

# Algoritmi e Strutture Dati

Tecniche risolutive per problemi intrattabili

Alberto Montresor

Università di Trento

2025/04/29

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



# Sommario

- 1 Introduzione
- 2 Algoritmi pseudo-polinomiali
- 3 Algoritmi di approssimazione
- 4 Algoritmi euristici
- 5 Algoritmi branch-&-bound

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 sono efficienti solo per 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 forniscono sperimentalmente risultati buoni
- **Efficienza:**
  - Algoritmi esponenziali **branch-&-bound**, che limitano lo spazio di ricerca con un'accurata potatura

# Sommario

- 1 Introduzione
- 2 Algoritmi pseudo-polinomiali**
- 3 Algoritmi di approssimazione
- 4 Algoritmi euristici
- 5 Algoritmi branch-&-bound

# 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  $\sum_{i \in S} a[i] = k$ ?

MY HOBBY:

EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

<https://xkcd.com/287/>

CHOTCHKIES RESTAURANT	
~ APPETIZERS ~	
MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80
~ SANDWICHES ~	
BARBECUE	6.55



## 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  $DP[0 \dots n][0 \dots k]$ .

$DP[i][r]$  è uguale a **true** se esiste un sottoinsieme dei primi  $i$  valori memorizzati in  $A$  la cui somma è pari a  $r$ , **false** altrimenti.

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

# Subset Sum – Programmazione Dinamica

Definiamo una tabella booleana  $DP[0 \dots n][0 \dots k]$ .

$DP[i][r]$  è uguale a **true** se esiste un sottoinsieme dei primi  $i$  valori memorizzati in  $A$  la cui somma è pari a  $r$ , **false** altrimenti.

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

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


---

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

---

```
boolean[][] DP = new boolean[0...n][0...k] = {false}
```

```
for i = 0 to n do
```

```
└ DP[i][0] = true                                % Obiettivo raggiunto
```

```
for r = 1 to k do
```

```
└ DP[0][r] = false                                % Valori terminati
```

```
for i = 1 to n do
```

```
└ for r = 1 to A[i] - 1 do
  └ DP[i][r] = DP[i - 1][r]                    % Valore troppo grande
```

```
└ for r = A[i] to k do
  └ DP[i][r] = DP[i - 1][r] or DP[i - 1][r - A[i]]    % Valore ok
```

```
return DP[n][k]
```

---

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

## Esempio

$$A = [5, 9, 10]$$

$$n = 3$$

$$k = 24$$

		r																								A		
		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		
i	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 % Obiettivo raggiunto
```

```
else if i == 0 then
```

```
    | return false % Valori terminati
```

```
else if A[i] > r then
```

```
    | return ssRec(A, i - 1, r) % Valore troppo grande
```

```
else
```

```
    | return ssRec(A, i - 1, r) or ssRec(A, i - 1, r - A[i]) % Valore ok
```

---

		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	
i	0	1	o	o	o	o	0	o	o	o	0	0	o	o	o	0	0	o	o	o	o	o	o	o	o	o	0	5
	1	o	o	o	o	o	1	o	o	o	o	o	o	o	o	0	0	o	o	o	o	o	o	o	o	o	0	9
	2	o	o	o	o	o	o	o	o	o	o	o	o	o	o	1	o	o	o	o	o	o	o	o	o	o	0	10
	3	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	1	

Subset Sum – Memoization –  $O(nk)$ ,  $O(2^n)$ 


---

