Algoritmi e Strutture Dati

Tecniche risolutive per problemi intrattabili

Alberto Montresor

Università di Trento

2020/04/30

Sommario

Chi si accontenta, gode

Proverbio

Le mieux est l'ennemi du bien
Il meglio è nemico del bene

Voltaire, La Bégueule, 1772

Introduzione

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

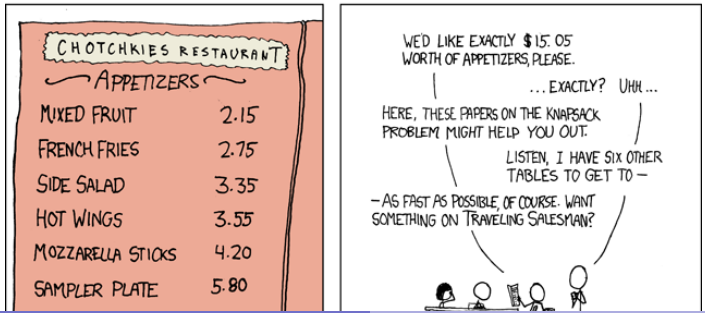
- Generalità:
  - Algoritmi pseudo-polinomiali che funzionano per solo alcuni casi particolari dell'input
- Ottimalità:
  - Algoritmi di approssimazione, che garantiscono di ottenere soluzioni "vicine" alla soluzione ottimale
- Formalità:
  - Algoritmi euristici, di solito basati su tecniche greedy o di ricerca locale, che forniscano sperimentalmente risultati buoni
- Efficienza:
  - Algoritmi esponenziali branch-&-bound, che limitano lo spazio di ricerca con un'accurata potatura

Esempio – Subset-Sum

Somma di sottoinsieme (SUBSET-SUM

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

https://xkcd.com/287/



Esempio – Subset-Sum

Somma di sottoinsieme (SUBSET-SUM

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

- 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

Subset Sum – 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.

$$DP[i][r] = \begin{cases} \\ \\ \\ \\ \end{cases}$$

$$\begin{aligned} & r = 0 \\ & r > 0 \wedge i = 0 \\ & r > 0 \wedge i > 0 \wedge A[i] > r \\ & r > 0 \wedge i > 0 \wedge A[i] \leq r \end{aligned}$$

Subset Sum – 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.

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

Subset Sum – Programmazione dinamica – $\Theta(nk)$

boolean subsetSum(**int**[] A, **int** n, **int** k)

boolean[][] DP = **new boolean**$[0 \dots n][0 \dots k]$ = {**false**}
for $i = 0$ **to** n **do**
  $DP[i][0]$ = **true** % $r = 0$
for $r = 1$ **to** k **do**
  $DP[0][r]$ = **false** % $r > 0 \wedge i = 0$
for $i = 1$ **to** n **do**
  for $r = 1$ **to** $A[i] - 1$ **do**
    $DP[i][r] = DP[i-1][r]$ % $A[i] > r$
  for $r = A[i]$ **to** k **do**
    $DP[i][r] = DP[i-1][r]$ **or** $DP[i-1][r - A[i]]$ % $A[i] \leq r$
return $DP[n][k]$

Subset Sum – Programmazione dinamica – $\Theta(nk)$

Esempio

$$A = [5, 9, 10]$$
$$n = 3$$
$$k = 24$$

i \ r	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	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	
1	1	∘	∘	∘	∘	1	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	**5**
2	1	∘	∘	∘	∘	1	∘	∘	∘	1	∘	∘	∘	∘	1	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	**9**
3	1	∘	∘	∘	∘	1	∘	∘	∘	1	1	∘	∘	∘	1	1	∘	∘	∘	1	∘	∘	∘	∘	1	**10**

Subset Sum – Backtracking – $O(2^n)$

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$(A, i-1, r-A[i])$

i \ r	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

Subset Sum – Memoization – $O(nk)$

```
boolean ssRec(int[] A, int i, int r, DICTIONARY DP)
if r == 0 then
    return true
else if r < 0 or i == 0 then
    return false
else
    boolean res = DP.lookup((i, r))
    if res == nil then
        res = ssRec(A, i − 1, r, DP)
        if A[i] < r then
            res = res or ssRec(A, i − 1, r − A[i], DP)
        DP.insert((i, r), res)
    return res
```

Subset Sum – Memoization – $O(nk)$

i \ r	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

Esempio

$A = [1, 1, 1, 1, 1]$

$n = 5$

$k = 5$

i \ r	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

Complessità – Riassunto

Programmazione dinamica	$\Theta(nk)$
Backtracking	$O(2^n)$
Memoization con dizionario	$O(nk), O(2^n)$

Dynamic Programming, Memoization $\equiv$ "Careful brute force"

Erik Demeine

Complessità – Discussione

$O(nk)$ è una complessità polinomiale?

Complessità – Discussione

$O(nk)$ è una complessità polinomiale?

- No, 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 $O(nk) = O(n \cdot 2^t)$, esponenziale

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

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

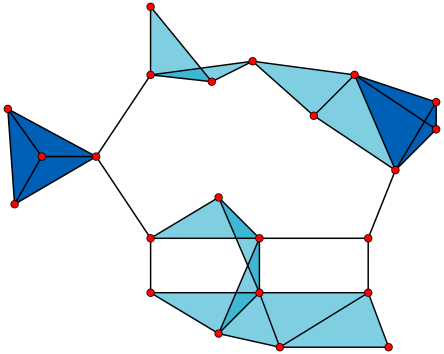
Esempi

Nome	I	$\#$	d
SUBSET-SUM	$\{n, k, A\}$	$\max\{n, k, \max(A)\}$	$O(n \log \#)$
CLIQUE			
TSP			

Esempio – Cricca

Cricca (CLIQUE)

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

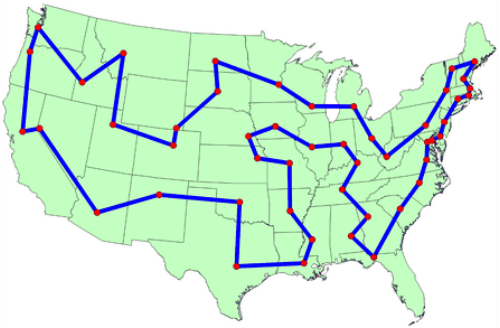


https://en.wikipedia.org/wiki/Clique_(graph_theory)

Esempio – Commesso viaggiatore

Commesso viaggiatore (Traveling salesperson - TSP)

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.



https://optimization.mccormick.northwestern.edu/images/a/aa/48StatesTSP.png

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

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

Esempi

Nome	I	$\#$	d
SUBSET-SUM	$\{n, k, A\}$	$\max\{n, k, \max(A)\}$	$O(n \log \#)$
CLIQUE	$\{n, m, k, G\}$	$\max\{n, m, k\}$	$O(n + m + \log \#)$
TSP	$\{n, k, d\}$	$\max\{n, k, \max(d)\}$	$O(n^2 \log \#)$

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

Definizione

Sia R_{pol} il problema R ristretto a quei dati d'ingresso per i quali $\#$ è limitato superiormente da $p(d)$, con p funzione polinomiale in d.
R è fortemente $\mathbb{NP}$-completo se R_{pol} è $\mathbb{NP}$-completo

Definizione

Se un problema $\mathbb{NP}$-completo non è fortemente $\mathbb{NP}$-completo,
allora è debolmente $\mathbb{NP}$-completo.

