Traveling Salesman Problem

Giacomo Bonazzi - Marco Caldera

Università degli Studi di Torino

Ottimizzazione Combinatoria

17 dicembre 2019

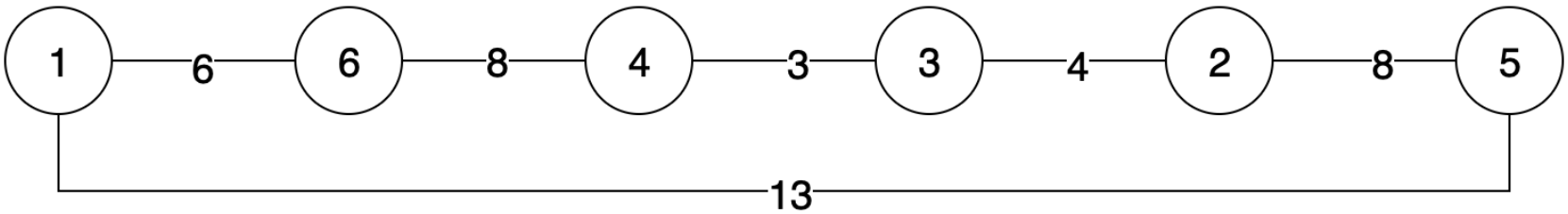
Outline

Problema del commesso viaggiatore

Dato:

- Un insieme di **città**
- Un insieme di **distanze** tra ciascuna coppia di città

bisogna trovare il tragitto di lunghezza minima che deve compiere un commesso viaggiatore in modo che vengano visitate tutte le città una ed una sola volta.



Risorse

Per testare la correttezza degli algoritmi abbiamo utilizzato la libreria **TSPLIB** che contiene **istanze** con **risultati** e **percorsi ottimi** per il problema del TSP

Abbiamo scelto 5 problemi su cui concentrarci:

- berlin52 [7542]
- eil101 [629]
- kroA100 [21282]
- pr76 [108159]
- st70 [675]

Distanza euclidea

La distanza tra i nodi viene calcolata attraverso la formula della **distanza euclidea**:

$$\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Ogni nodo ha due coordinate, x ed y, da cui è possibile calcolare la distanza che ogni altro nodo ha da esso.

Preparazione risorse

Fase 1

I file delle risorse sono file **.tsp** che contengono determinate informazioni sul problema e l'insieme di dati su cui esso è costruito.

Lo scopo di questa parte è quello di trasformare ogni file *.tsp* in un file **.csv** leggibile agevolmente con Python.

Ogni riga di questo file contiene:

- Numero del nodo
- Coordinate del nodo

Preparazione risorse

Fase 2

Nella seconda fase abbiamo invece creato degli oggetti python contenenti il **percorso ottimo** del problema.

In questo modo durante i test abbiamo potuto facilmente confrontare l'ottimo con il risultato della computazione.

```
st70_opt = map(lambda x: x - 1, [1, 36, 29, 13, ...])
pr76_opt = map(lambda x: x - 1, [1, 76, 75, 2, ...])
```

Outline

Strutture dati - Matrice

Come prima cosa abbiamo scelto le strutture dati per rappresentare e manipolare il problema.

Le istanze vengono rappresentate sotto forma di **matrice** $n \times n$ con n numero di nodi del problema.

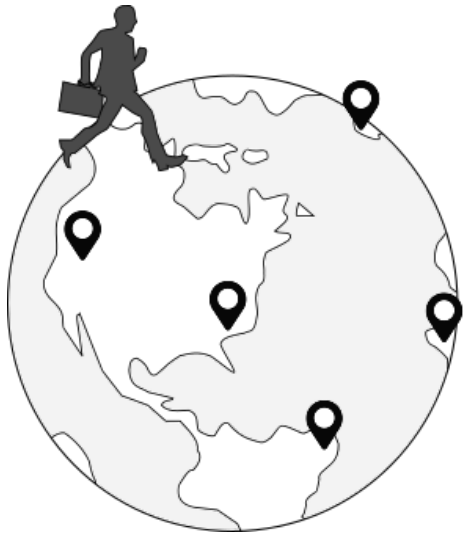
$$\begin{bmatrix} \infty & 666.0 & 281.0 & 396.0 & 291.0 \\ 666.0 & \infty & 649.0 & 1047.0 & 945.0 \\ 281.0 & 649.0 & \infty & 604.0 & 509.0 \\ 396.0 & 1047.0 & 604.0 & \infty & 104.0 \\ 291.0 & 945.0 & 509.0 & 104.0 & \infty \end{bmatrix}$$

Strutture dati - Vettore Soluzione

Il risultato della computazione è invece fornito sotto forma di **array**. Una lista contenente i nodi, in un certo ordine, tali da formare un determinato percorso ammissibile.

$$[0,\ 2,\ 4,\ ...,\ 3,\ 1]$$

Il percorso rappresenta il tour che il commesso viaggiatore deve effettuare per visitare tutti i nodi e tornare a quello di partenza.



