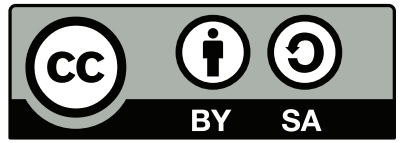
Algoritmi e Strutture Dati

Tecniche risolutive per problemi intrattabili

Alberto Montresor

Università di Trento

2018/11/07



Sommario

Chi si accontenta, gode

Proverbio

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

Voltaire, La Bégueule, 1772

Introduzione

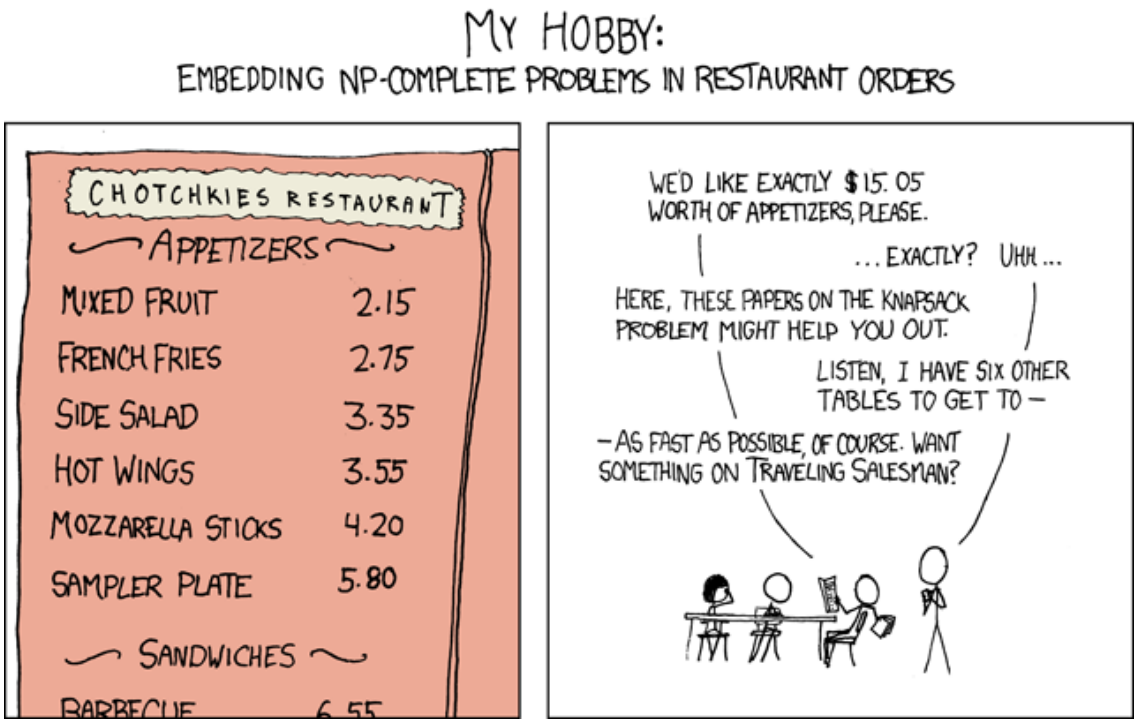
Non si può avere tutto dalla vita; bisogna rinunciare a qualcosa:

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

Subset-Sum

Somma di sottoinsieme (Subset sum)

Dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi ed un intero positivo k, esiste un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} a_i = k$?



https://xkcd.com/287/

Subset Sum

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

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

Essendo un problema decisionale, è possibile semplificare e utilizzare spazio $\Theta(k)$ invece che $\Theta(nk)$.

Subset Sum

```
boolean subsetSum(int[] A, int n, int k)
boolean[] DP = new boolean[0...k]
DP[0][0] = true
for r = 1 to k do
    DP[0][r] = false
for i = 1 to n do
    for r = A[i] to k do
        DP[i][r] = DP[i − 1][r] or DP[i − 1][r − A[i]]
return DP[n][k]
```

Subset Sum

Input: $A = [7, 11, 13], k = 24$

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

Complessità

Analisi complessità

- Complessità: $O(nk)$
- Dimensione dei dati in ingresso: $O(n \log k)$, in quanto $\forall i : A[i] \leq k$ (valori pià grandi possono essere esclusi)
- Se $k = O(n^c)$ con c costante, allora subsetSum ha complessità polinomiale $O(n^{c+1})$
- Se $k = O(2^n)$, allora subsetSum ha complessità superpolionomiale $O(n2^n)$

Osservazione

La complessità di subsetSum() non dipende soltanto dalla cardinalità n dell'insieme in ingresso, ma anche dai valori contenuti nell'insieme

Problemi fortemente 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	$\{n, k, a_1, \ldots, a_n\}$	$\max\{n, k, \max\{a_i\}\}$	$O(n \log \#)$
TSP	$\{n, k, [d_{ij}]\}$	$\max\{n, k, \max\{d_{ij}\}\}$	$O(n^2 \log \#)$
CLIQUE	$\{n, m, k, G\}$	$\max\{n, m, k\}$	$O(n + m + \log \#)$

Problemi fortemente NP-completi

Definizione

Sia R_p 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 NP-completo se R_p è NP-completo

Esempio – 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 limitato superiormente da $O(n + m)$
- Il problema ristretto è identico a CLIQUE, che è NP-completo

Problemi debolmente NP-completi

Definizione

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

Esempio – SUBSET-SUM è debolmente NP-completo

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

Algoritmi pseudo-polinomiale

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

L'algoritmo che abbiamo visto per SUBSET-SUM è pseudo-polinomiale

Teorema

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

Problemi debolmente/fortemente NP-completi

Partizione (PARTITION)

Dato un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi, esiste un sottoinsieme S di $\{1, \ldots, n\}$ 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?

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 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 NP-completo?

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

Algoritmi di approssimizzazione

Premessa

- I problemi più interessanti sono in forma di ottimizzazione
- Tali problemi sono "più difficili" dei corrispondenti problemi in forma di decisione
- Se il problema di decisione è $\mathbb{NP}$-completo, non sono noti algoritmi polinomiali per il problema di ottimizzazione ($\mathbb{NP}$-hard)
- Esistono algoritmi polinomiali che trovano soluzioni ammissibili più o meno vicine a quella ottima

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

Dato un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi (i "volumi" di n "oggetti") ed un intero positivo k (la "capacità" di una "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

$k = 8$

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:
$$\alpha(n)\frac{\sum_{i=1}^{n} a_i}{k} \leq \alpha(n)N^* < N \leq \frac{\sum_{i=1}^{n} a_i}{k/2}$$

  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

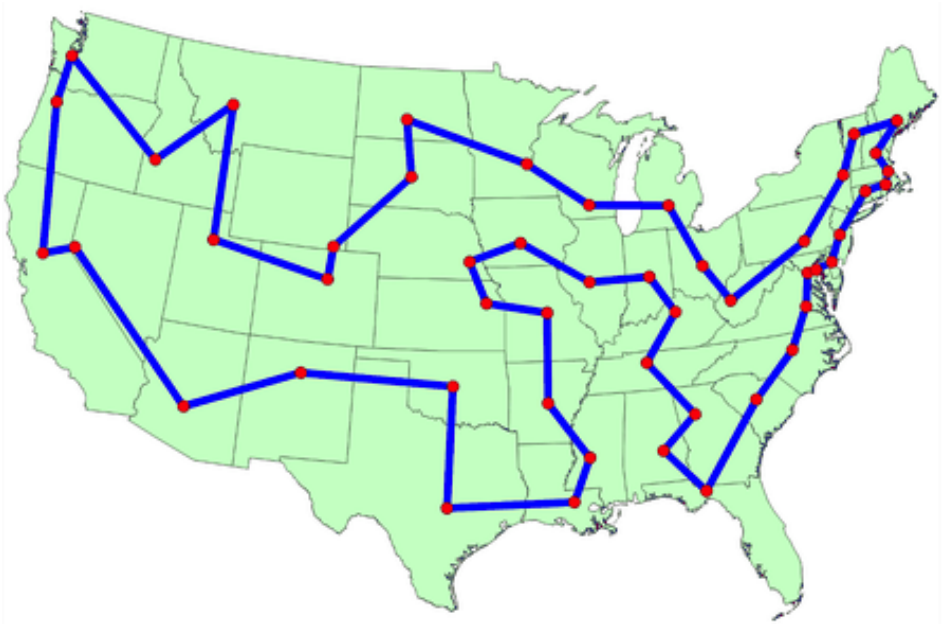
$$N < \frac{11}{9}N^* + 4$$

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

Commesso viaggiatore

Commesso viaggiatore (TSP)

Siano date n città e le distanze (positive) d_{ij} tra esse 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.