```
boolean ssRec(int[] A, int i, int r, DICTIONARY DP)
```

---

```
if r == 0 then
```

```
    | return true
```

```
% Obiettivo raggiunto
```

```
else if i == 0 then
```

```
    | return false
```

```
% Valori terminati
```

```
else
```

```
    boolean res = DP.lookup((i, r))
```

```
    if res == nil then
```

```
        | res = ssRec(A, i - 1, r, DP)
```

```
% Valore non preso
```

```
        | if A[i] < r then
```

```
            | res = res or ssRec(A, i - 1, r - A[i], DP) % Valore preso
```

```
            | DP.insert((i, r), res)
```

```
    return res
```

---

Subset Sum – Memoization –  $O(nk)$ ,  $O(2^n)$ 

		r																								A			
		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			
i	0	1	◦	◦	◦	◦	0	◦	◦	◦	0	0	◦	◦	◦	◦	0	0	◦	◦	◦	◦	◦	◦	◦	◦	◦	0	5
	1	◦	◦	◦	◦	◦	1	◦	◦	◦	◦	◦	◦	◦	◦	◦	0	0	◦	◦	◦	◦	◦	◦	◦	◦	◦	0	9
	2	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	1	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	0	10
	3	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	◦	1	

## Esempio

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

$$n = 5$$

$$k = 5$$

		r						A
		0	1	2	3	4	5	
i	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 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  $I$

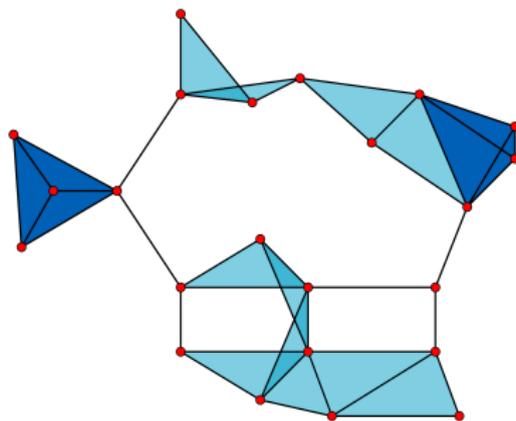
## Esempi

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?

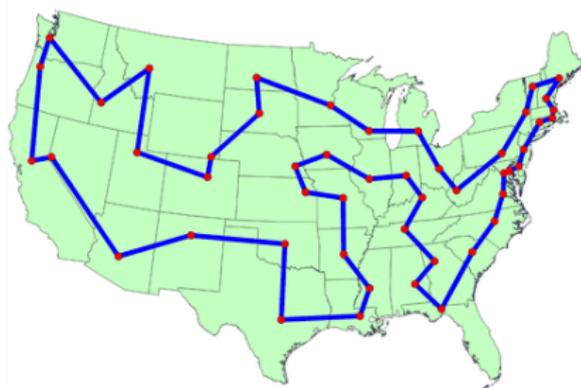


[https://en.wikipedia.org/wiki/Clique\\_\(graph\\_theory\)](https://en.wikipedia.org/wiki/Clique_(graph_theory))

## 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.



# Problemi fortemente, debolmente 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 NP-completi

### Definizione

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

### Definizione

Se un problema NP-completo non è fortemente NP-completo, allora è **debolmente NP-completo**.

## Esempio: Problema debolmente 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 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 NP-completo

# Algoritmi pseudo-polinomiali

## Definizione

Un algoritmo ha complessità **pseudo-polinomiale** se risolve un certo problema  $R$ , per qualsiasi dato  $I$  d'ingresso, in tempo  $T_p(\#, d)$ , funzione con tempo polinomiale (non costante) in  $\#$

## Esempio

Gli algoritmi per SUBSET-SUM e KNAPSACK sono pseudo-polinomiali

## Teorema

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

*Nota: Non vediamo la dimostrazione*

## Esempio: Problema fortemente 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 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 **già** limitato superiormente da  $O(n + m)$
- Il problema ristretto è identico a CLIQUE, che è NP-completo

## Esempio: Problema fortemente 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.

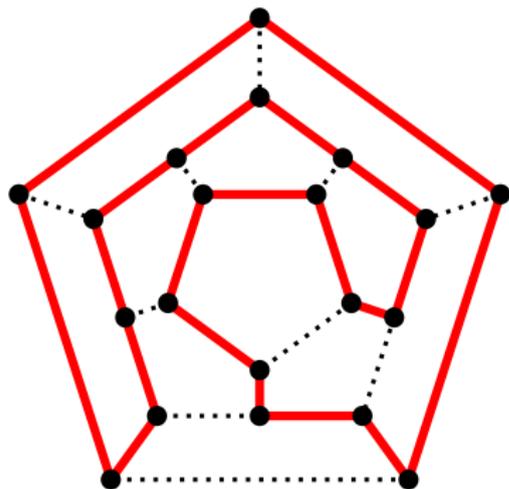
### TSP è fortemente NP-completo

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

# 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 NP-completo
- È uno dei 21 problemi elencati nell'articolo di Karp

[https://en.wikipedia.org/wiki/Hamiltonian\\_path#/media/File:Hamiltonian\\_path.svg](https://en.wikipedia.org/wiki/Hamiltonian_path#/media/File:Hamiltonian_path.svg)

## Esempio: Problema fortemente NP-completo

Dimostriamo che TSP è fortemente 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 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 NP-completo?**

Come lo risolvereste?