Esempio: Problema debolmente $\mathbb{NP}$-completo

Somma di sottoinsieme (SUBSET-SUM

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

Esempio – SUBSET-SUM è debolmente $\mathbb{NP}$-completo

- $\forall A[i] \leq k$ (valori più grandi di k vanno esclusi)
- Se $k = O(n^c)$, allora $\# = \max\{n, k, a_1, \dots, a_n\} = O(n^c)$
- La soluzione basata su programmazione dinamica ha complessità $O(nk) = O(n^{c+1})$, quindi in $\mathbb{P}$.
- Quindi SUBSET-SUM non è fortemente $\mathbb{NP}$-completo

Algoritmi pseudo-polinomiali

Definizione

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.

Esempio

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 $\mathbb{P} = \mathbb{NP}$.

Esempio: Problema fortemente $\mathbb{NP}$-completo

Cricca (CLIQUE)

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

CLIQUE è fortemente $\mathbb{NP}$-completo

- $k \leq n$ (altrimenti la risposta è **false**)
- $\# = \max\{n, m, k\} = \max\{n, m\}$
- $d = O(n + m + \log \#) = O(n + m)$
- Quindi $\# = \max\{n, m\}$ è limitato superiormente da $O(n + m)$
- Il problema ristretto è identico a CLIQUE, che è $\mathbb{NP}$-completo

Esempio – Problema fortemente $\mathbb{NP}$-completo

Commesso viaggiatore (Traveling salesperson - TSP)

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.

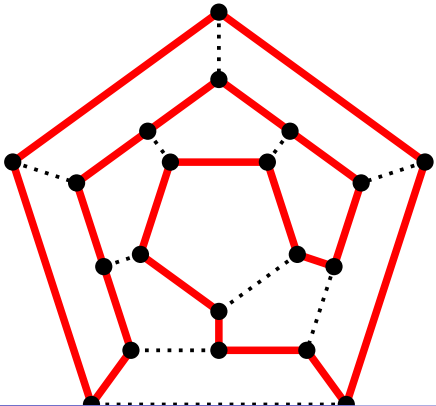
TSP è fortemente $\mathbb{NP}$-completo

- Per assurdo, supponiamo 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 - assurdo a meno che $\mathbb{P} = \mathbb{NP}$

Circuito hamiltoniano

Circuito hamiltoniano (HAMILTONIAN-CIRCUIT)

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



Complessità

- HAMILTONIAN-CIRCUIT è $\mathbb{NP}$-completo
- È uno dei 21 problemi elencati nell'articolo di Karp

Esempio – Problema fortemente $\mathbb{NP}$-completo

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
- Se esistesse un algoritmo pseudopolinomiale A per TSP, HAMILTONIAN CIRCUIT potrebbe essere risolto da A in tempo polinomiale

Problemi debolmente/fortemente $\mathbb{NP}$-completi?

Partizione (PARTITION)

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

Esempio

14	6	12	3	7	2

Domanda – PARTITION è debolmente $\mathbb{NP}$-completo?

Problemi debolmente/fortemente $\mathbb{NP}$-completi?

Partizione (PARTITION)

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

Esempio

14	6	12	3	7	2

Domanda – PARTITION è debolmente $\mathbb{NP}$-completo?

Sì, perchè è possibile ridurlo a SUBSET-SUM scegliendo come valore k la metà di tutti i valori presenti:

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

Problemi debolmente/fortemente $\mathbb{NP}$-completi

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 $1 \leq j \leq n$?

Domanda – 3-PARTITION è debolmente $\mathbb{NP}$-completo?

Problemi debolmente/fortemente $\mathbb{NP}$-completi

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 $1 \leq j \leq n$?

Domanda – 3-PARTITION è debolmente $\mathbb{NP}$-completo?

No, non esiste un algoritmo pseudo-polinomiale per risolvere 3-partition.

Algoritmi di approssimizzazione

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 algoritmi polinomiali che trovano soluzioni ammissibili più o meno vicine a quella ottima

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

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:

$$c(x^*) \leq c(x) \leq \alpha(n)c(x^*) \qquad \alpha(n) > 1 \qquad \text{(Minimizzazione)}$$
$$\alpha(n)c(x^*) \leq c(x) \leq c(x^*) \qquad \alpha(n) < 1 \qquad \text{(Massimizzazione)}$$

- $\alpha(n)$ può essere una costante, valida per tutti gli n
- Identificare un valore $\alpha(n)$ e dimostrare che l'algoritmo lo rispetta è ciò che rende un buon algoritmo un algoritmo di approssimazione

Esempio

Bin packing

Dati:

- un vettore A contenente n interi positivi (i volumi degli oggetti)
- un intero positivo k (la capacità di una scatola, tale che $\forall i : A[i] \leq k$),

si vuole trovare una partizione di $\{1, \ldots, n\}$ nel minimo numero di sottoinsiemi disgiunti ("scatole") tali che $\sum_{i \in S} A[i] \leq k$ per ogni insieme S della partizione

Esempio

3	7	2	5	4	3	5

Domanda

Come risolvereste il problema?

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3				

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3	7			

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3, 2	7			

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3, 2	7	5		

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3, 2	7	5	4	

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3, 2, 3	7	5	4	

First-fit

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. (Algoritmo Greedy)

Esempio

3	7	2	5	4	3	5

$k = 8$

3, 2, 3	7	5	4	5

Approssimazione First-Fit

- Sia $N > 1$ il numero di scatole usate da FIRST-FIT
  (se $N = 1$, FIRST-FIT è ottimale)
- Il numero minimo di scatole N^* è limitato da:
$$N^* \geq \frac{\sum_{i=1}^{n} A[i]}{k} = \frac{29}{8} = 3.625$$
- 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:
$$N < \frac{\sum_{i=1}^{n} A[i]}{k/2} = 2\frac{\sum_{i=1}^{n} A[i]}{k} \leq 2N^* = \alpha(n)N^*$$

  che implica $\alpha(n) = 2$

Approssimazione First-Fit

- E' possibile dimostrare un risultato migliore per FF:

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

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

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

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

Commesso viaggiatore con disuguaglianze particolari

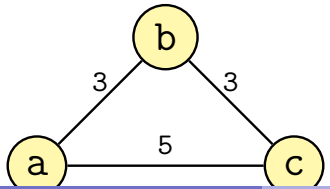
Commesso viaggiatore con dis. triangolari (Δ-TSP)

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

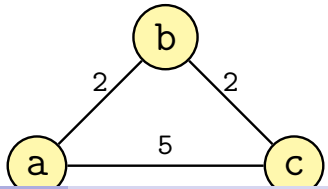
$$d[i][j] \leq d[i][k] + d[k][j] \quad \forall i, j, k : \quad 1 \leq i, j, k \leq n$$

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.