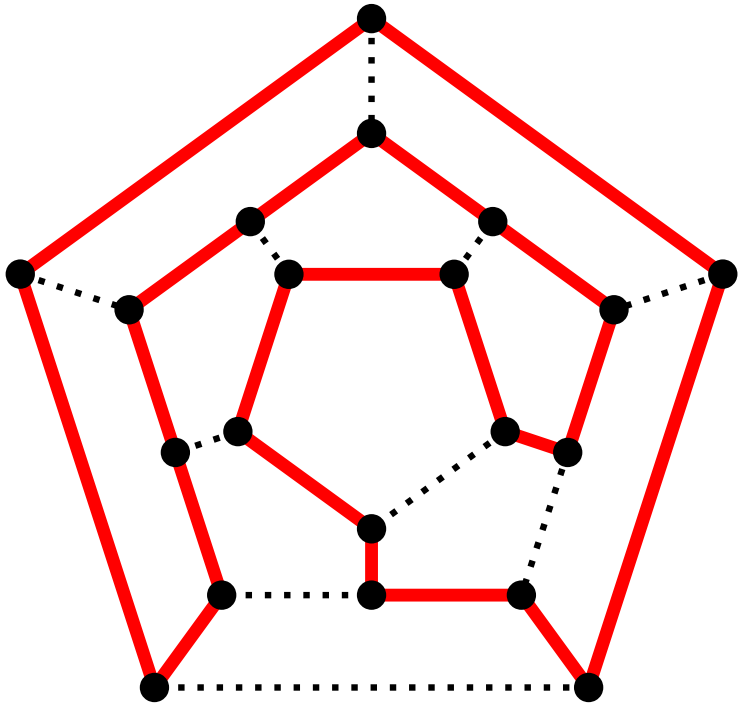


https://optimization.mccormick.northwestern.edu/images/e/ea/48StatesTSP.png

Somiglianze con circuito hamiltoniano

Circuito hamiltoniano (HAMILTONIAN-CIRCUIT)

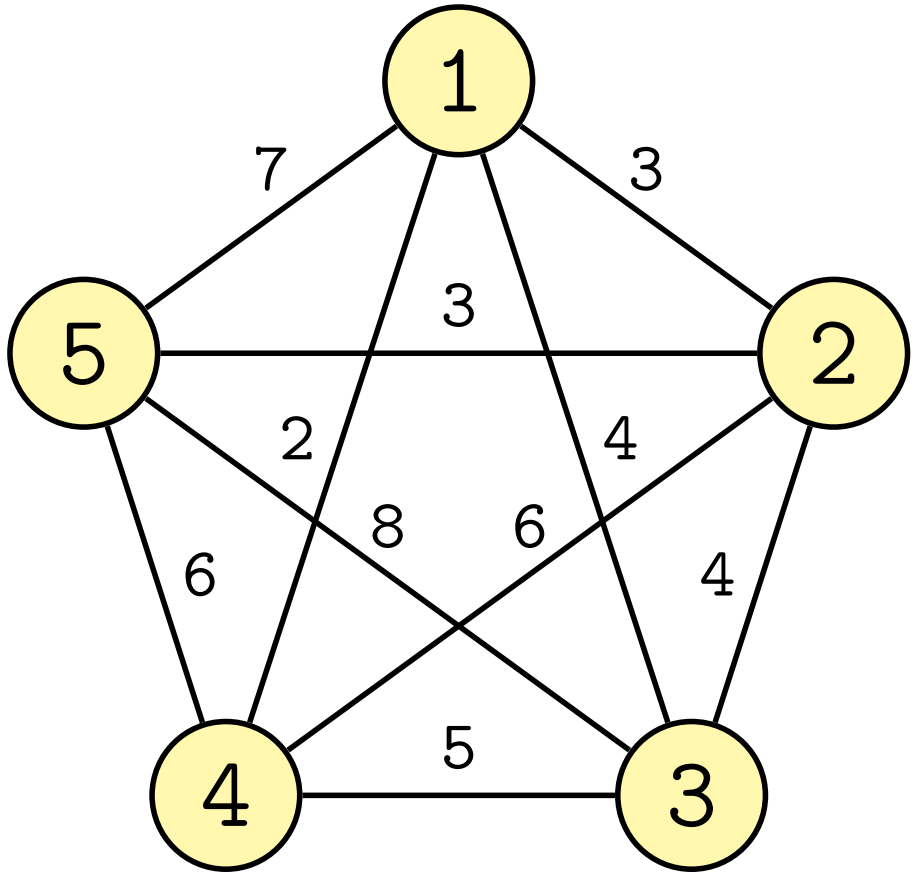
Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?



https://en.wikipedia.org/wiki/Hamiltonian_path#/media/File:Hamiltonian_path.svg

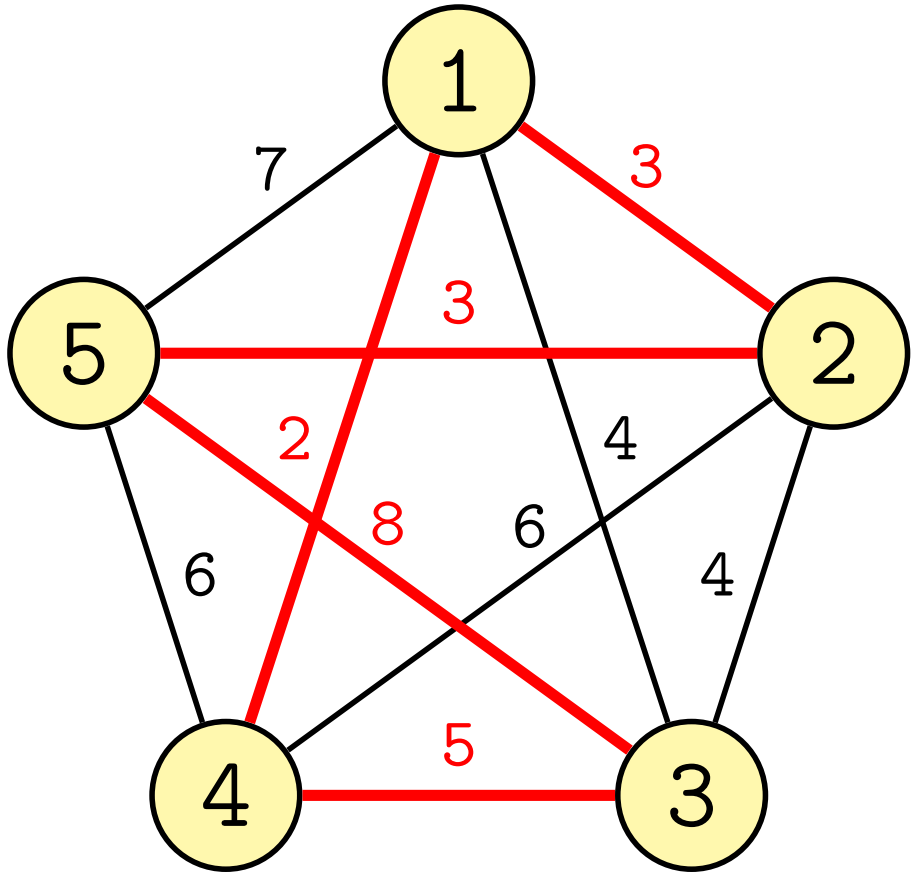
Commesso viaggiatore vs circuito hamiltoniano pesato