# Problemi debolmente/fortemente 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 NP-completo?

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 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 NP-completo?

Poichè è possibile risolverlo tramite un algoritmo pseudo-polinomiale, PARTITION è debolmente NP-completo

## Problemi debolmente/fortemente NP-completi?

### 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  $T_j$  è la stessa, per  $1 \leq j \leq n$ ?

**Domanda – 3-PARTITION è debolmente NP-completo?**

## Problemi debolmente/fortemente NP-completi?

### 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  $T_j$  è la stessa, per  $1 \leq j \leq n$ ?

### Domanda – 3-PARTITION è debolmente NP-completo?

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

# Sommario

- 1 Introduzione
- 2 Algoritmi pseudo-polinomiali
- 3 Algoritmi di approssimazione**
- 4 Algoritmi euristici
- 5 Algoritmi branch-&-bound

# Algoritmi di approssimazione

## Premessa

- I problemi più interessanti sono in forma di ottimizzazione
- Se il problema decisionale è  $\text{NP}$ -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  $n$ :

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

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

- $\alpha(n)$  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

# Bin-packing approssimato

## 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  $\{1, \dots, n\}$  nel minimo numero di sottoinsiemi disgiunti (“scatole”) tali che  $\sum_{i \in S} A[i] \leq k$  per ogni insieme  $S$  della partizione



# Bin-packing approssimato

**Esempio** –  $k = 8$

3	7	2	5	4	3	5
---	---	---	---	---	---	---

**Esempio** –  $k = 10$

3	2	7	5	3	5	10	5
---	---	---	---	---	---	----	---

**Esercizi**

- Provate a risolvere questi esempi "a mano"
- Ragionate su possibili algoritmi

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

--	--	--	--	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3				
---	--	--	--	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3	7			
---	---	--	--	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3, 1	7			
------	---	--	--	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3, 1	7	5		
------	---	---	--	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3, 1	7	5	4	
------	---	---	---	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3, 1, 3	7	5	4	
---------	---	---	---	--

# First-fit

## Algoritmo FIRST-FIT (Greedy)

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.

## Esempio

3	7	1	5	4	3	5
---	---	---	---	---	---	---

$$k = 8$$

3, 1, 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

- È possibile dimostrare un risultato migliore per First Fit:

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

- Variante 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

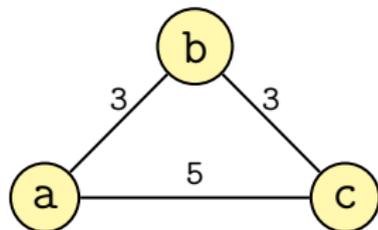
## Commesso viaggiatore con dis. triangolari ( $\Delta$ -TSP)

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

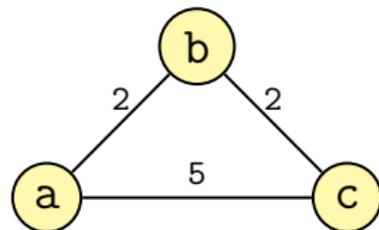
$$d[i][j] \leq d[i][k] + d[k][j] \quad \forall i, j, k : 1 \leq i, j, k \leq n$$

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



# $\Delta$ -TSP è NP-completo

## Lemma

HAMILTONIAN-CIRCUIT  $\leq_p$   $\Delta$ -TSP

## Dimostrazione

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

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

- Diseguaglianze triangolari:  $d[i][j] \leq 2 \leq d[i][k] + d[k][j]$
- $G$  ha un circuito hamiltoniano  $\Leftrightarrow$   
esiste un percorso  $\Delta$ -TSP in  $d$  di lunghezza  $n$