Con diseguaglianza triangolare



Senza diseguaglianza triangolare



Δ-TSP è $\mathbb{NP}$-completo

Dimostriamo che HAMILTONIAN-CIRCUIT $\leq_p$ Δ-TSP

- Sia $G = (V, E)$ un grafo non orientato
- Definiamo una matrice di distanze a partire da G

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

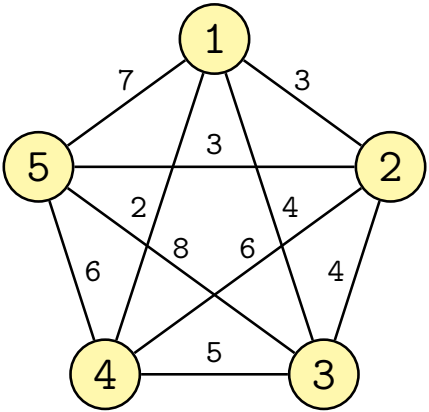
- Il grafo G ha un circuito hamiltoniano se e solo se è possibile trovare un percorso da commesso viaggiatore lungo n
- Valgono le diseguaglianze triangolari:

$$d[i][j] \leq 2 \leq d[i][k] + d[k][j]$$

Commesso viaggiatore vs circuito hamiltoniano pesato

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

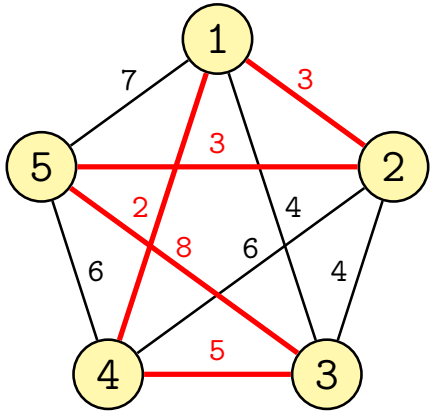
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Commesso viaggiatore vs 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	

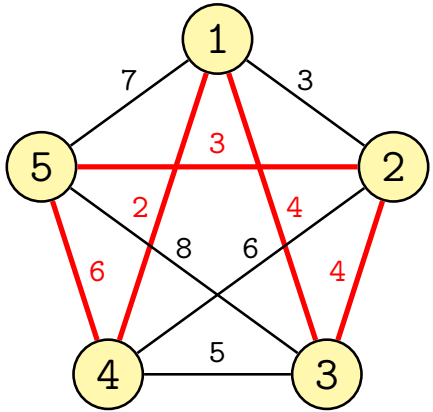


Costo: 21

Commesso viaggiatore vs circuito hamiltoniano pesato

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

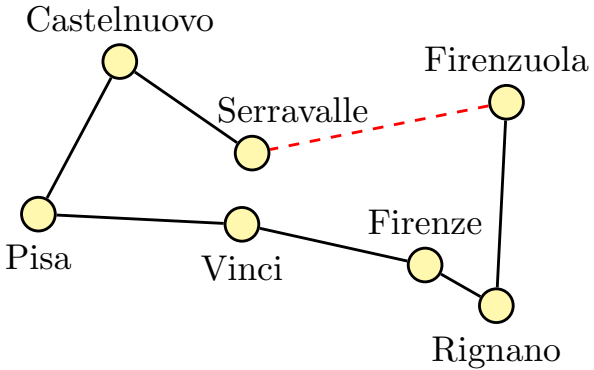
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 19

Algoritmo di approssimazione per Δ-TSP

- Interpretiamo Δ-TSP come il problema di trovare un circuito hamiltoniano di peso minimo su un grafo completo
- Si consideri un circuito hamiltoniano e si cancelli un suo arco
- Si ottiene un albero di copertura



Algoritmo di approssimazione per Δ-TSP

Teorema

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

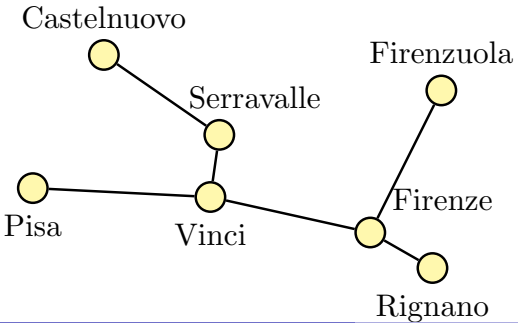
Dimostrazione

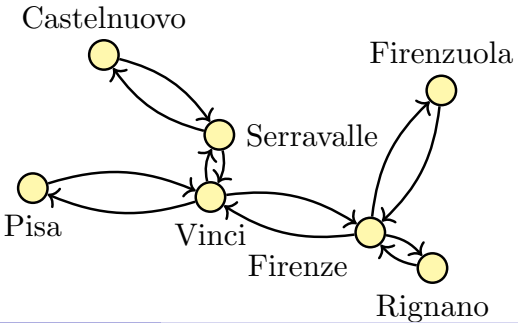
Per assurdo

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

Algoritmo per Δ-TSP

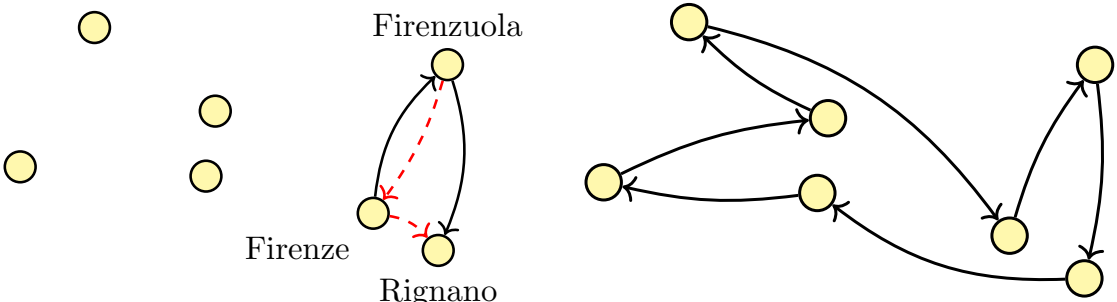
- Si individua un minimo albero di copertura di peso *mst* e se ne percorrono gli archi due volte, prima in un senso e poi nell'altro
- In questo modo, si visita ogni città almeno una volta
- La distanza complessiva di tale circuito è uguale a $2 \cdot mst$
- Ma non è un circuito hamiltoniano!





Algoritmo per Δ-TSP

- Si evita di passare per città già visitate - si saltano
- Per la disuguaglianza triangolare, il costo $c(\pi)$ del circuito così ottenuto è inferiore o uguale a $2 \cdot mst$
- Quindi: $c(\pi) \leq 2 \cdot mst < 2 \cdot c(\pi^*) \quad \Rightarrow \quad \alpha(n) = 2$
  dove $c(\pi^*)$ è il costo del circuito hamiltoniano ottimo



Algoritmo per Δ-TSP

Note

- La complessità dell'algoritmo è pari a $O(n^2 \log n)$:
  - $O(n^2 \log n)$ per algoritmo di Kruskal
  - $O(n)$ per visita in profondità del MST raddoppiato con $2n$ archi
- 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

Teorema

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

Dimostrazione

Per chi è interessato, nel libro.