	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

	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 con disuguaglianze triangolari

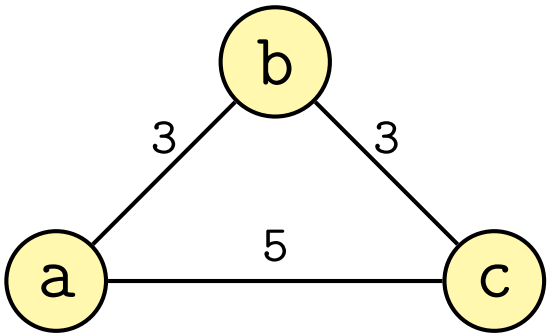
Commesso viaggiatore con disuguaglianze triangolari (Δ-TSP)

Siano date n città e le distanze (positive) d_{ij} tra esse, tali per cui vale la regola delle diseguaglianze triangolari:

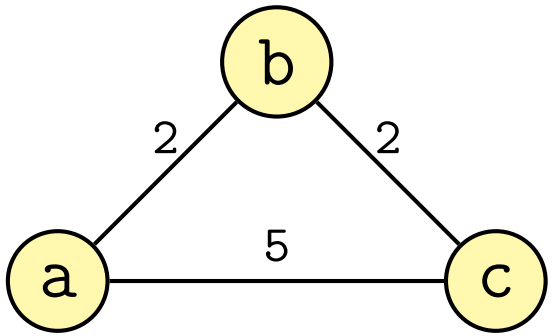
$$d_{ij} \leq d_{ik} + d_{kj} \quad \text{per} \quad 1 \leq i, j, k \leq n$$

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 disegualianza triangolare



Senza disegualianza triangolare



Δ-TSP è $\mathbb{NP}$-completo

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

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

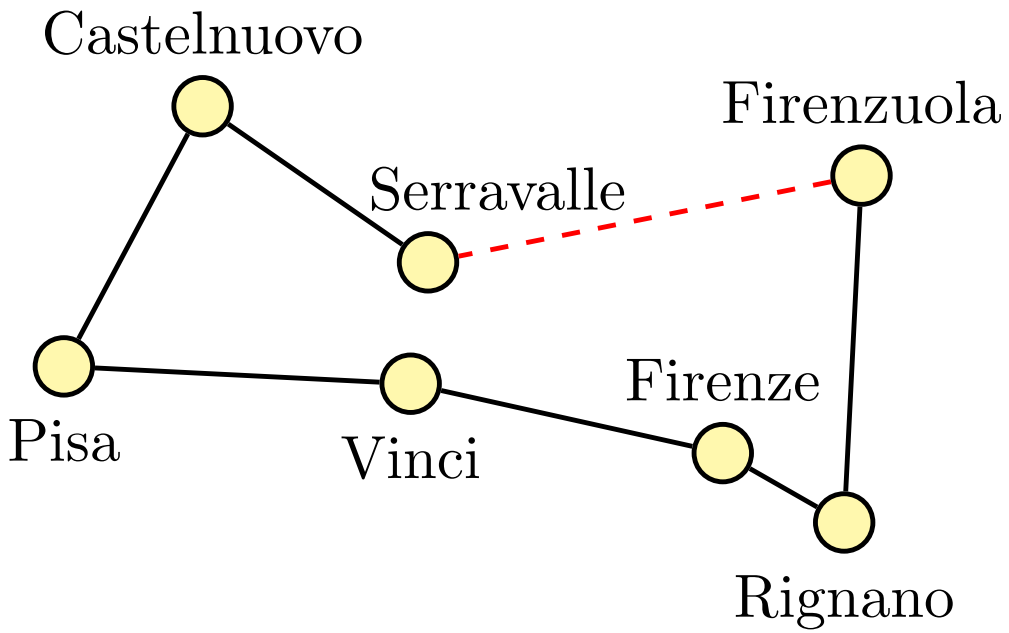
$$d_{ij} = \begin{cases} 1 & (i,j) \in E \\ 2 & (i,j) \notin E \end{cases}$$

- 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:

$$d_{ij} \leq 2 \leq d_{ik} + d_{kj}$$

Lower bound 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



Lower bound 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 $mst < c(\pi)$

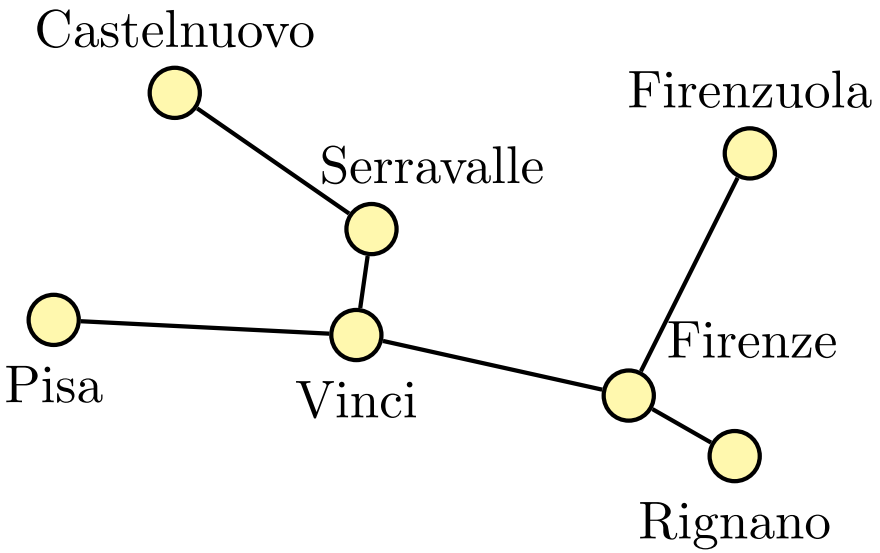
Dimostrazione

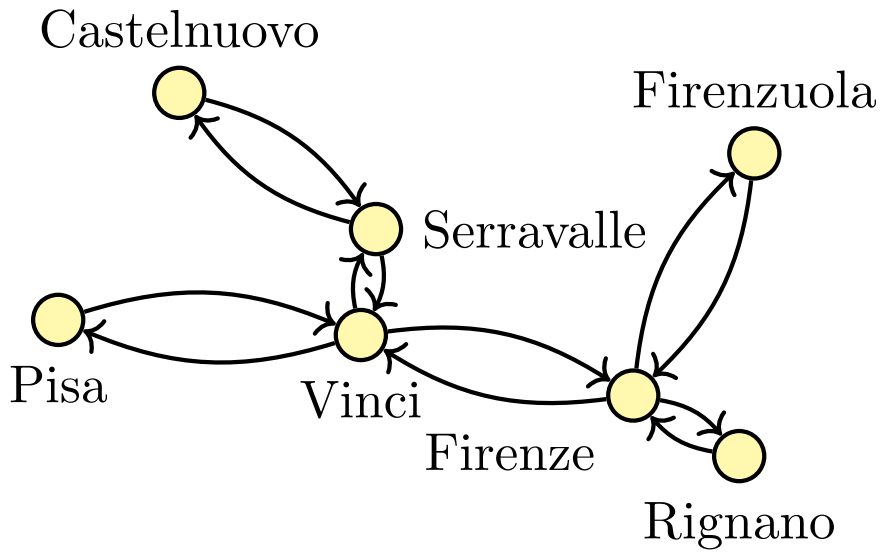
Per assurdo

- 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

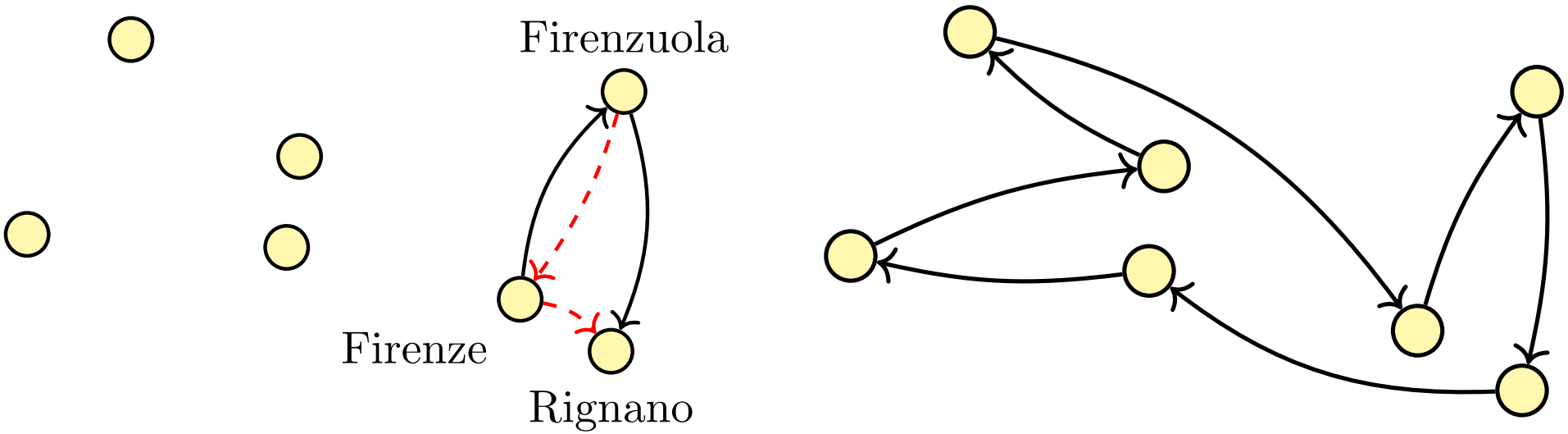
- Si individua un minimo albero di copertura di peso mst e se ne percorrono gli archi due volte, prima in un senso e poi nell'altro
- In questo modo, si visita ogni città almeno una volta
- La distanza complessiva di tale circuito è uguale a $2 \cdot mst$
- Ma non è un circuito hamiltoniano!





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 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 approssimazione assoluta per il TSP tale che COMMESSO VIAGGIATORE tale che $c(x') \leq sc(x^*)$, con $s \geq 2$ intero positivo, a meno che non sia $\mathbb{P} = \mathbb{NP}$.

Dimostrazione

- Ragionamo per assurdo; supponiamo che un tale algoritmo A esista
- Si consideri la riduzione HAMILTONIAN-CIRCUIT $\leq_p$ TSP
- Ovvero, sia dato un grafo $G = (V, E)$ e sia $[d_{ij}]$ una matrice di distanze tale che:

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

Non approssimabilità di TSP

Teorema

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

Dimostrazione

- Due casi:
  - Se G ha un circuito Hamiltoniano, allora $c(x^*) = n$, quindi A fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \leq sc(x^*) = sn$
  - Se G non ha un circuito Hamiltoniano, allora $c(x^*) > sn$, quindi A fornisce, in tempo polinomiale, una soluzione x' tale che $c(x') \geq c(x^*) > sn$
- Quindi A risolve HAMILTONIAN-CIRCUIT in tempo polinomiale, impossibile a meno che $\mathbb{P} = \mathbb{NP}$

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

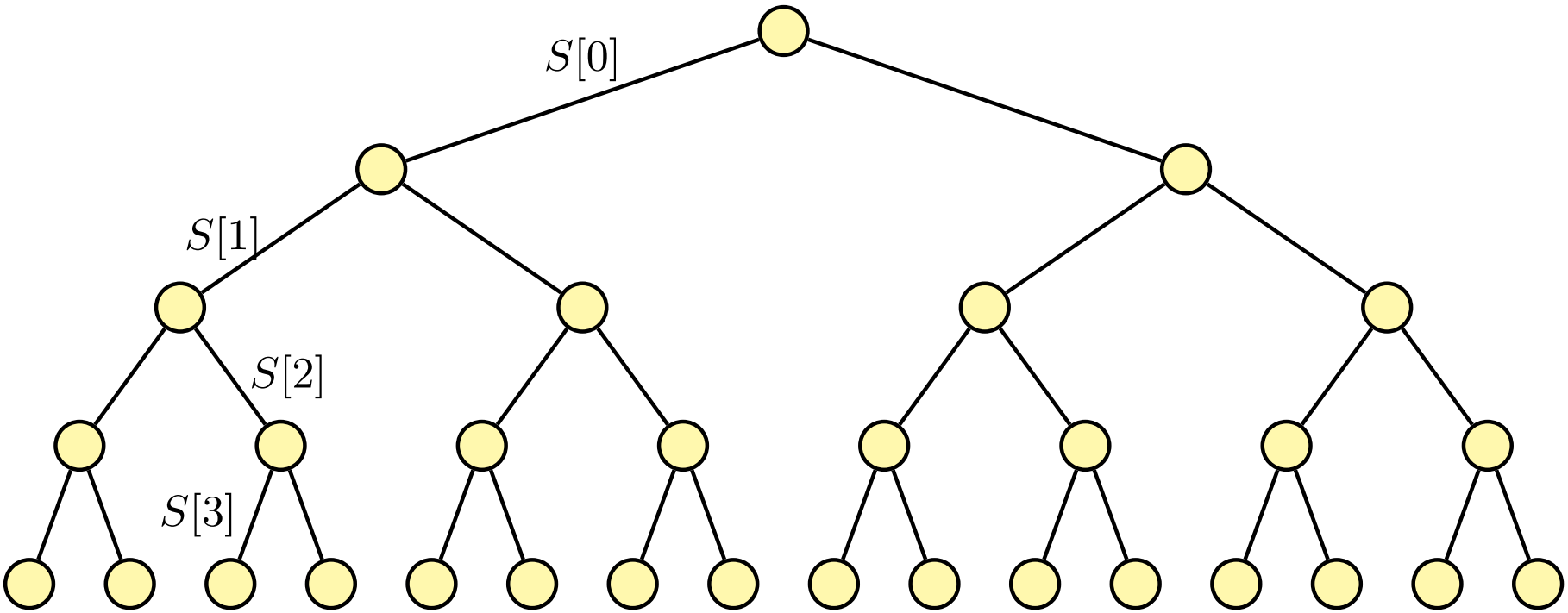
Alcune assunzioni – senza perdere (troppa) generalità

- 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

```
boolean enumeration(ITEM[] S, int n, int i, ...)
SET C = choices(S, n, i, ...)          % Determina C in funzione di S[1...i-1]
foreach c ∈ C do
    S[i] = c
    if isAdmissible(S, n, i) then
        if processSolution(S, n, i, ...) then
            return true
    if enumeration(S, n, i + 1, ...) then
        return true
return false
```