# $\Delta$ -TSP è NP-completo

## Lemma

HAMILTONIAN-CIRCUIT  $\leq_p$   $\Delta$ -TSP

## Teorema

$\Delta$ -TSP è NP-completo

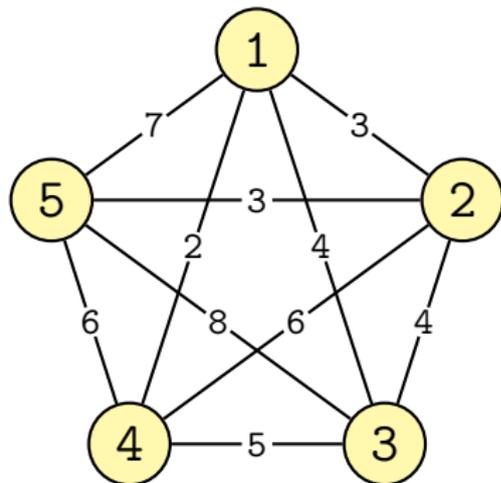
## Dimostrazione

Poiché HAMILTONIAN-CIRCUIT (un problema NP-completo) è riducibile a  $\Delta$ -TSP, ne segue che  $\Delta$ -TSP è NP-completo.

# Commesso viaggiatore vs circuito hamiltoniano pesato

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

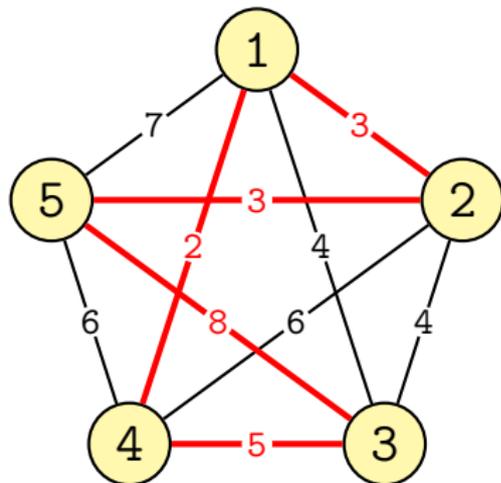
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



# Commesso viaggiatore vs circuito hamiltoniano pesato

- Interpretiamo ( $\Delta$ )-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	

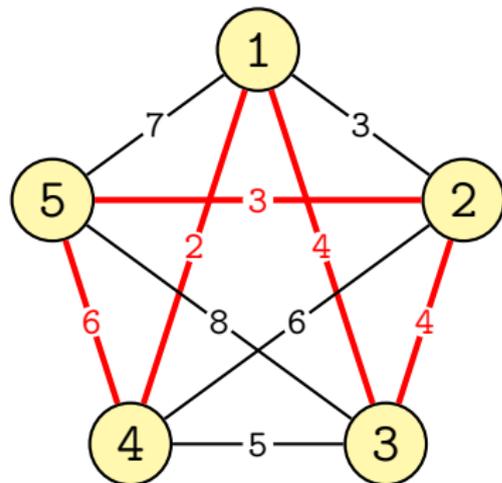


Costo: 21

# Commesso viaggiatore vs circuito hamiltoniano pesato

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

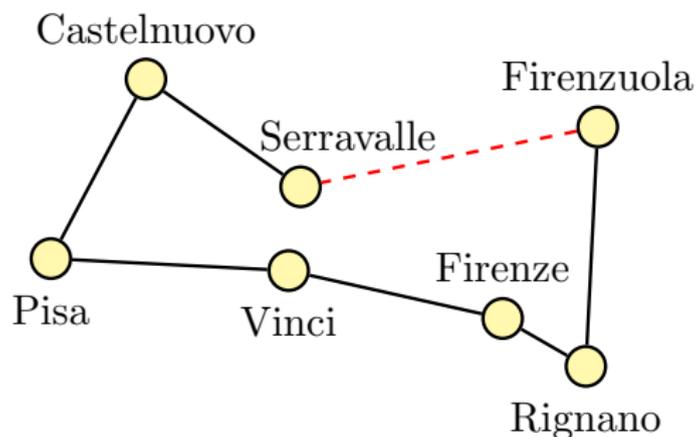
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 19

## Algoritmo di approssimazione per $\Delta$ -TSP

- Interpretiamo  $\Delta$ -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 $\Delta$ -TSP

## Teorema

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

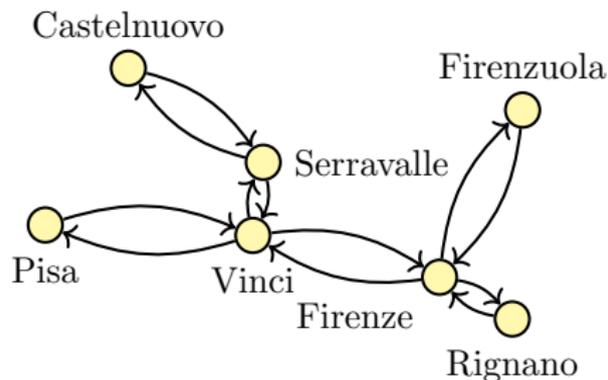
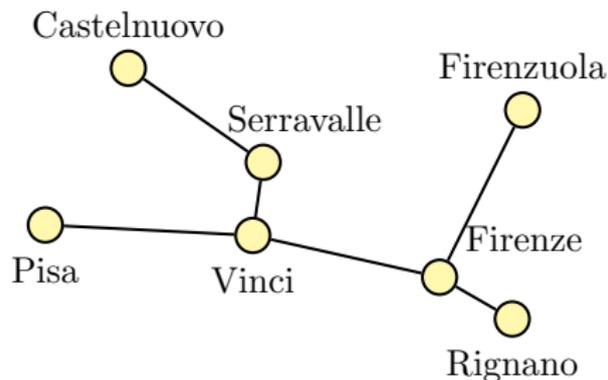
## Dimostrazione

Per assurdo

- Supponiamo che esista un circuito hamiltoniano  $\pi$  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

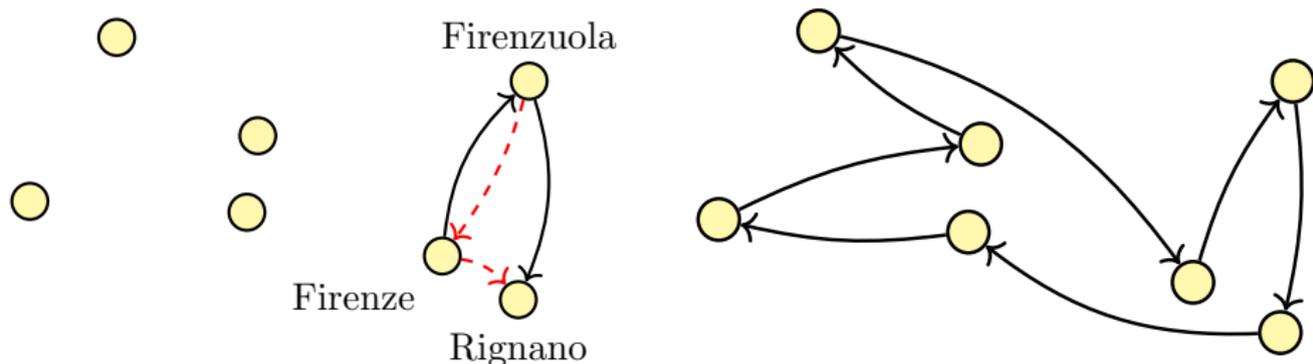
## Algoritmo per $\Delta$ -TSP

- 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 $\Delta$ -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:  $c(\pi) \leq 2 \cdot mst < 2 \cdot c(\pi^*) \Rightarrow \alpha(n) = 2$   
dove  $c(\pi^*)$  è il costo del circuito hamiltoniano ottimo



## Algoritmo per $\Delta$ -TSP

- 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...

## Algoritmo per $\Delta$ -TSP

- 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...
- Anna R. Karlin, Nathan Klein, Shayan Oveis Gharan. A (Slightly) Improved Approximation Algorithm for Metric TSP. STOC'21, best paper award. "For some  $\epsilon > 10^{-36}$ , we give a  $3/2 - \epsilon$  approximation algorithm for metric TSP."

# Non approssimabilità di TSP

## Teorema

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

## Dimostrazione

Per chi è interessato, nel libro.

# Sommario

- 1 Introduzione
- 2 Algoritmi pseudo-polinomiali
- 3 Algoritmi di approssimazione
- 4 Algoritmi euristici**
- 5 Algoritmi branch-&-bound

# 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 *ευρισκω* (eurisko), "Trovare, scoprire"

## Tecniche possibili

- Greedy
- Ricerca locale

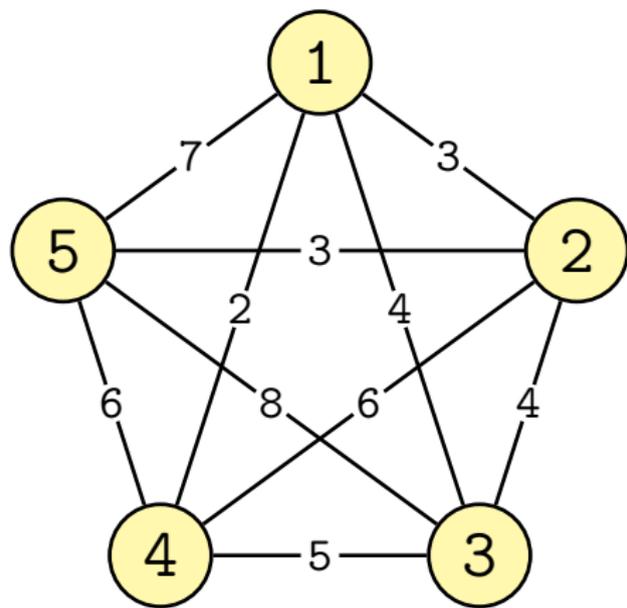
# TSP – Greedy (1)

## Shortest edges first

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

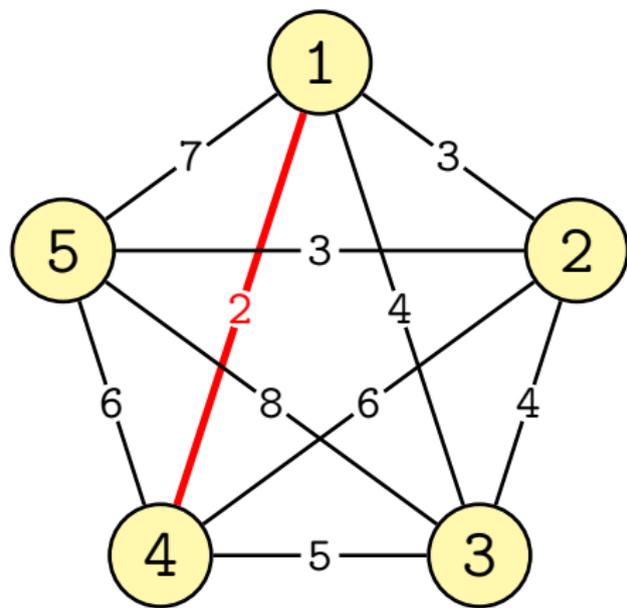
## Esempio

	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	



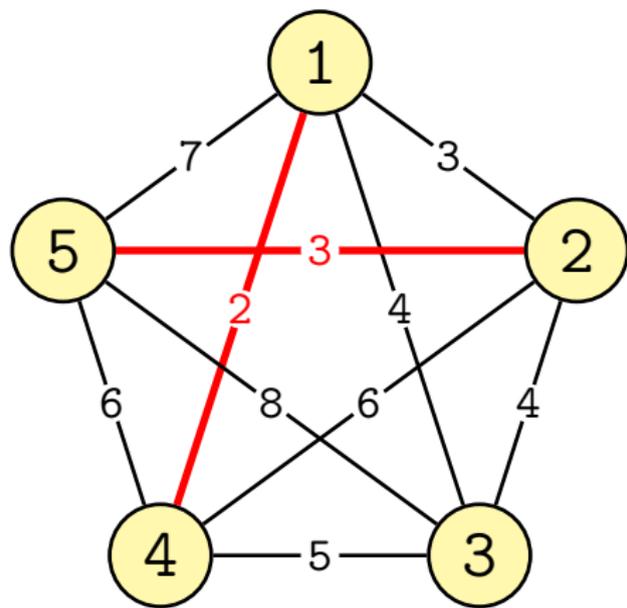
## Esempio

	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	



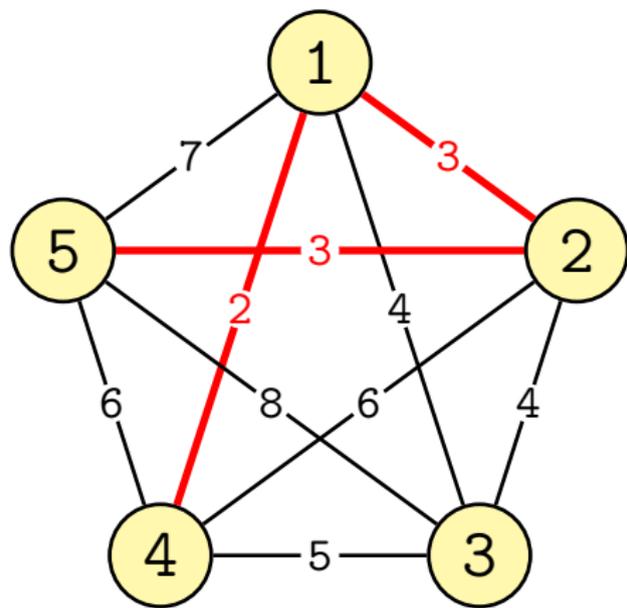
## Esempio

	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	



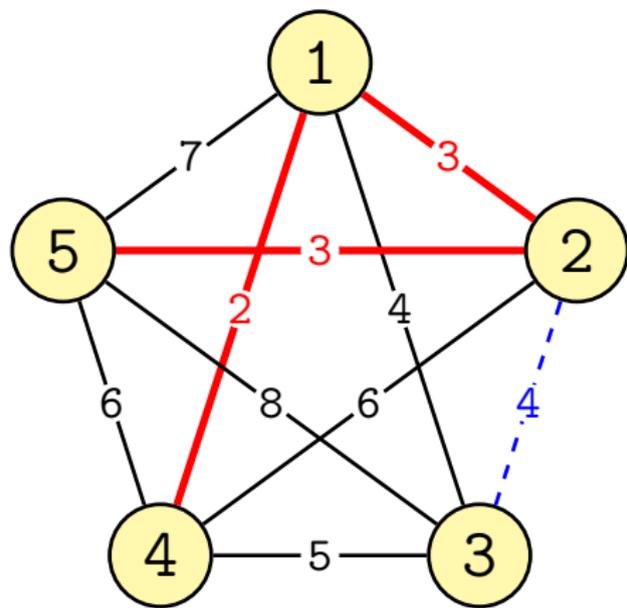
## Esempio

	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	



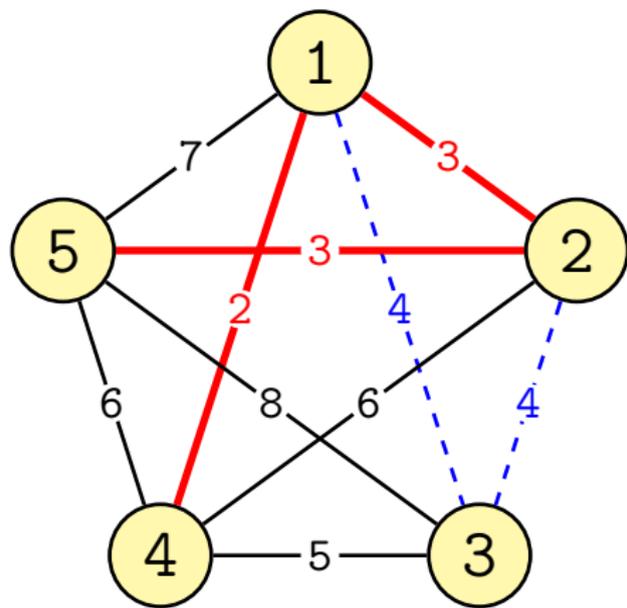
## Esempio

	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	



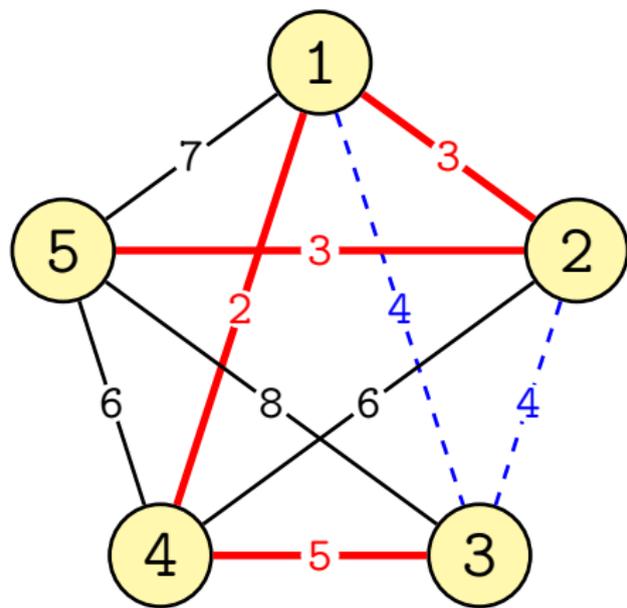
## Esempio

	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	



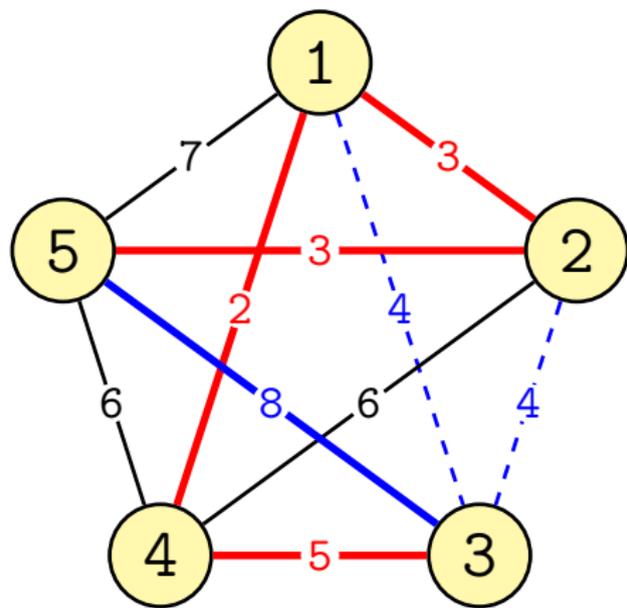
## Esempio

	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
```