Algoritmi euristici

Euristiche

Quando si è presi dalla disperazione a causa della enorme difficoltà di un problema di ottimizzazione NP-hard, si può ricorrere ad algoritmi "euristici" che forniscono una soluzione ammissibile

- non necessariamente ottima
- non necessariamente approssimata

Dal greco antico $\epsilon\upsilon\rho\iota\sigma\kappa\omega$ (eurisko), "Trovare, scoprire"

Tecniche possibili

- Greedy
- Ricerca locale

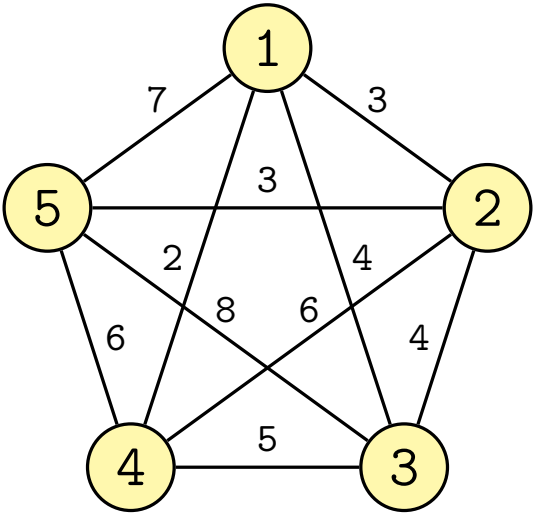
TSP – Greedy (1)

Shortest edges first

- Ordiniamo gli archi per pesi non decrescenti
- Aggiungiamo archi alla soluzioni 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 (MFSET)
- A questo punto, si è trovata una catena Hamiltoniana
- Si chiude il circuito aggiungendo l'arco tra i due nodi estremi della catena