Backtrack: ripasso



Branch-&-bound

Upper bound per il costo della soluzione minima

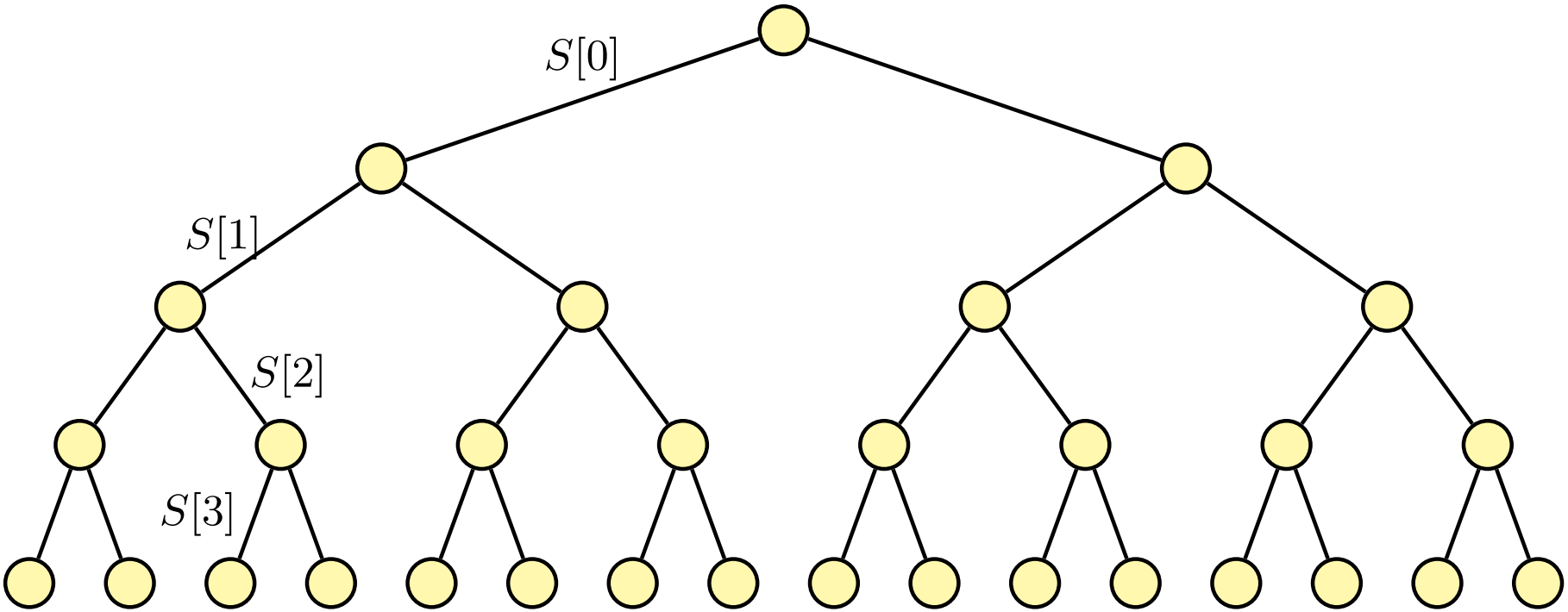
- Durante l'enumerazione, mantengo informazioni sulla miglior soluzione ammissibile *minSol* e il suo costo *minCost*
- *minCost* costituisce un limite superiore (upper bound) per il costo della soluzione minima

Lower bound per il costo della soluzione minima

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

Branch-&-bound

Se $\mathsf{lb}(S, i)$ è maggiore o uguale a $minCost$, allora si può evitare di generare ed esplorare il sottoalbero delle scelte radicato in tal nodo



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 essere il più vicino possibile al costo della soluzione ottima

Schema generale

branch&bound(ITEM[] S, **int** n, **int** i, ...)

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

Commesso viaggiatore – Branch-&-bound

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

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

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

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

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

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

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

$$D[h] = \min_{p,q}\{d[p, h] + d[h, q] : h \neq p \neq q\}, \text{ per ogni } h \notin S.$$