---

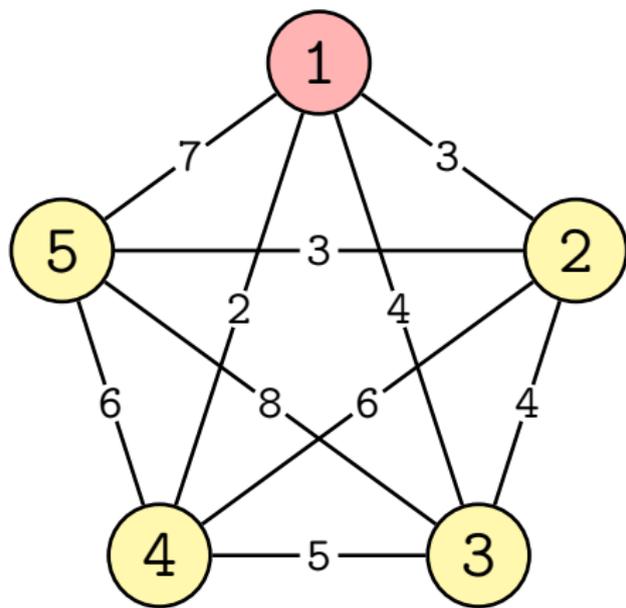
## TSP – Greedy (2)

### 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

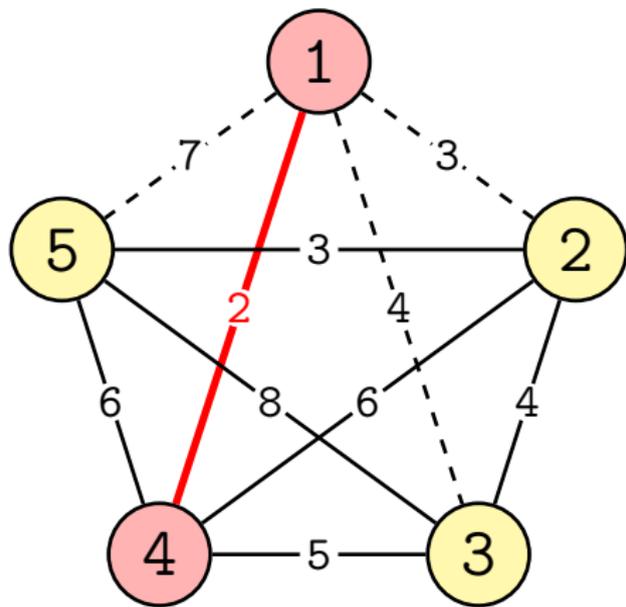
## TSP – Greedy (2)

	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	



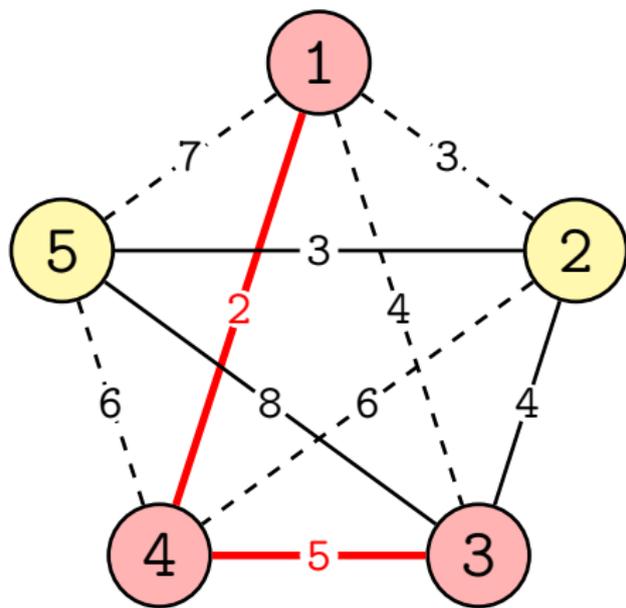
## TSP – Greedy (2)

	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	



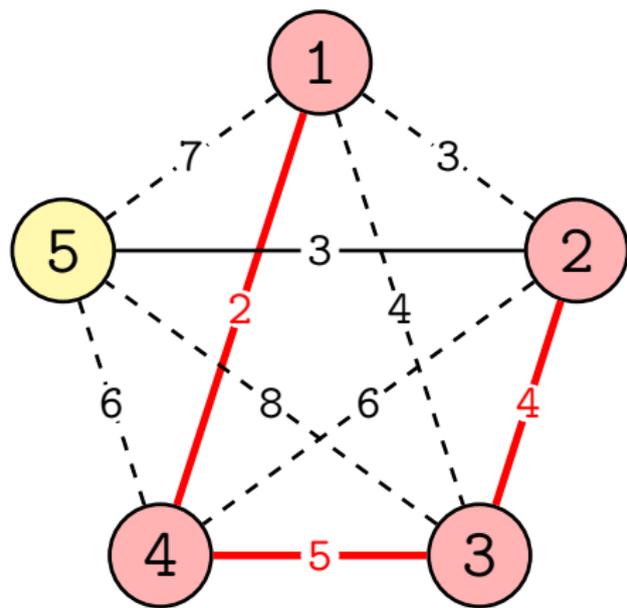
## TSP – Greedy (2)