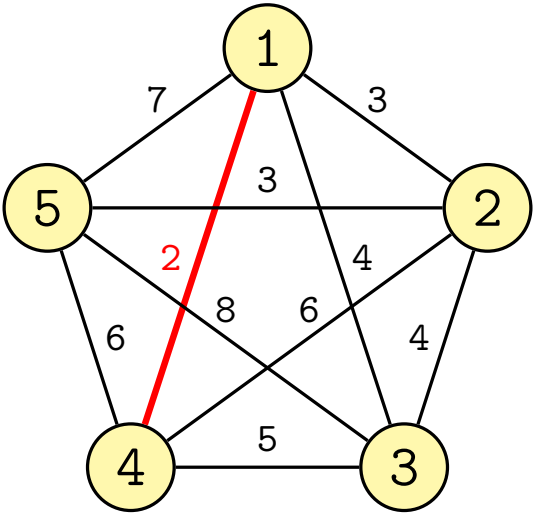
Esempio

	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	



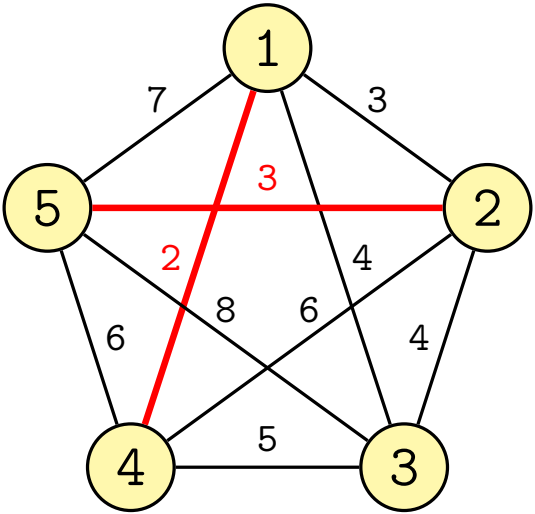
Esempio

	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	



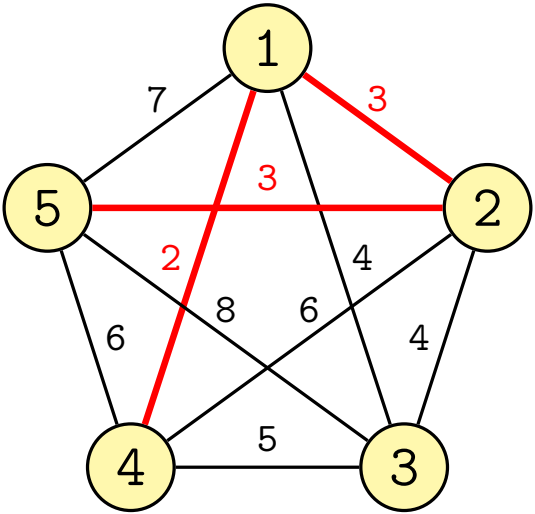
Esempio

	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	



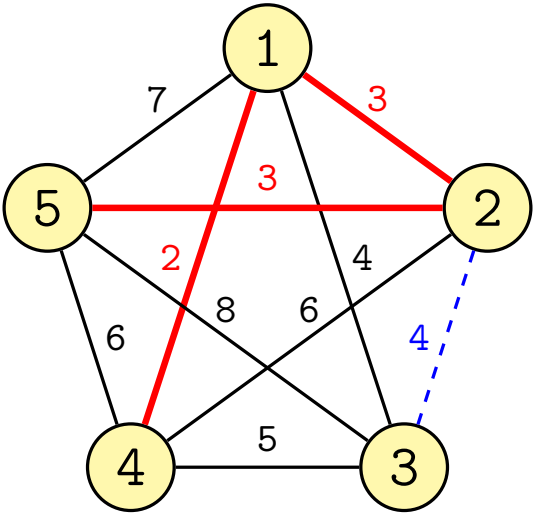
Esempio

	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	



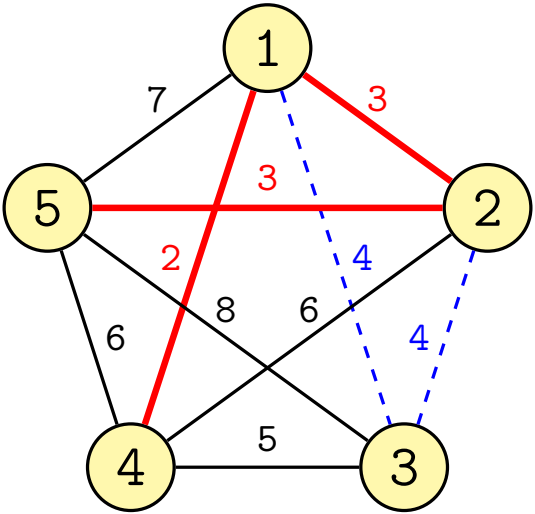
Esempio

	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	



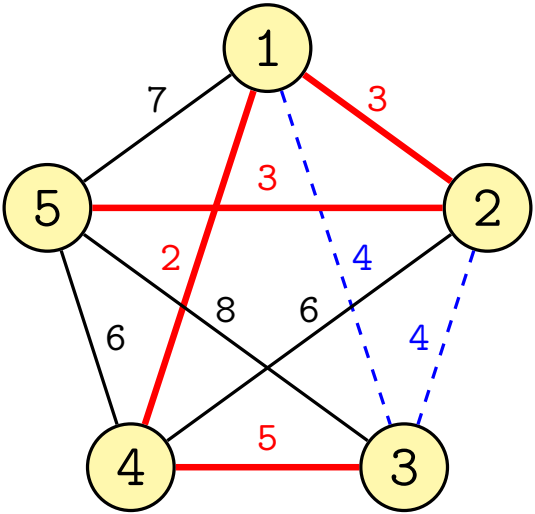
Esempio

	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	



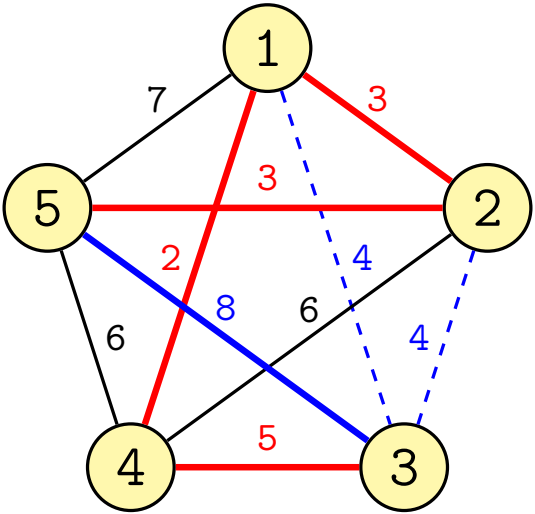
Esempio

	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	



Esempio

	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	



Costo: 21

TSP – Greedy (1)

```
SET greedyTsp(GRAPH G)
SET result = Set()
MFSET M = Mfset(G.n)
int[] edges = new int[1...n] = { 0 }                    % N. archi nella catena
A = {ordina gli archi per peso non decrescente}
foreach (u, v) ∈ A do
    if edges[u] < 2 and edges[v] < 2 and M.find(u) ≠ M.find(v) then
        result.insert( (u, v) )
        edges[u] = edges[u] + 1
        edges[v] = edges[v] + 1
        M.merge(u, v)
int u = 1; while edges[u] ≠ 1 do u = u + 1
int v = u + 1; while edges[v] ≠ 1 do v = v + 1
result.insert( (u, v) )
return result
```

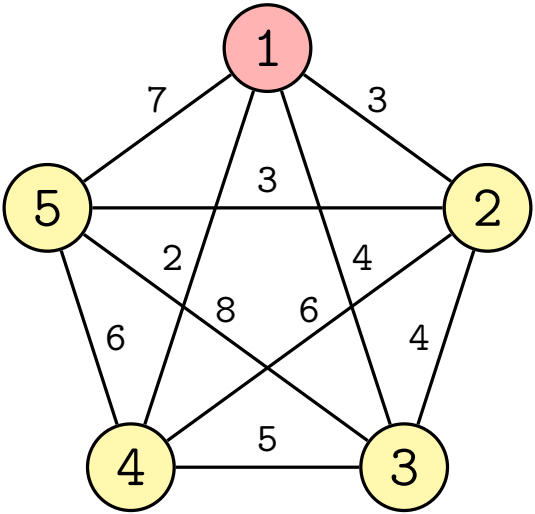
TSP – Greedy (2)

Nearest neighbor

- Si parte da una città
- 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
- Si può lavorare direttamente sulla matrice

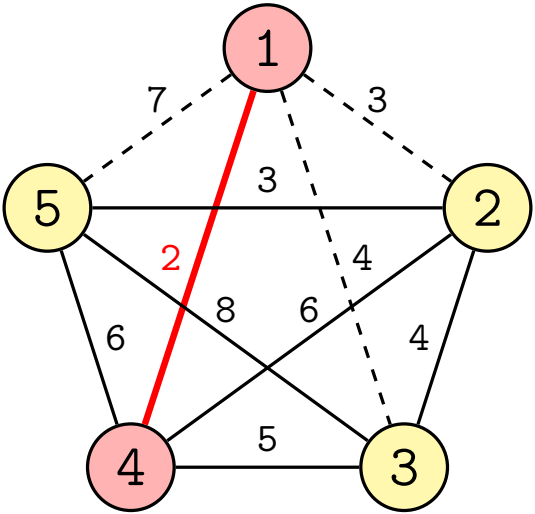
TSP – Greedy (2)

	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	



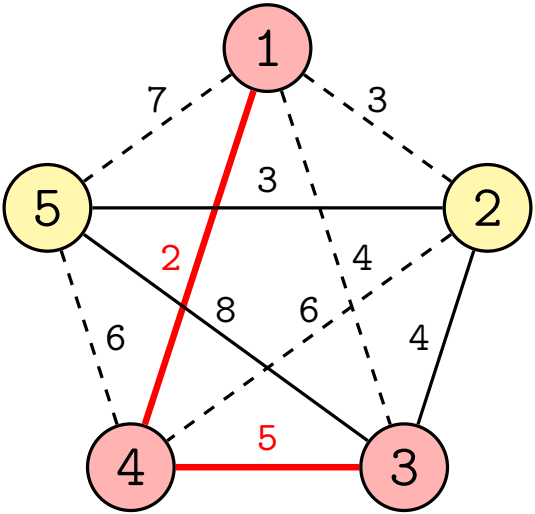
TSP – Greedy (2)

	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	



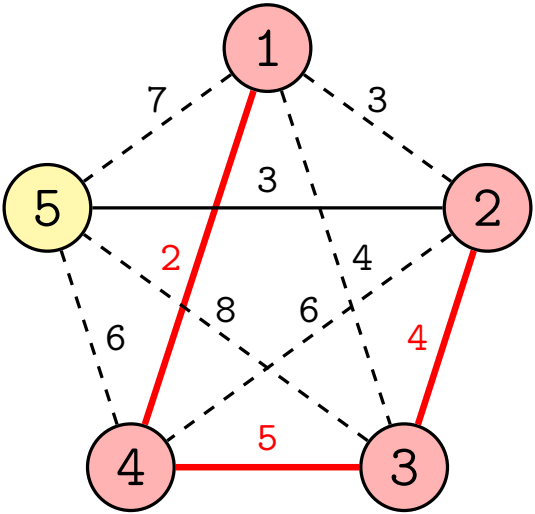
TSP – Greedy (2)

	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	



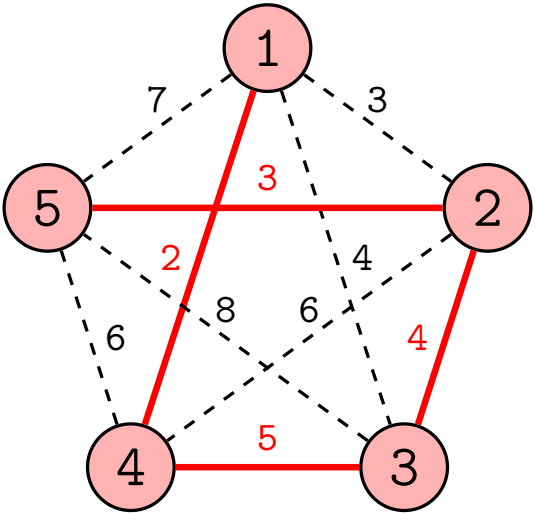
TSP – Greedy (2)