Outline

Nearest Neighbor v1

L'euristica più famosa per costruire un tour iniziale è **nearest neighbor**.

L'algoritmo partendo da una nodo iniziale visita il nodo più vicino, non ancora esaminato, e prosegue sino ad ispezionare tutti i nodi e tornare a quello di partenza.

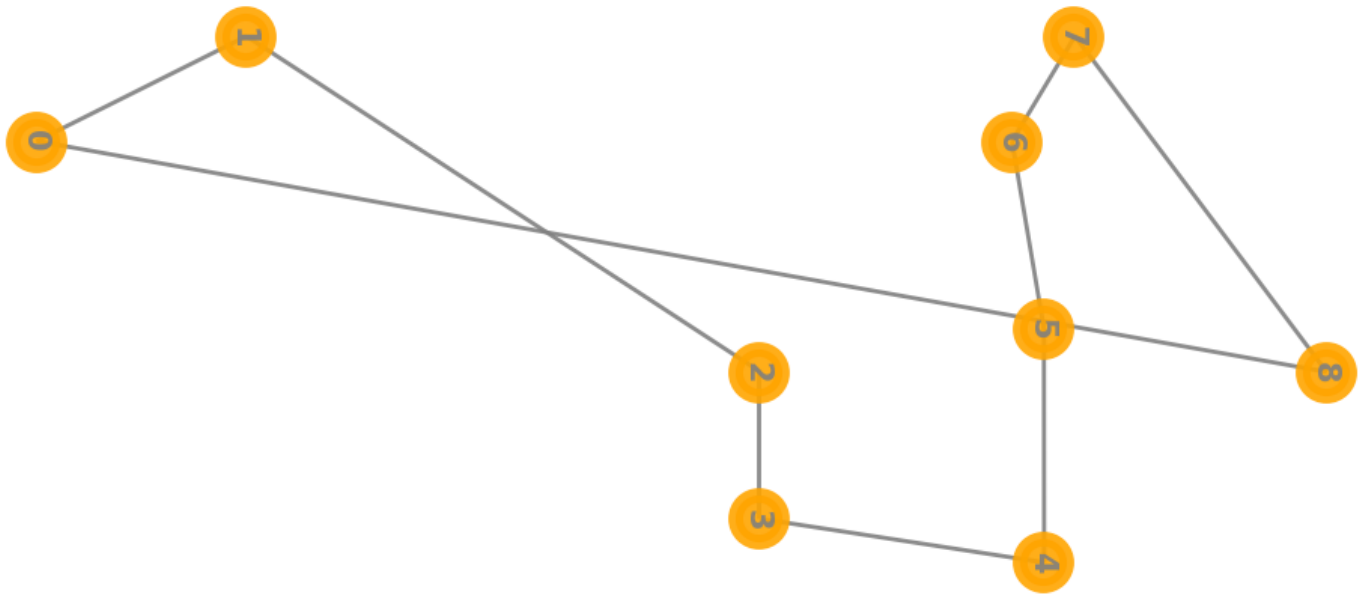


Figura: Esempio di NN partendo dal nodo 0

Nearest Neighbor v2

Per ottenere una migliore soluzione iniziale si può pensare di eseguire **NN partendo da ogni nodo** e restituire la soluzione più promettente.

Tendenzialmente si preferisce questo secondo approccio che garantisce una migliore soluzione iniziale e quindi maggiori garanzie di ottenere una buona soluzione finale.