Lower bound

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

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

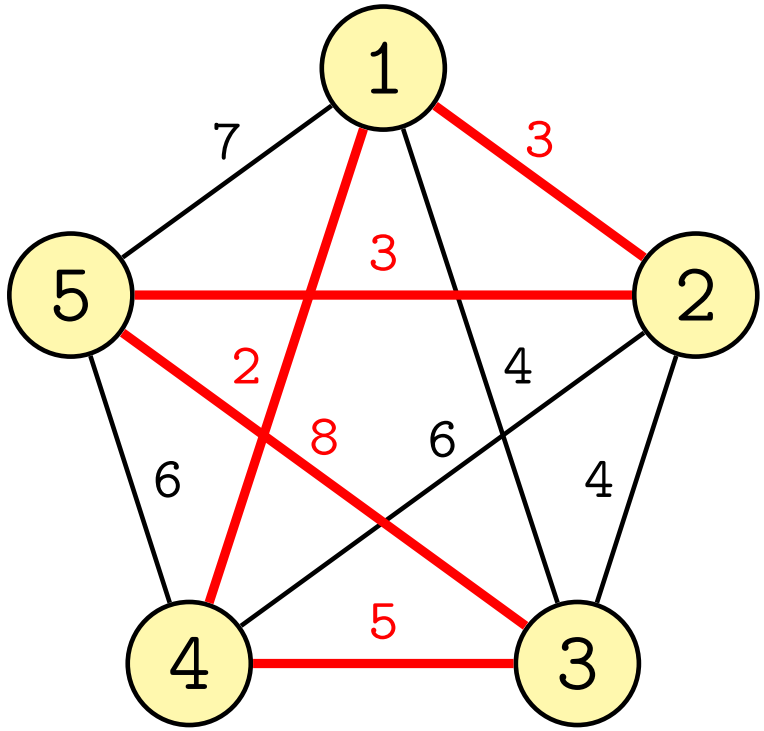
Commesso viaggiatore – Branch-&-bound

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

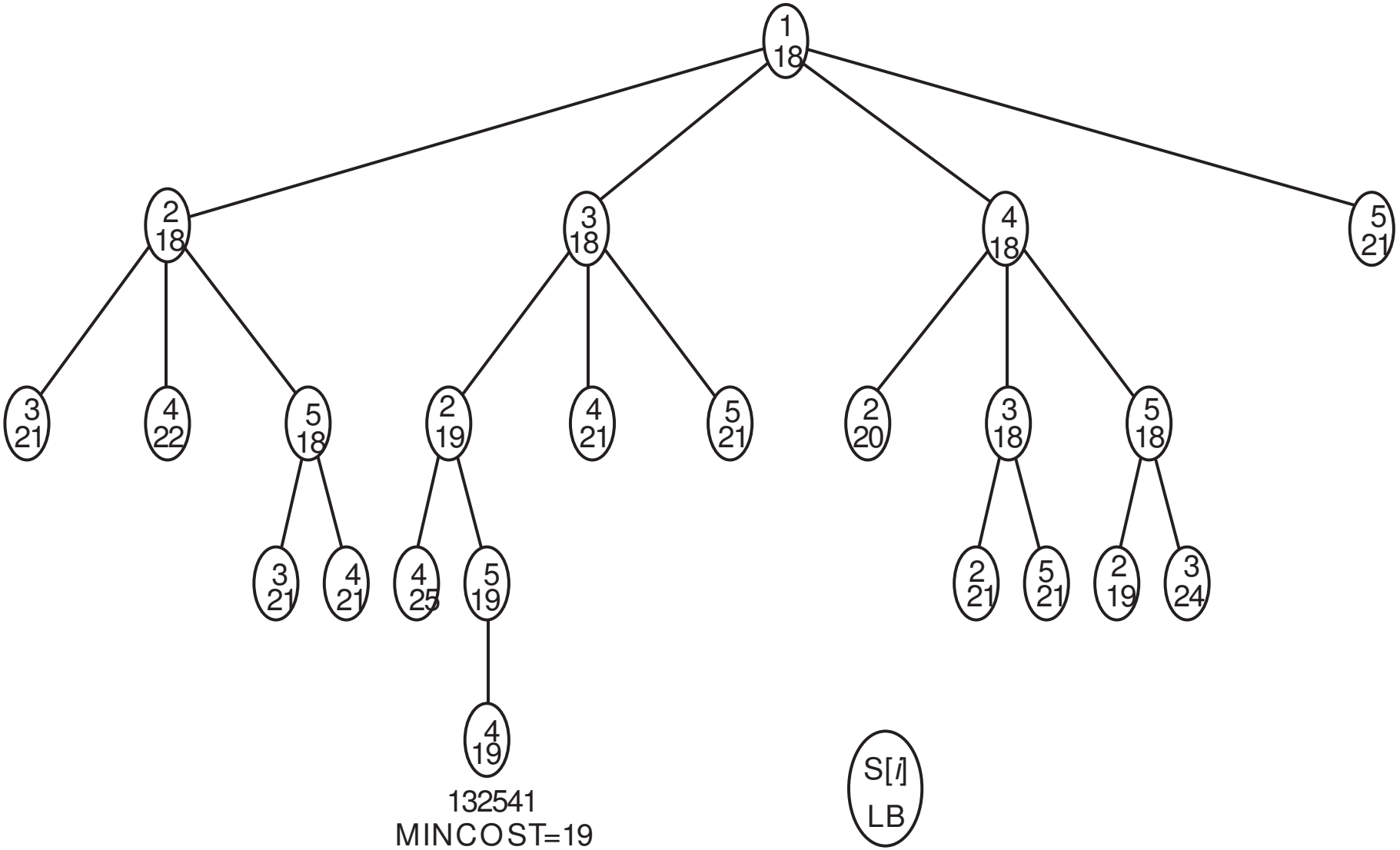
foreach $c \in R$ **do**
- $S[i] = c$
- R.remove(c)
- $C[i] = C[i-1] + d[S[i-1]][S[i]]$
- {calcola A, B, e $D[h]$ per ogni $h \in R$}
- **int** $lb = C[i] +$ iif$(i < n, \lceil (\sum_{h \notin S} D[h])/2 \rceil, d[S[i]][S[1]])$
- **if** $lb < minCost$ **then**
  - **if** $i < n$ **then**
    - bbTsp$(S, C, R, n, i+1)$
  - **else**
    - $C[n] = lb$
    - $minSol = S$
    - $minCost = C[n]$
- R.insert(c)

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
18
2
18
3
18
4
18
5
21
3
21
4
22
5
18
2
19
4
21
5
21
2
20
3
18
5
18
3
21
4
21
4
25
5
19
2
21
5
21
2
19
3
24
4
19
132541
MINCOST=19
S[l]
LB

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

Algoritmi euristici

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

- non necessariamente ottima
- non necessariamente approssimata

Tecniche possibili

- Greedy
- Ricerca locale

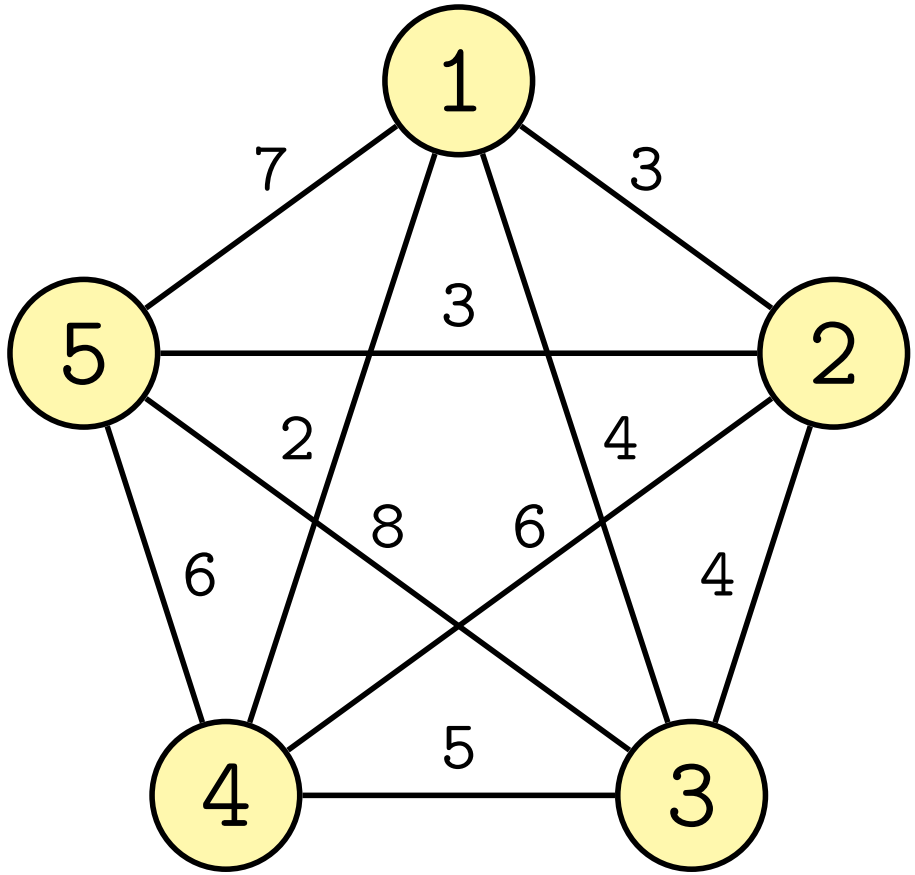
TSP – Greedy (1)

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

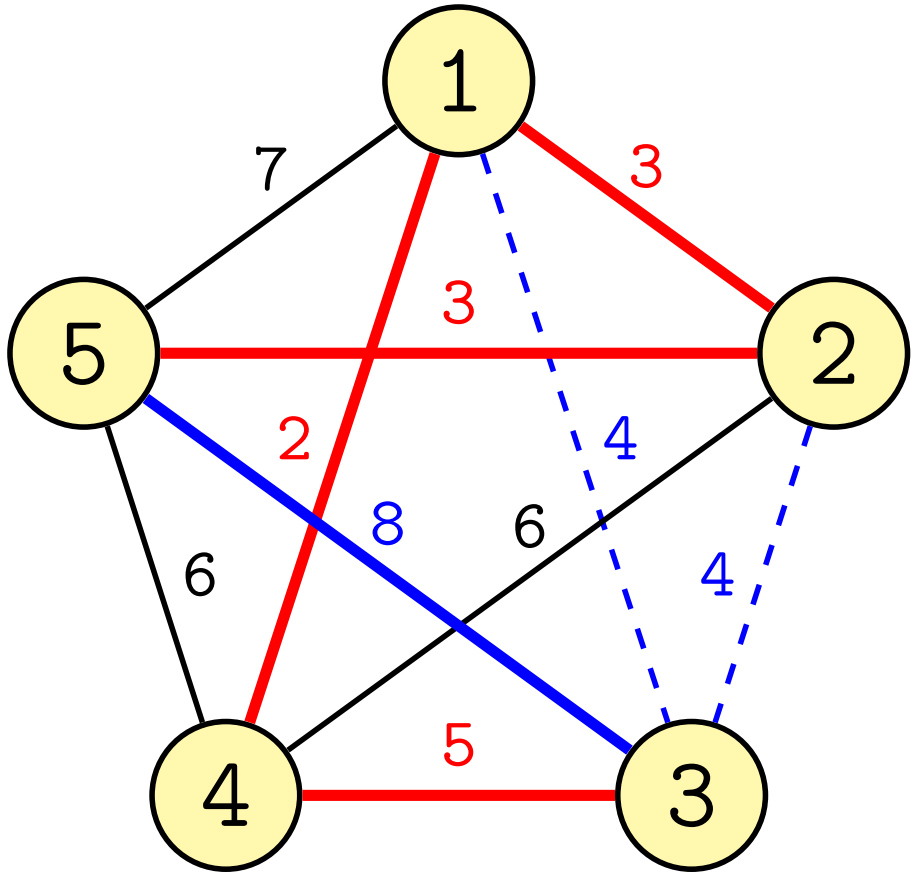
Esempio

	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	



Costo: 21

TSP – Greedy (1)