	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	



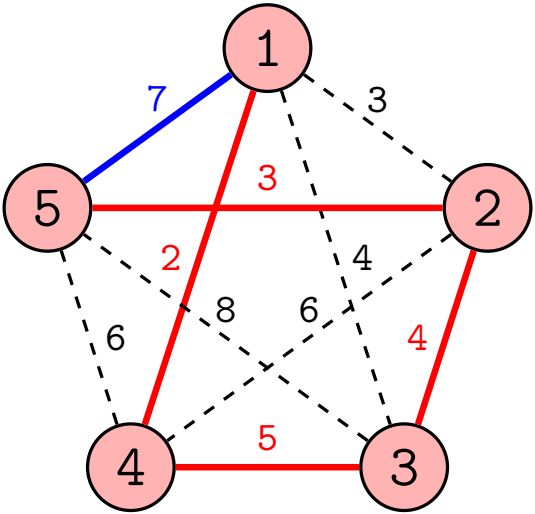
TSP – Greedy (2)

	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	



TSP – Greedy (2)

	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	



Costo: 21

TSP – Greedy

- Costo computazionale:
  - Greedy 1: $O(n^2 \log n)$ (ordinamento archi)
  - Greedy 2: $O(n^2)$
- La soluzione così ottenuta si può utilizzare come:
  - base di partenza per un algoritmo branch-&-bound
  - può essere migliorata ancora tramite ricerca locale

TSP – Approccio ricerca locale

Ricerca locale

Sia π un circuito Hamiltoniano del grafo completo derivante dal problema TSP. Si consideri il seguente intorno:

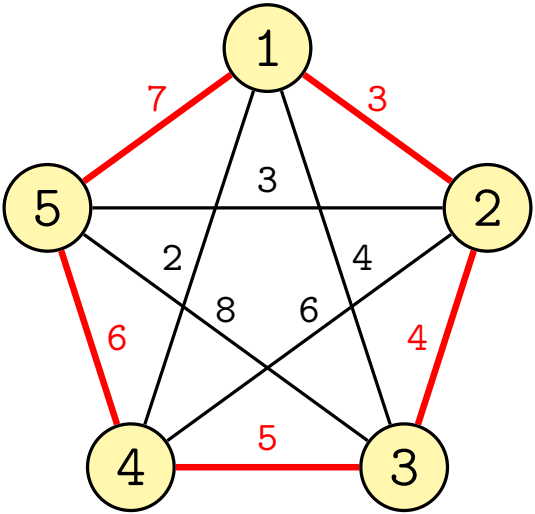
$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito}\}$$

Note

- $|I_2(\pi)| = n(n-1)/2 - n$
  - Ci sono $n(n-1)/2$ coppie di archi del circuito
  - n di esse sono consecutive
  - Una volta spezzato un circuito, esiste un solo modo per riconnetterlo
- Costo per esaminare $I_2(\pi)$: $O(n^2)$

Esempio

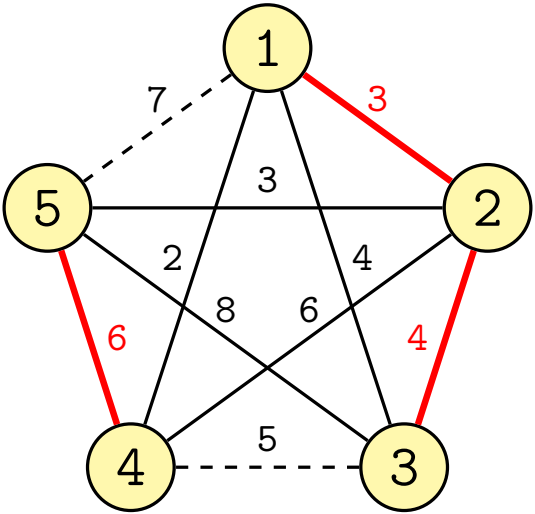
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 25

Esempio

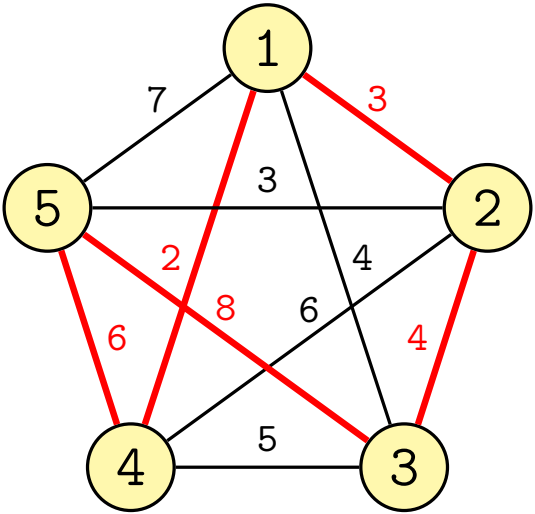
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 25

Esempio

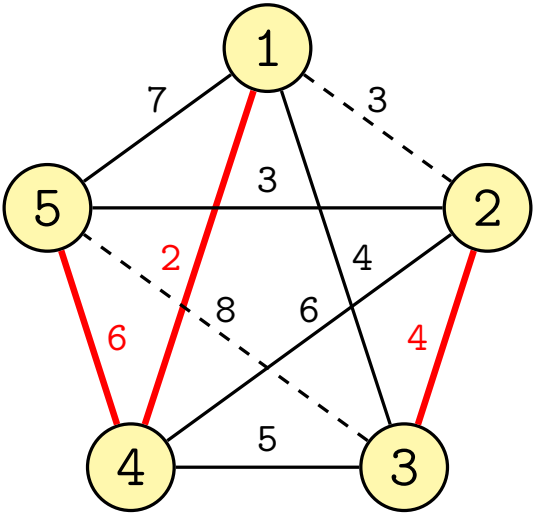
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 23

Esempio

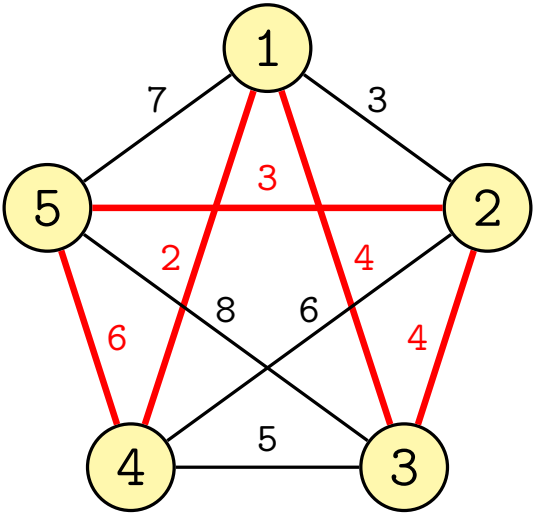
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 23

