluzione ottima. Il limite superiore è dato dal *minCost*.

```
branch&bound(ITEM[] S, int n, int i, int minCost, int minSol)
    SET C = scelte(S, n, i, ...)  % determina l'insieme in funzione delle scelte precedenti
    // esamino ogni scelta
    per ciascun c ∈ C fai
        S[i] = c
        // calcolo il lower bound
        int lb = lb(S, i)  % calcolato in base alle scelte fatte fin'ora
        se lb < minCost allora
            // il limite inferiore non eccede il costo minimo
            se i < n allora
                // sono arrivato "in fondo"
                // faccio ricorsivamente le scelte successive
                branch&bound(S, n, i + 1, minCost, minSol)
            altrimenti se c(S,i) < minCost allora
                // la soluzione trovata è migliore del minimo parziale
                minSol = S  % aggiorno la soluzione minima parziale
                minCost = c(S,i)  % aggiorno il costo minimo parziale
```

Problema del commesso viaggiatore Applichiamo questo ragionamento al problema del commesso viaggiatore. 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 ammissibile 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 $S[1]$.
Facciamoci un po' di calcoli: $C[i]$ è il costo che ho sostenuto per fare i primi i passi.

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

Il lower bound della distanza per tornare a $S[1]$ ($O(n)$)

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

Lower bound della distanza per andarsene da $S[i]$ ($O(n)$)

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

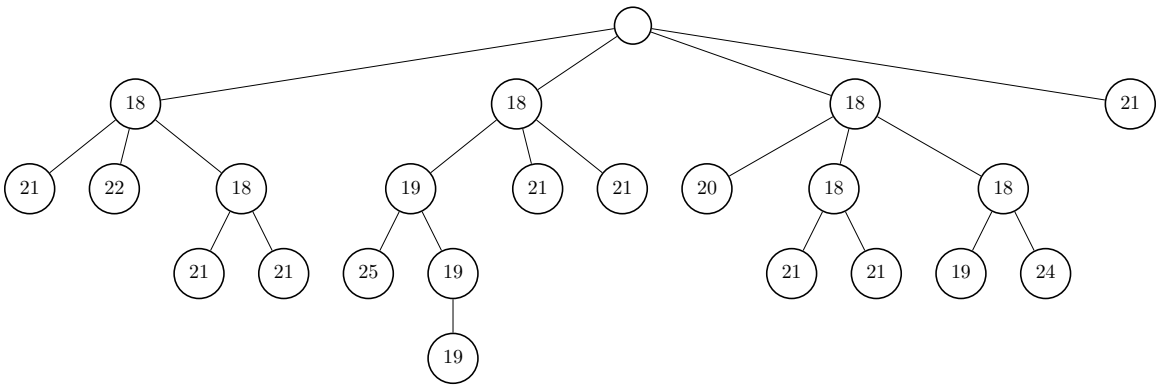
Lower bound della distanza percorsa per attraversare una qualsiasi di queste ultime $n - i$ città, provenendo da (e dirigendosi verso) un'altra di queste $n - i$ città ($O(n^3)$)

$$D[h] = \min_{p,q}\{d[p, h] + d[h, q] : h \neq p \neq q\},$$

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

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

```
bbTsp(Item[] S, int[] C, Set R, int n, int i)
    per ciascun c ∈ R fai
        S[i] = c
        R.remove(c)
        C[i] = C[i − 1] + d[S[i − 1]][S[i]]
        { calcola A, B, D[H] per ogni h ∈ R }
        int lb = C[i] + iif( i < n, ⌈(Σ_{h∉S} D[h])/2⌉, d[S[i]][S[1]] )
        // inizializza minCost ad una permutazione casuale
        minCost = random
        se lb < minCost allora
            se i < n allora
                bbTsp(S, C, R, n, i + 1)
            altrimenti
                C[n] = lb
                minSol = S
                minCost = C[n]
        R.remove(c)
```



19.4 Algoritmi euristici

Si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile, non necessariamente ottima né approssimata. Possiamo utilizzare le tecniche di programmazione per algoritmi ingordi o di ricerca locale.

Shortest edges first Cambiamo approccio per il problema del commesso viaggiatore. Ordiniamo gli archi per pesi non decrescenti e aggiungiamo archi alla soluzioni seguendo questo ordine finché non sono stati aggiunti $n-1$ archi, dove n è il numero di nodi. Però, attenzione, poter aggiungere un arco, occorre verificare che: arabic*) per ciascuno dei suoi nodi non siano stati già scelti due archi; arabic*) che non si formino circuiti (MFSET); A questo punto, si è trovata una catena Hamiltoniana e si chiude il circuito aggiungendo l'arco tra i due nodi estremi della catena.

```
greedyTsp(GRAPH G)
    SET S = Set
    MFSET M = Mfset(G.size)
    // archi in ingresso ad un nodo
    int[ ] in = new int[1...G.size]

    // inizializzazione
    da i = 1 fino a G.size fai
        in[i] = 0

    { ordina gli archi per peso decrescente }

    per ciascun [u,v] ∈ G.E fai
        se in[u] ≠ 2 and in[v] ≠ 2 and M.find(u) ≠ M.find(v) allora
            // non si è formato un ciclo

            S.insert(⟨u,v⟩)
            in[u]++
            in[v]++
            M.merge(u,v)

    // comment
    int u = 1
    finché in[1] ≠ 1 fai  u++

    int v = u + 1
    finché not in[v] ≠ 1 fai  v++

    // chiusura del circuito hamiltoniano
    S.insert(⟨u,v⟩)

    ritorna S
```

Analisi della complessità L'algoritmo costa $\mathcal{O}(n^2 \log n)$ a causa dell'*ordinamento degli archi*.

La soluzione ottenuta può essere la base di partenza per un algoritmo branch&bound che può essere migliorata tramite ricerca locale.

Nearest neighbor Si parte da una città e si seleziona come prossima città quella più vicina. Si va avanti così, evitando città già visitate. Quando si sono visitate tutte le città si torna alla città di partenza.

Analisi della complessità Questo algortimo ha complessità $\mathcal{O}(n^2)$ perché per ogni città devo guardare tutte le altre.