	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	



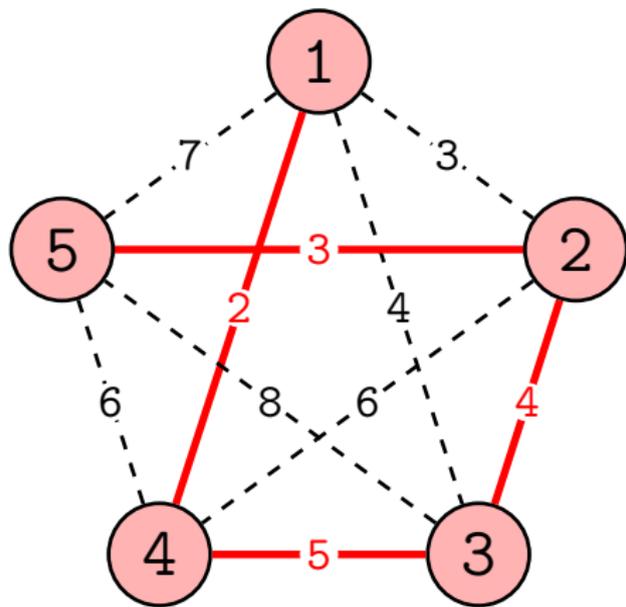
## TSP – Greedy (2)

	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	



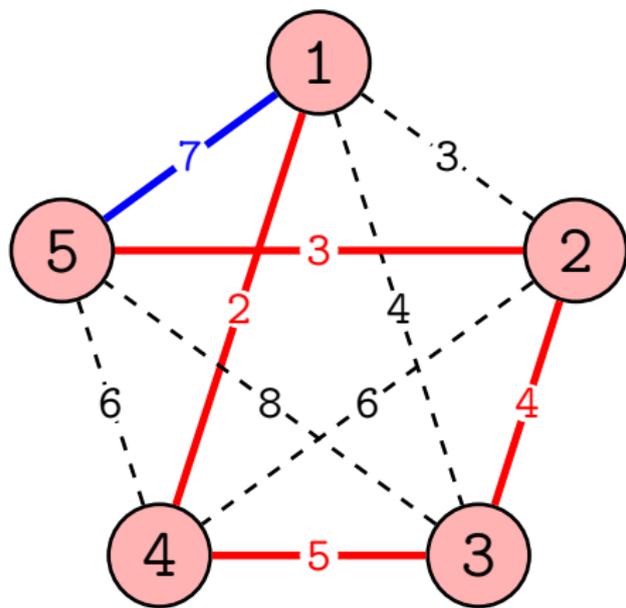
## TSP – Greedy (2)

	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  $\pi$  un circuito Hamiltoniano del grafo completo derivante dal problema TSP. Si consideri il seguente intorno:

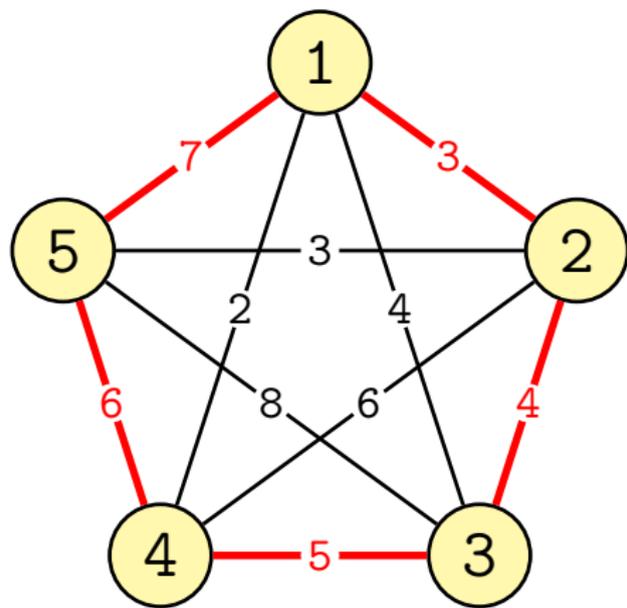
$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi non compresi nel circuito}\}$$

## 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

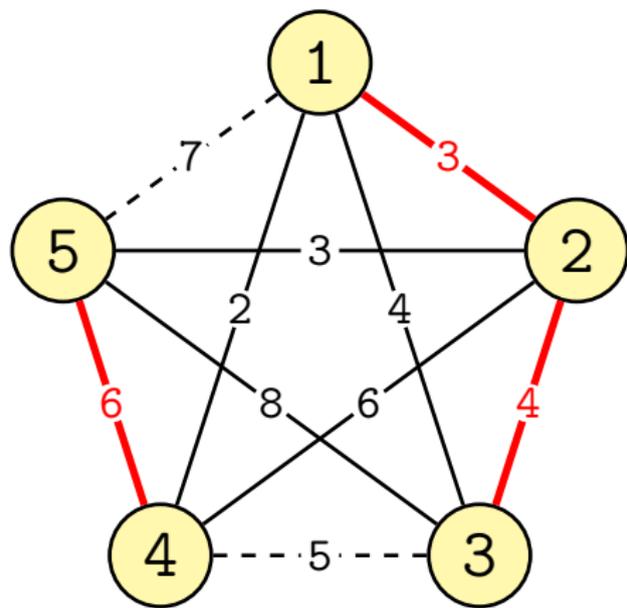
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 25

## Esempio

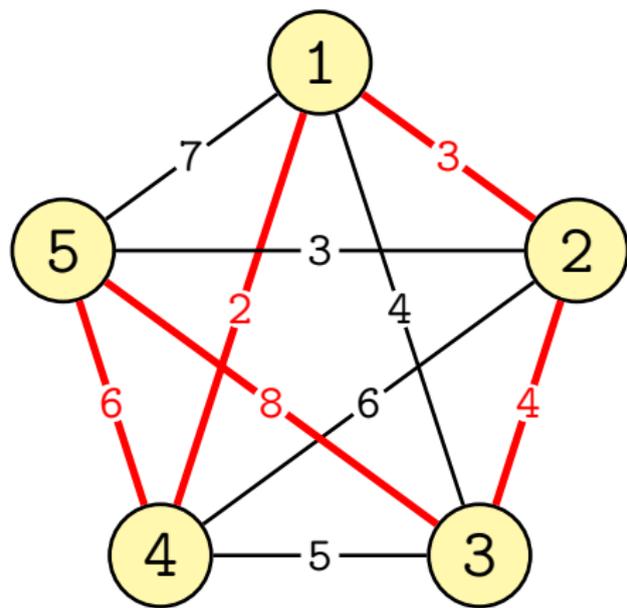
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 25

## Esempio

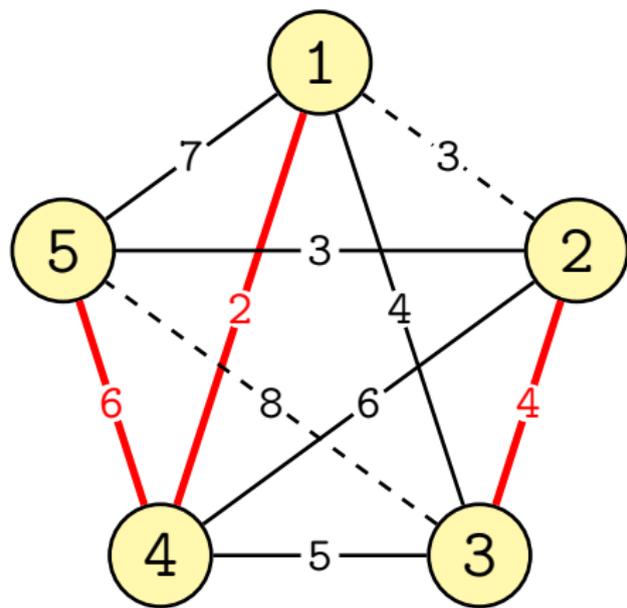
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 23

## Esempio

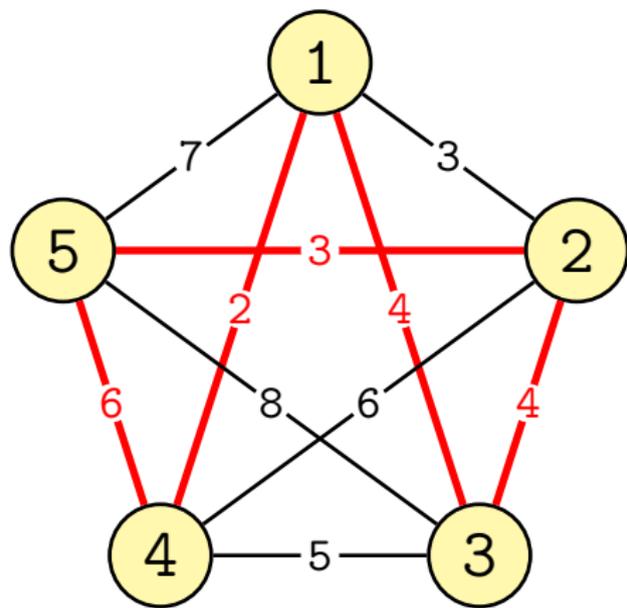
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



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

# Sommario

- 1 Introduzione
- 2 Algoritmi pseudo-polinomiali
- 3 Algoritmi di approssimazione
- 4 Algoritmi euristici
- 5 Algoritmi branch-&-bound**