Esempio

	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	



Costo: 19

Algoritmi euristici

A heuristic is an algorithm that doesn't work.
(Except in practice. Sometimes. Maybe)

Jeff Erickson

Branch-&-bound

Branch-&-bound

Per risolvere un problema di ottimizzazione NP-arduo, si può modificare la procedura enumeration(), vista nella sezione su Backtrack, in modo da "potare" certe sequenze di scelte che si rivelino incapaci di generare la soluzione ottima

Assunzioni – senza perdere (troppa) generalità

- Problema di minimizzazione
- Ogni sequenza di scelte abbia costo non negativo
- Ogni scelta, aggiunta alle scelte già effettuate, non faccia diminuire il costo della soluzione parziale così costruita

Backtrack: ripasso

```
enumeration(⟨dati problema⟩, ITEM[] S, int i, ⟨dati parziali⟩)
if isAdmissible(⟨dati problema⟩, S, i, ⟨dati parziali⟩) then
    processSolution(⟨dati problema⟩, S, i, ⟨dati parziali⟩)
    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 = choices(⟨dati problema⟩, S, i, ⟨dati parziali⟩)
    foreach c ∈ C do
        S[i] = c
        if enumeration(⟨dati problema⟩, S, i + 1, ⟨dati parziali⟩) then
            return true                  % Trovata soluzione, restituisco true
    return false                         % Nessuna soluzione, restituisco false
```

Branch-&-bound

Upper bound

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

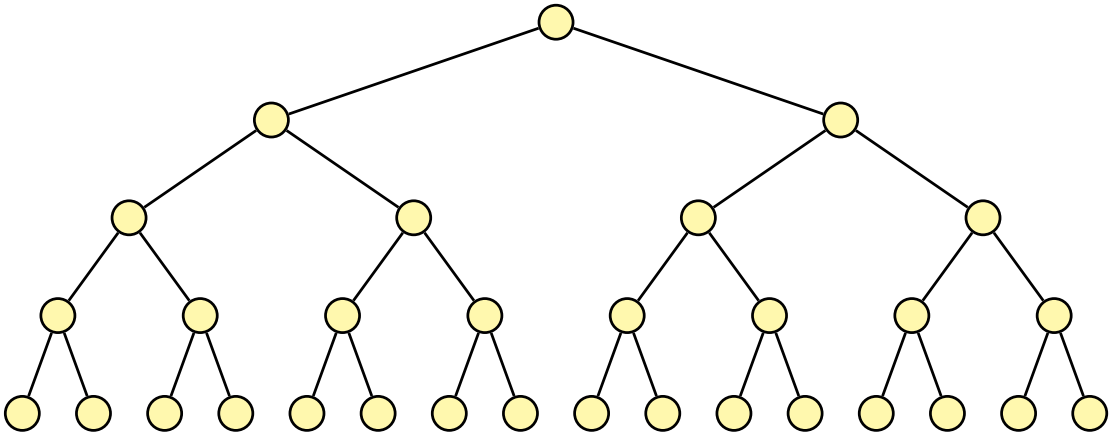
Lower bound

- Si supponga di avere disposizione una opportuna funzione lower bound $\mathsf{lb}(\langle \textit{dati problema} \rangle, S, i, \langle \textit{dati parziali} \rangle)$, che:
  - dipenda dalla sequenza di scelte fatte $S[1 \ldots i]$
  - garantisca che tutte le soluzioni ammissibili generabili facendo nuove scelte abbiano costo $\geq \mathsf{lb}()$

Branch-&-bound

Potatura

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



Branch-&-bound

Note

- Questo metodo non migliora la complessità (superpolinomiale) della procedura enumeration()
- Ne abbassa drasticamente il tempo di esecuzione in pratica
- Tutto dipende dalla funzione lb, che deve approssimare il più possibile il costo della soluzione ottima

Schema generale

branch&bound($\langle$*dati problema*$\rangle$, ITEM[] S, **int** i, $\langle$*dati parziali*$\rangle$)

```
SET C = choices(⟨dati problema⟩, S, i, ⟨dati parziali⟩)
foreach c ∈ C do
    S[i] = c
    int lb = lb(⟨dati problema⟩, S, i, ⟨dati parziali⟩)
    if lb < minCost then
        if i < n then
            branch&bound(⟨dati problema⟩, S, i + 1, ⟨dati parziali⟩)
        else
            if cost(S, i) < minCost then
                minSol = S                    % Variabile globale
                minCost = cost(S, i)          % Variabile globale
```

Commesso viaggiatore – Branch-&-bound

- Sia n il numero di città
- $d[h][k]$ la distanza intera positiva 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
  - attraversare le città $S[1 \dots i]$
  - passare da $S[i]$ ad una qualsiasi delle rimanenti $n - i$ città
  - attraversare queste ultime città in un ordine qualsiasi
  - da una di queste ritornare ad $S[1]$

Commesso viaggiatore – Branch-&-bound

- Distanza percorsa finora

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

- Lower bound della distanza per "uscire" da $S[i]$ $(O(n))$

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

- Lower bound della distanza per tornare a $S[1]$ $(O(n))$

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

- Lower bound della distanza percorsa per attraversare una qualsiasi città h delle $n - i$ città ancora da attraversare, provenendo da (e dirigendosi verso) una città non compresa in $S[2 \dots n]$ $(O(n^3))$

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

Lower bound

Se $i < n$, un possibile lower bound $\mathsf{lb}(d, S, i)$ calcola la somma:

- del costo $cost[i]$ per arrivare al nodo $S[i]$, già speso
- metà del costo ottenuto sommando:
  - il lower bound *out* del costo per andare dal nodo $S[i]$ ad un qualunque altro nodo
  - il lower bound per attraversare i nodi non contenuti in S
  - il lower bound *back* del costo per tornare al nodo $S[1]$ da un qualunque altro nodo

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

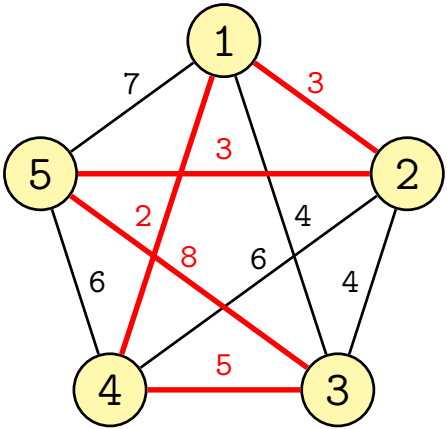
Commesso viaggiatore – Branch-&-bound

bbTsp(ITEM[] S, **int** *cost*, SET R, **int** n, **int** i)