SET greedyTsp(GRAPH G)

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

TSP – Greedy (1)

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

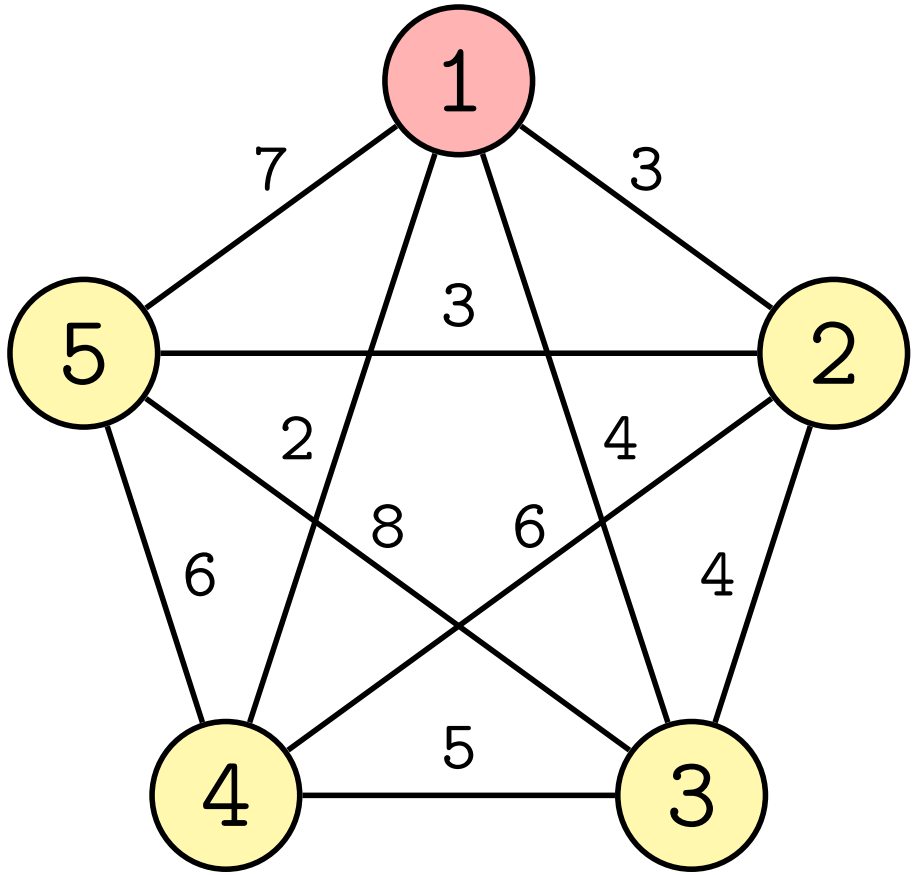
TSP – Greedy (2)

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
- Costo: $O(n^2)$

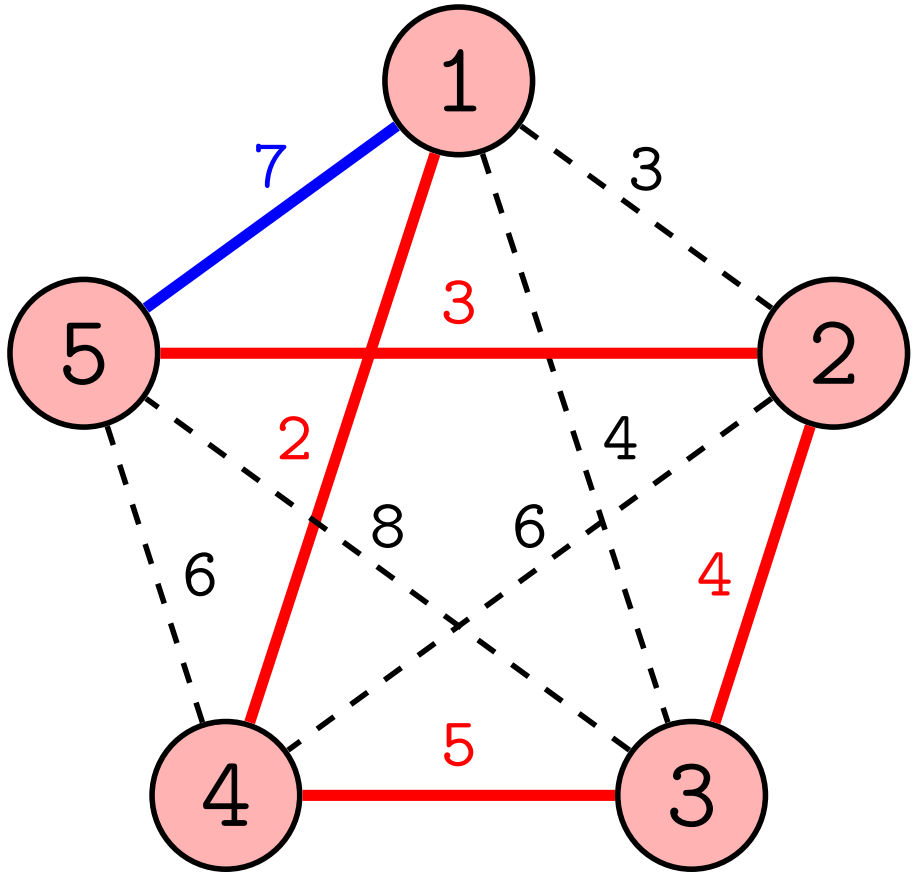
TSP – Greedy (2)

	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 – Approccio ricerca locale

Ricerca locale

Sia π un circuito Hamiltoniano del grafo completo derivante dal problema TSP. Un'euristica ragionevole di ricerca locale può essere progettata sulla base del seguente intorno:

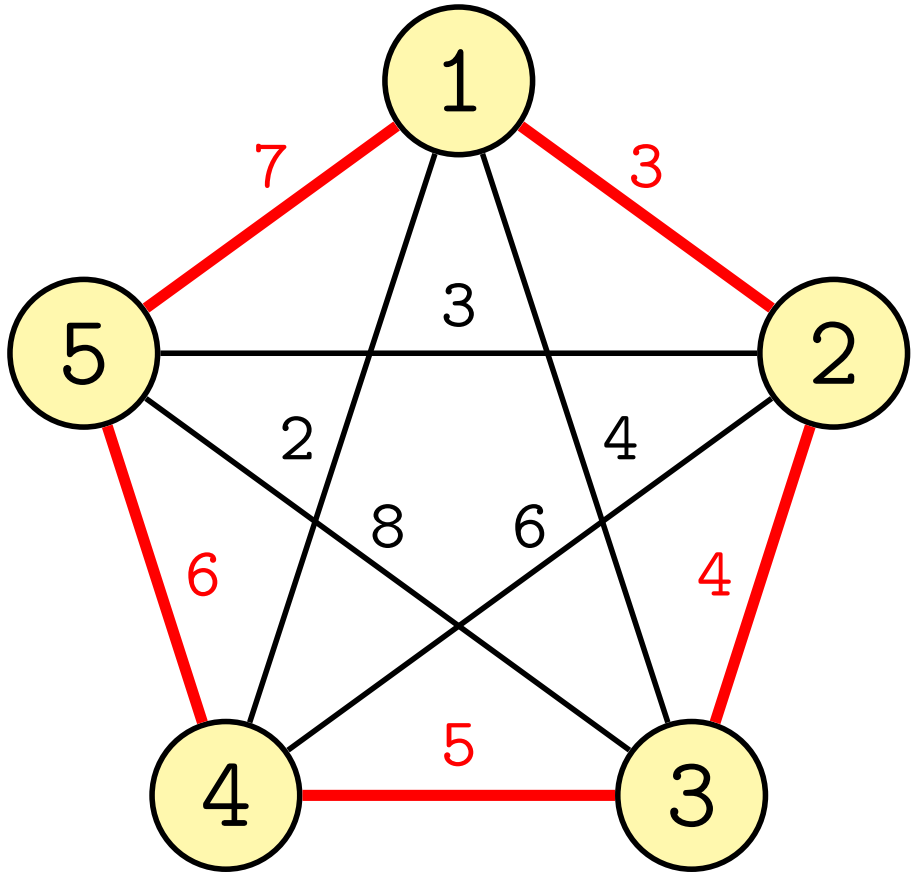
$$I_2(\pi) = \{\pi' : \pi' \text{ è ottenuto da } \pi \text{ cancellando due archi non consecutivi del circuito e sostituendoli con due archi esterni al circuito}\}$$

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

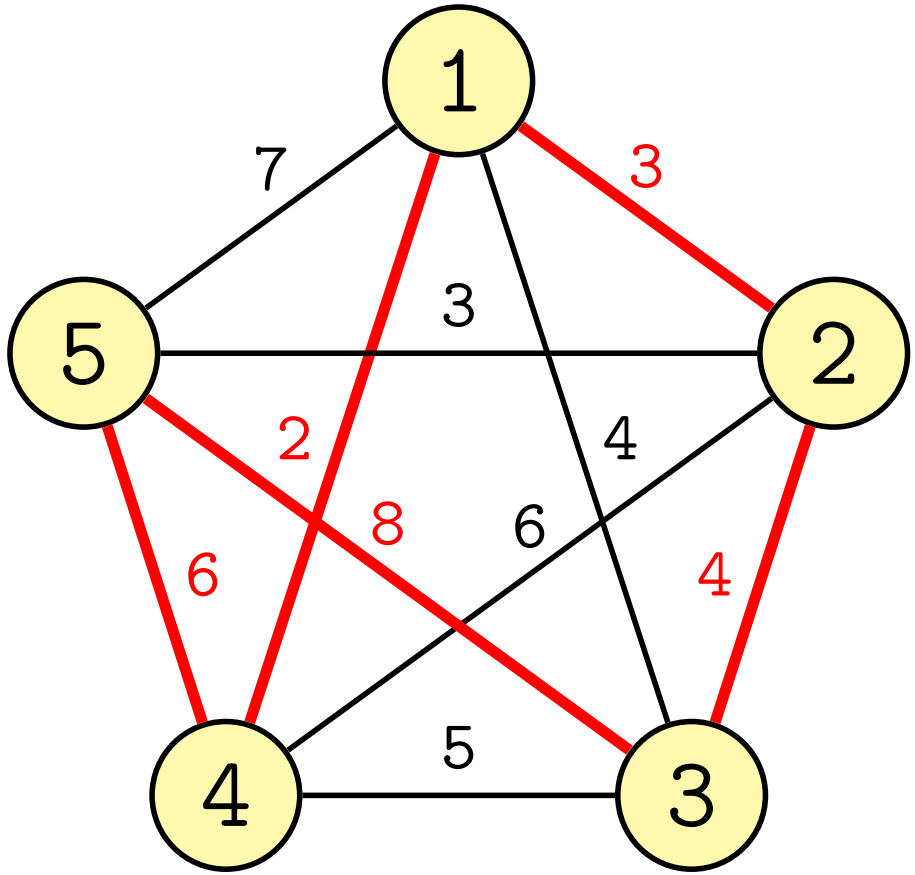
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 25

Esempio

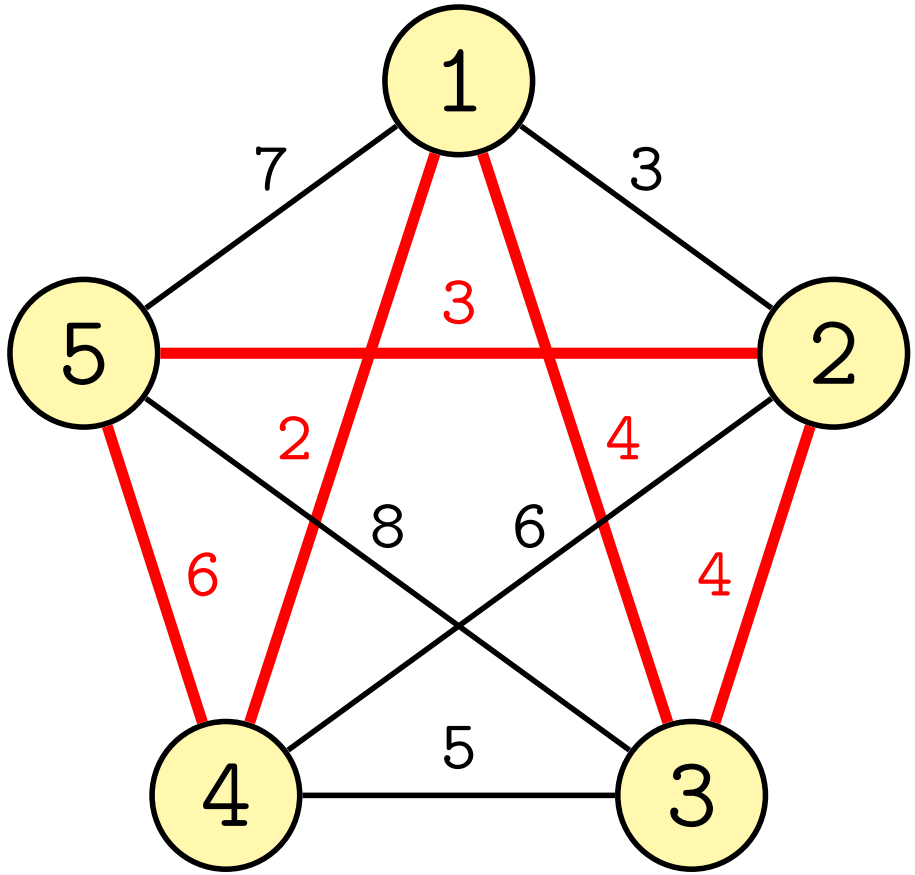
	1	2	3	4	5
1		3	4	2	7
2	3		4	6	3
3	4	4		5	8
4	2	6	5		6
5	7	3	8	6	



Costo: 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

Spunti di lettura

Rego, César; Gamboa, Dorabela; Glover, Fred; Osterman, Colin (2011), "Traveling salesman problem heuristics: leading methods, implementations and latest advances", European Journal of Operational Research, 211 (3): 427–441. https://www.sciencedirect.com/science/article/pii/S0377221710006065?via%3Dihub

David Williamson, David Shmoys (2010). The Design of Approximation Algorithms. Cambridge University Press. http://www.designofapproxalgs.com/book.pdf

David Williamson, David Shmoys (2010). The Design of Approximation Algorithms. Cambridge University Press. http://www.designofapproxalgs.com/book.pdf