# 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( $\langle \text{dati problema} \rangle$ , ITEM[]  $S$ , int  $i$ ,  $\langle \text{dati parziali} \rangle$ )
if isAdmissible( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ ) then
    processSolution( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ )
    return true % Trovata soluzione, restituisco true
else if reject( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i$ ,  $\langle \text{dati parziali} \rangle$ ) then
    return false % Impossibile trovare soluzioni, restituisco false
else
    SET  $C = \text{choices}(\langle \text{dati problema} \rangle, S, i, \langle \text{dati parziali} \rangle)$ 
    foreach  $c \in C$  do
         $S[i] = c$ 
        if enumeration( $\langle \text{dati problema} \rangle$ ,  $S$ ,  $i + 1$ ,  $\langle \text{dati parziali} \rangle$ ) 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**  $lb(\langle \text{dati problema} \rangle, S, i, \langle \text{dati parziali} \rangle)$ , che:
  - dipenda dalla sequenza di scelte fatte  $S[1 \dots i]$
  - garantisca che tutte le soluzioni ammissibili generabili facendo nuove scelte abbiano costo  $\geq lb()$



# 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 effettiva

# Schema generale

---

```
branch&bound(<dati problema>, ITEM[] S, int i, <dati parziali>)
```

---

```
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

## Situazione al passo $i$ -esimo

- 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  $\{1, \dots, n\}$

## Lower bound

Riuscite a calcolare un lower bound a partire da questi dati?

## Lower bound – Versione 1

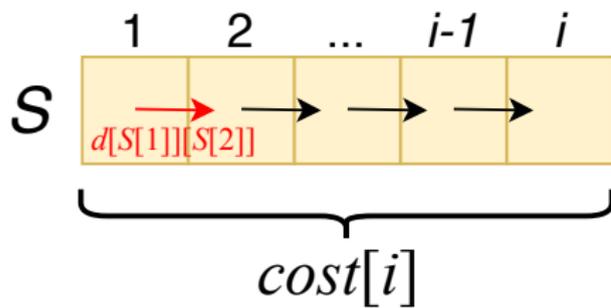
### Costo effettivo della soluzione trovata finora

La somma totale degli archi percorsi nel cammino percorso finora.

$$cost[i] = \sum_{h=2}^i d[S[h-1]][S[h]]$$

### Lower bound

$$lb(d, S, i) = cost[i]$$



## Lower bound – Versione 2

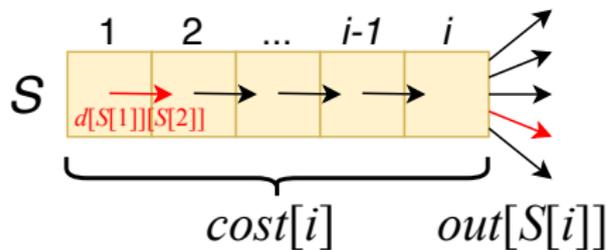
### Lower bound per "uscire" dall'ultimo nodo

Il valore minimo fra tutti gli archi che escono da  $S[i]$  e vanno in uno dei nodi non ancora scelti.

$$out[S[i]] = \min_{k \notin S[1..i]} d[S[i]][k]$$

### Lower bound

$$lb(d, S, i) = cost[i] + out[S[i]]$$



## Lower bound – Versione 3

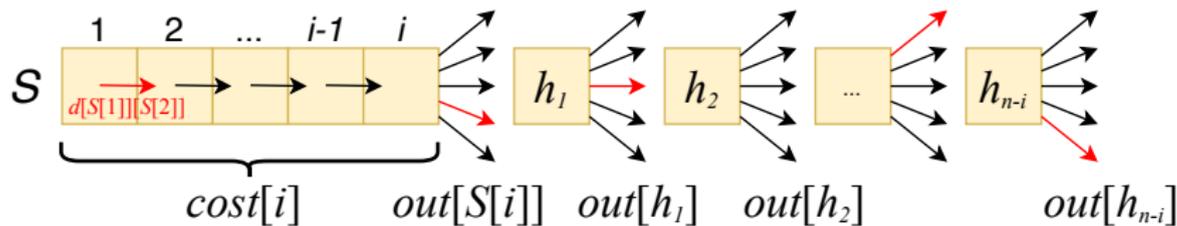
### Lower bound per "uscire" dai nodi non ancora scelti

Per ogni nodo  $h$  non scelto, il valore minimo fra gli archi che escono da esso e vanno in un nodo non ancora scelto oppure in  $S[1]$  (chiusura del circuito)

$$\forall h \notin S[1 \dots h] : out[h] = \min_{k \notin S[2 \dots i]} d[h][k]$$

### Lower bound

$$lb(d, S, i) = cost[i] + out[S[i]] + \sum_{h \notin S[1 \dots i]} out[h]$$

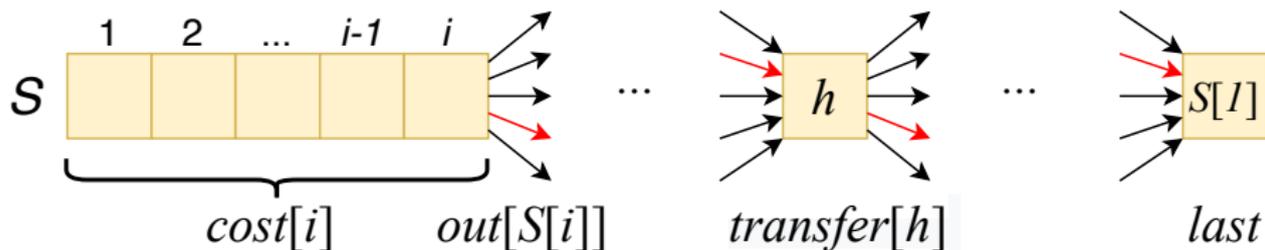


## Lower bound – Versione 4

## Lower bound per "attraversare" i nodi non ancora scelti

Per ogni nodo  $h$  non scelto, il costo minimo dato dalla somma di due archi  $(p, h)$  e  $(h, q)$  distinti che entrano e escono da  $h$ , proveniendo da  $S[i]$  o da un nodo non ancora scelto; andando in un nodo non ancora scelto oppure in  $S[1]$

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

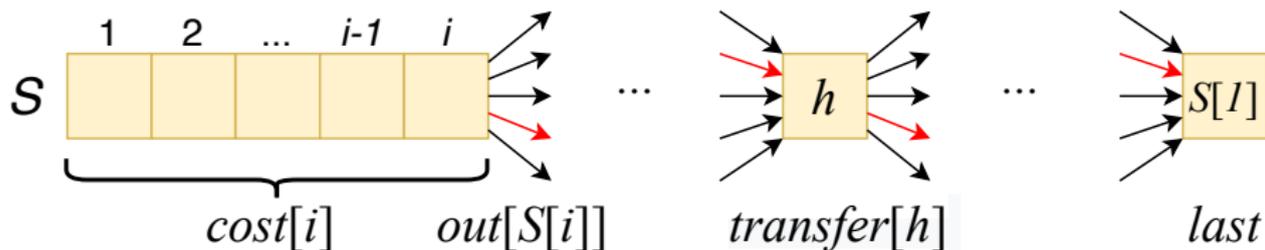


## Lower bound – Versione 4

## Lower bound per tornare nel primo nodo

Il valore minimo fra tutti gli archi che vanno da uno dei nodi non ancora scelti a  $S[1]$

$$last = \min_{h \notin S[1\dots i]} d[h][S[1]]$$

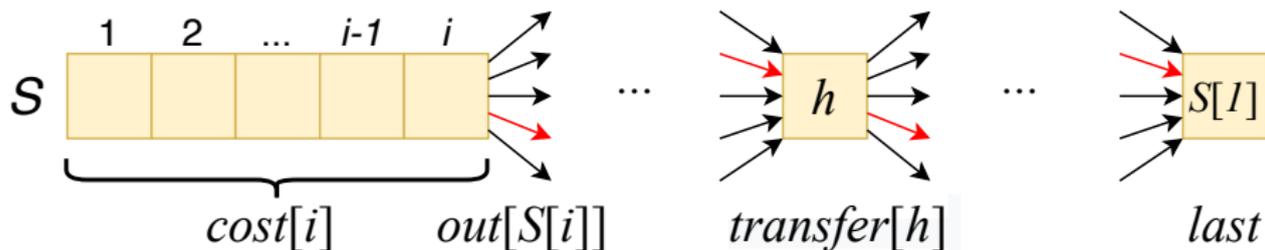


## Lower bound – Versione 4

## Lower bound

Si divide per un fattore 2 perché tutti gli archi vengono contati due volte, in uscita per un nodo e in entrata per un altro

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



## Comnesso viaggiatore – Branch-&amp;-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  $out$ ,  $last$ , e  $transfer[h]$  per ogni  $h \in R$ }    **int**  $lb = cost[i] + \lceil (out + \sum_{h \notin S} transfer[h] + last) / 2 \rceil$     **if**  $lb < minCost$  **then**      **bbTsp**( $S$ ,  $cost + d[S[i-1]][S[i]]$ ,  $R$ ,  $n$ ,  $i + 1$ )  **else**     $cost = cost + d[S[i]][S[1]]$     **if**  $cost < minCost$  **then**       $minSol = S$        $minCost = cost$ 

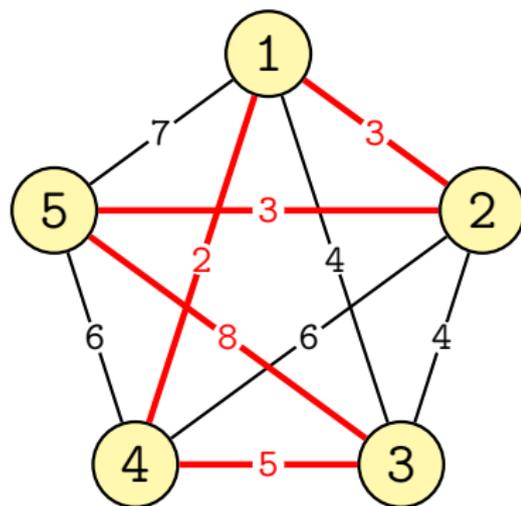
% Variabile globale

% Variabile globale

 $R.insert(c)$

# 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

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

### Possibili miglioramenti

- È possibile variare l'ordine di visita dell'albero delle scelte