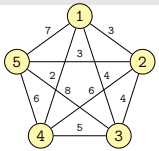
SET *choices* = copy(R)
foreach $c \in$ *choices* **do**
  $S[i] = c$
  R.remove(c)
  if $i < n$ **then**
    {calcola *out*, *back*, e *transfer*[h] per ogni $h \in R$}
    int $lb = cost[i] + \left\lceil \left(out + \sum_{h \notin S} transfer[h] + back \right) / 2 \right\rceil$
    if $lb < minCost$ **then**
      bbTsp(S, $cost + d[S[i-1]][S[i]]$, R, n, $i+1$)
  else
    $lb = lb + d[S[i]][S[1]]$
    if $lb < minCost$ **then**
      $minSol = S$
      $minCost = lb$
  R.insert(c)

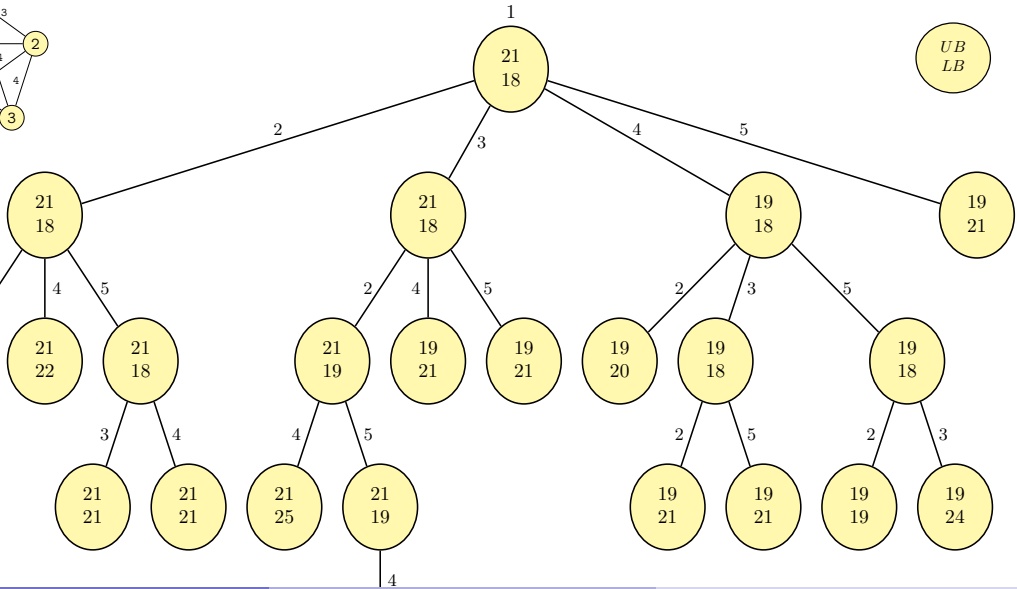
Dettagli

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



Esempio


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


1
21
18
UB
LB
2
3
4
5
21
18
21
18
19
18
19
21
3
4
5
2
4
5
2
3
5
21
21
21
22
21
18
21
19
19
21
19
21
19
20
19
18
19
18
3
4
4
5
2
5
2
3
21
21
21
21
21
25
21
19
19
21
19
21
19
19
19
24
4

Esempio

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

Possibili miglioramenti

- E' possibile variare l'ordine di visita dell'albero delle scelte
  - DFS vs Best-first
- E' possibile variare il meccanismo di branching
  - Sui nodi, sugli archi, etc.
- E' possibile cercare dei lower bound più stretti
  - Held, M., and Karp, R. M. (1971), "The Traveling Salesman Problem and Minimum Spanning Trees: part II", Mathematical Programming 1:6-25

Spunti di lettura

Bibliografia

- Jeff Erickson. Approximation Algorithms.
  `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/J-approx.pdf`
  (Per approfondire)
- David Williamson, David Shmoys (2010). The Design of Approximation Algorithms. Cambridge University Press.
  `http://www.designofapproxalgs.com/book.pdf`
  (Tanta roba)
- David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. The Traveling Salesman Problem: A Computational Study. Princeton University Press, 2007.
  (600 pagine *solo* su TSP)

Appunti di storia

SOLUTION OF A LARGE-SCALE TRAVELING-SALESMAN PROBLEM*

G. DANTZIG, R. FULKERSON, AND S. JOHNSON
The Rand Corporation, Santa Monica, California
(Received August 9, 1954)

It is shown that a certain tour of 49 cities, one in each of the 48 states and Washington, D. C., has the shortest road distance.

THE TRAVELING-SALESMAN PROBLEM might be described as follows: Find the shortest route (tour) for a salesman starting from a given city, visiting each of a specified group of cities, and then returning to the original point of departure. More generally, given an n by n symmetric matrix $D=(d_{IJ})$, where d_{IJ} represents the 'distance' from I to J, arrange the points in a cyclic order in such a way that the sum of the d

Appunti di storia

* Historical Note: The origin of this problem is somewhat obscure. It appears to have been discussed informally among mathematicians at mathematics meetings for many years. Surprisingly little in the way of results has appeared in the mathematical literature.[10] It may be that the minimal-distance tour problem was stimulated by the so-called Hamiltonian game[1] which is concerned with finding the number of different tours possible over a specified network. The latter problem is cited by some as the origin of group theory and has some connections with the famous Four-Color Conjecture.[9] Merrill Flood (Columbia University) should be credited with stimulating interest in the traveling-salesman problem in many quarters. As early as 1937, he tried to obtain near optimal solutions in reference to routing of school buses. Both Flood and A. W. Tucker (Princeton University) recall that they heard about the problem first in a seminar talk by Hassler Whitney at Princeton in 1934 (although Whitney, recently queried, does not seem to recall the problem). The relations between the traveling-salesman problem and the transportation problem of linear programming appear to have been first explored by M. Flood, J. Robinson, T. C. Koopmans, M. Beckmann, and later by I. Heller and H. Kuhn.[4,5,6]

Appunti di storia

In order to try the method on a large problem, the following set of 49 cities, one in each state and the District of Columbia, was selected:

1. Manchester, N. H.
2. Montpelier, Vt.
3. Detroit, Mich.
4. Cleveland, Ohio
5. Charleston, W. Va.
6. Louisville, Ky.
7. Indianapolis, Ind.
8. Chicago, Ill.
9. Milwaukee, Wis.
10. Minneapolis, Minn.
11. Pierre, S. D.
12. Bismarck, N. D.
13. Helena, Mont.
14. Seattle, Wash.
18. Carson City, Nev.
19. Los Angeles, Calif.
20. Phoenix, Ariz.
21. Santa Fe, N. M.
22. Denver, Colo.
23. Cheyenne, Wyo.
24. Omaha, Neb.
25. Des Moines, Iowa
26. Kansas City, Mo.
27. Topeka, Kans.
28. Oklahoma City, Okla.
29. Dallas, Tex.
30. Little Rock, Ark.
34. Birmingham, Ala.
35. Atlanta, Ga.
36. Jacksonville, Fla.
37. Columbia, S. C.
38. Raleigh, N. C.
39. Richmond, Va.
40. Washington, D. C.
41. Boston, Mass.
42. Portland, Me.
A. Baltimore, Md.
B. Wilmington, Del.
C. Philadelphia, Penn.
D. Newark, N. J.

Appunti di storia