  - DFS vs Best-first
- È possibile variare il meccanismo di branching
  - Sui nodi, sugli archi, etc.
- È 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>  
(500 pagine)
- 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)

## 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.

**T**HE 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  $d_{IJ}$  between consecutive points is minimal. Since there are only a finite

## Appunti di storia

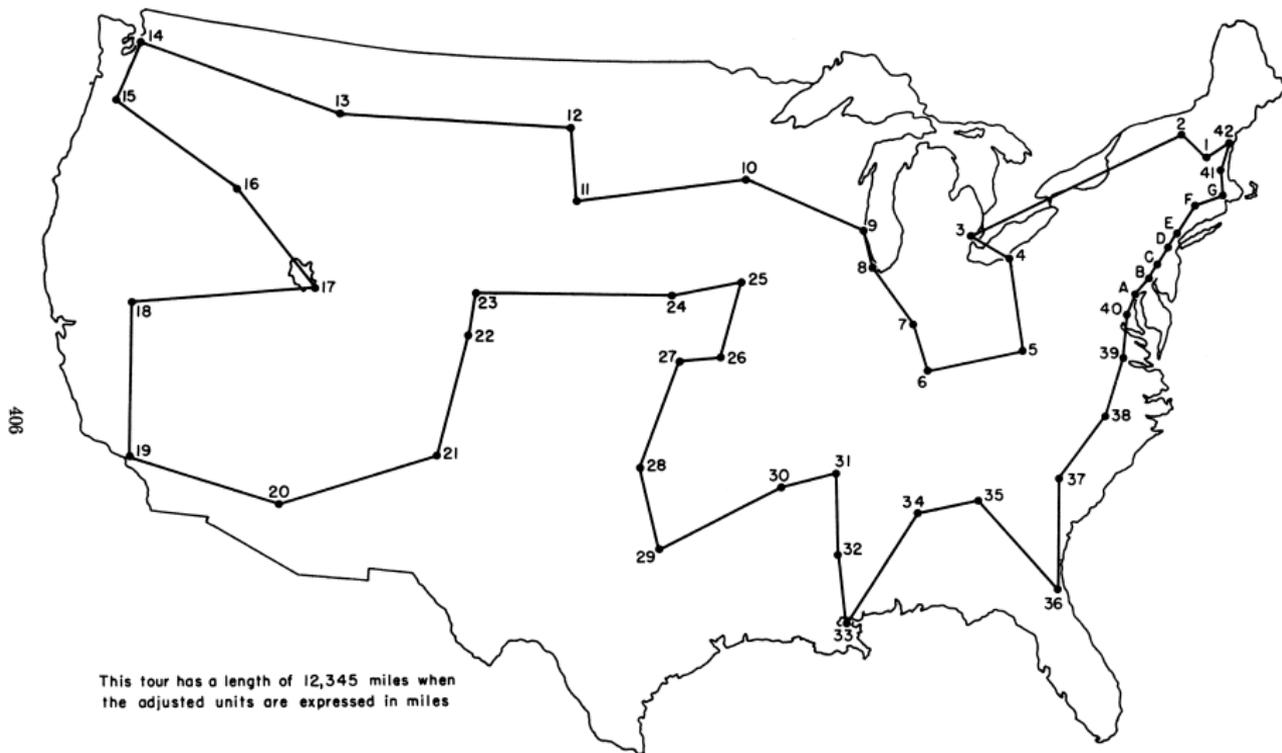
\* HISTORICAL NOTE: The origin of this problem is somewhat obscure. It appears to have been discussed informally among mathematicians at mathematics meetings for many years. Surprisingly little in the way of results has appeared in the mathematical literature.<sup>10</sup> It may be that the minimal-distance tour problem was stimulated by the so-called Hamiltonian game<sup>1</sup> 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.<sup>9</sup> 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.<sup>4,5,6</sup>

## 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.     | 18. Carson City, Nev.    | 34. Birmingham, Ala.   |
| 2. Montpelier, Vt.       | 19. Los Angeles, Calif.  | 35. Atlanta, Ga.       |
| 3. Detroit, Mich.        | 20. Phoenix, Ariz.       | 36. Jacksonville, Fla. |
| 4. Cleveland, Ohio       | 21. Santa Fe, N. M.      | 37. Columbia, S. C.    |
| 5. Charleston, W. Va.    | 22. Denver, Colo.        | 38. Raleigh, N. C.     |
| 6. Louisville, Ky.       | 23. Cheyenne, Wyo.       | 39. Richmond, Va.      |
| 7. Indianapolis, Ind.    | 24. Omaha, Neb.          | 40. Washington, D. C.  |
| 8. Chicago, Ill.         | 25. Des Moines, Iowa     | 41. Boston, Mass.      |
| 9. Milwaukee, Wis.       | 26. Kansas City, Mo.     | 42. Portland, Me.      |
| 10. Minneapolis, Minn.   | 27. Topeka, Kans.        | A. Baltimore, Md.      |
| 11. Pierre, S. D.        | 28. Oklahoma City, Okla. | B. Wilmington, Del.    |
| 12. Bismarck, N. D.      | 29. Dallas, Tex.         | C. Philadelphia, Penn. |
| 13. Helena, Mont.        | 30. Little Rock, Ark.    | D. Newark, N. J.       |
| 14. Seattle, Wash.       | 31. Memphis, Tenn.       | E. New York, N. Y.     |
| 15. Portland, Ore.       | 32. Jackson, Miss.       | F. Hartford, Conn.     |
| 16. Boise, Idaho         | 33. New Orleans, La.     | G. Providence, R. I.   |
| 17. Salt Lake City, Utah |                          |                        |

# Appunti di storia



This tour has a length of 12,345 miles when the adjusted units are expressed in miles

FIG. 16. The optimal tour of 49 cities.

# Solved TSP instances

1962

Procter and Gamble's Contest

33 cities

**HELP! WE'RE LOST!**

**HELP "CAR 54"... AND WIN CASH**  
54...\$1,000 PRIZES  
ONE...\$10,000 GRAND PRIZE

Map by Reed Wherry

Help Taddy and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.  
All you do is draw connecting straight lines from location to location to show the shortest round trip route.

**HERE'S THE CORRECT START...**  
Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Waco, West Virginia? Check the easy instructions on back of this entry blank for details.

© PROCTER & GAMBLE 1962

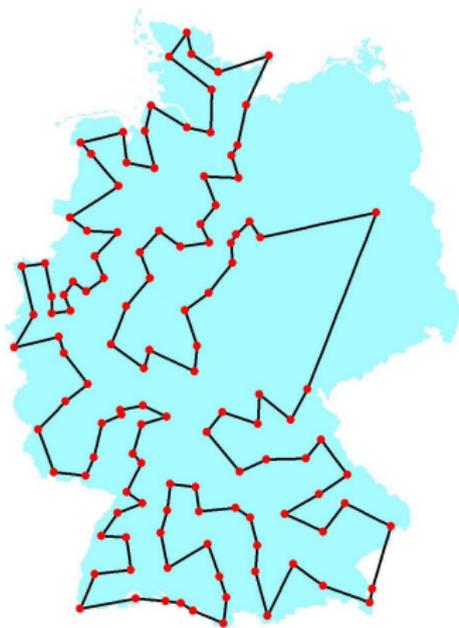
OFFICIAL RULES ON REVERSE SIDE

# Solved TSP instances

1977

Groetschel

120 cities of West Germany

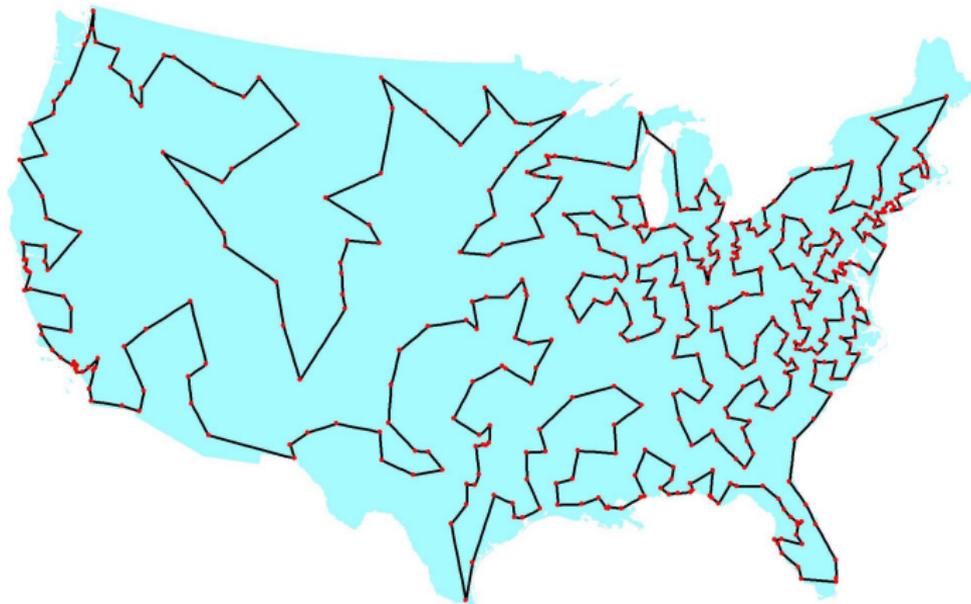


# Solved TSP instances

1987

Padberg and Rinaldi

532 AT&T switch locations in the USA

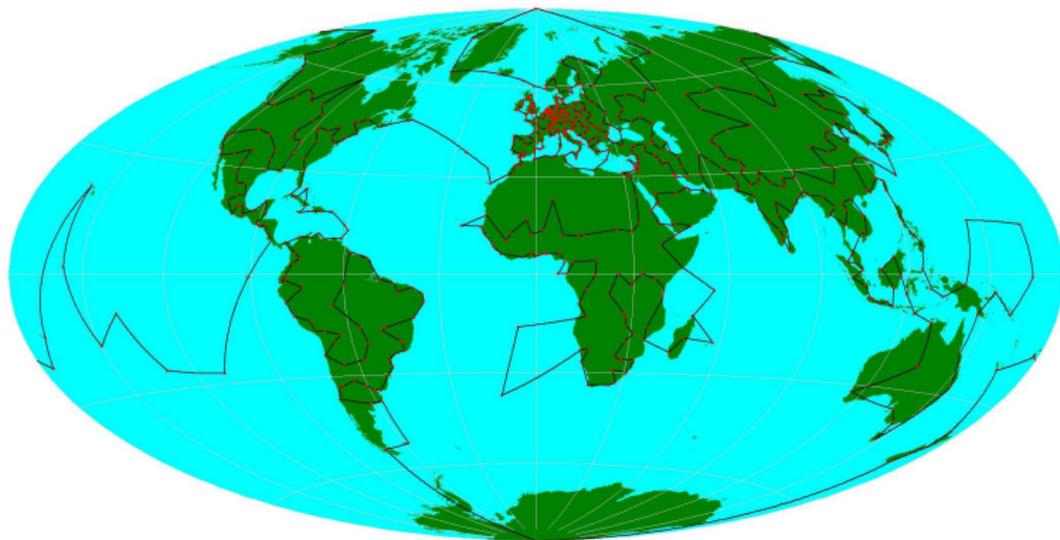


# Solved TSP instances

1987

Groetschel and Holland

666 interesting places in the world

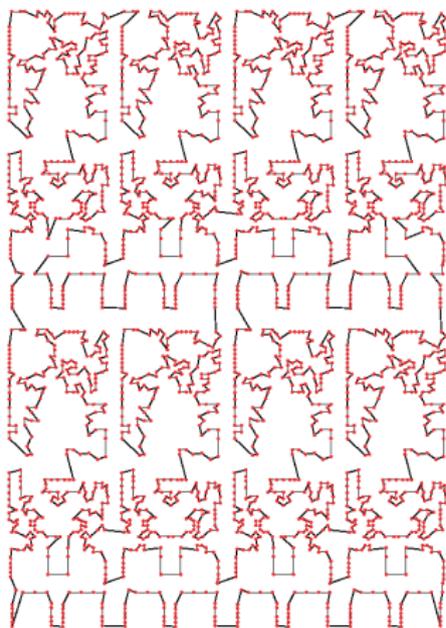


# Solved TSP instances

1987

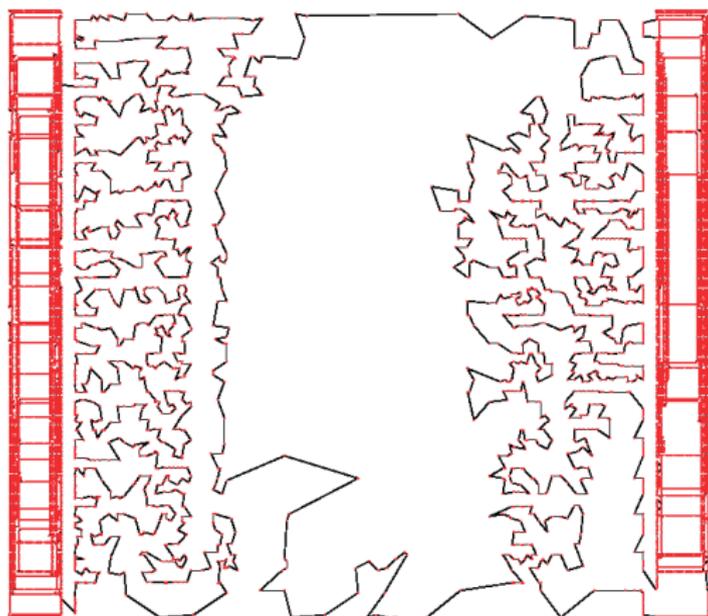
Padberg and Rinaldi

a layout of 2 392 points obtained from Tektronics Incorporated



# Solved TSP instances

1994  
D.Applegate, R.Bixby, V.Chvatal, W.Cook  
7 397 points in a programmable logic array application at AT&T Bell  
Laboratories

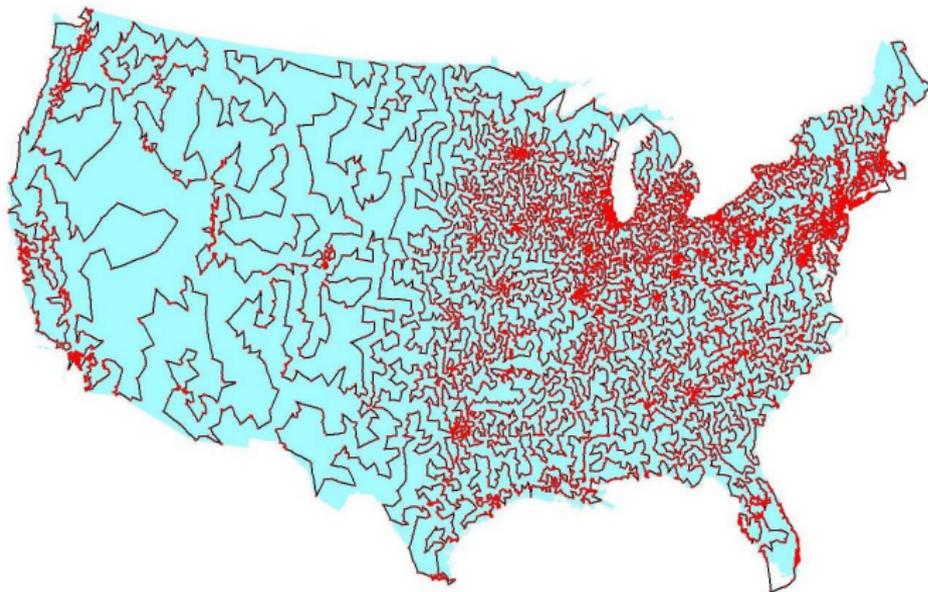


# Solved TSP instances

1998

D.Applegate, R.Bixby, V.Chvatal, W.Cook

13 509 cities in the USA with populations greater than 500

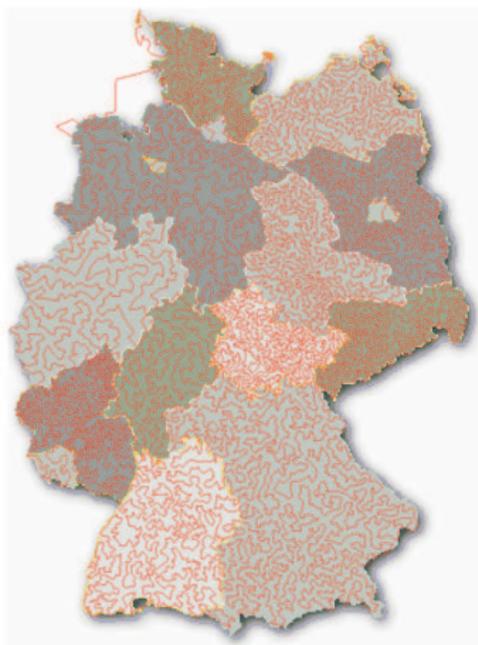


# Solved TSP instances

2001

D.Applegate, R.Bixby, V.Chvatal, W.Cook

15 112 cities in Germany



# Solved TSP instances

2004

D.Applegate, R.Bixby, V.Chvatal, W.Cook

24 978 cities in Sweden