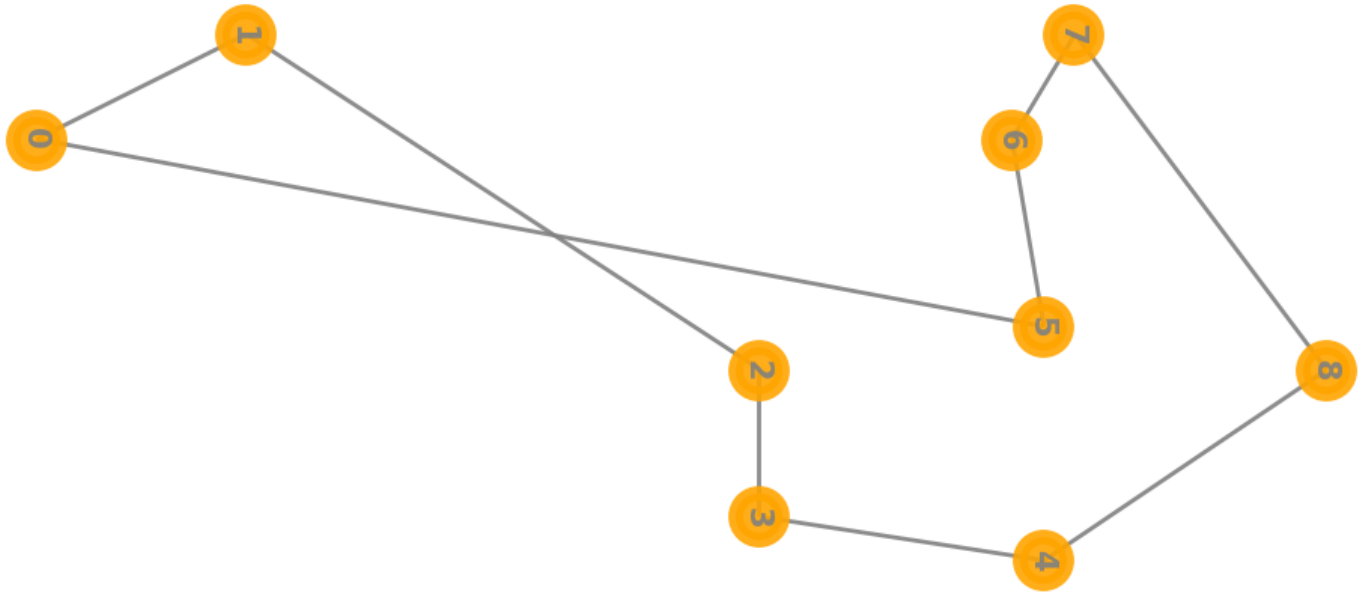


Figura: Esempio di soluzione più promettente

Random Tour v3

Un terzo approccio per ottenere la soluzione iniziale, al fine di velocizzarne la sua creazione, è quello di generare un **tour in modo randomico**.

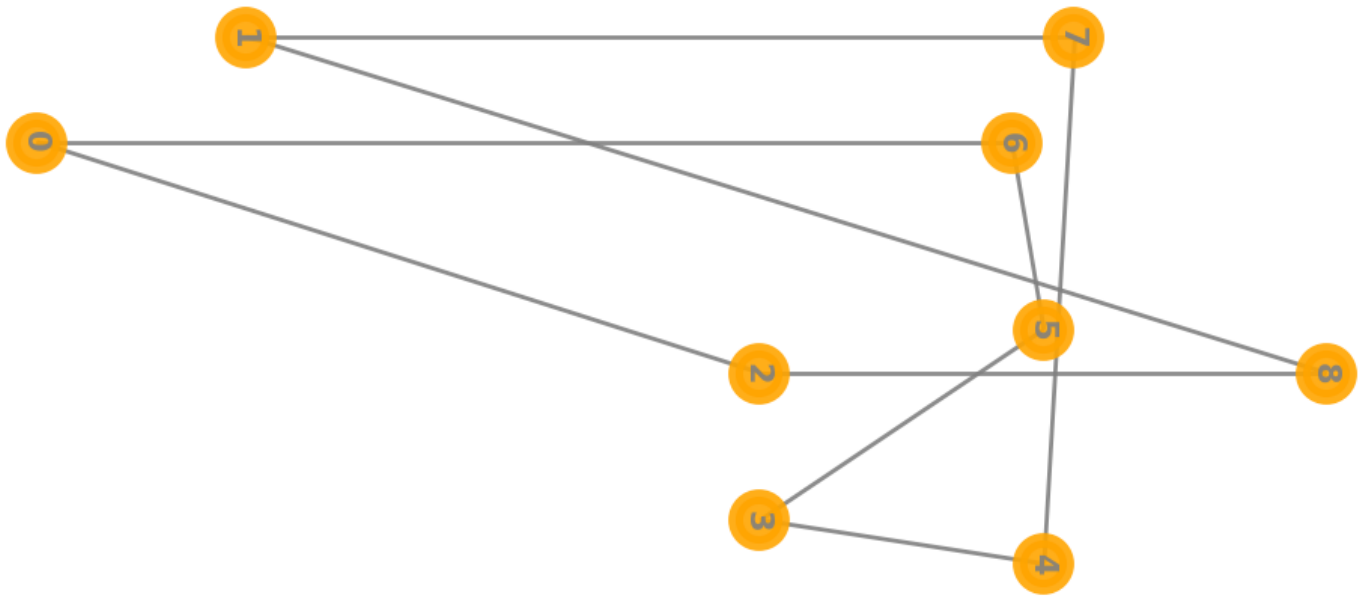


Figura: Esempio di soluzione generata randomicamente

Vantaggi e svantaggi

Nearest Neighbor v1

Costo computazionale: $O(n^2)$
Soluzione iniziale: **buona**

Nearest Neighbor v2

Costo computazionale: $O(n^3)$
Soluzione iniziale: **molto buona**

Random Tour

Costo computazionale: $O(n)$
Soluzione iniziale: **pessima**

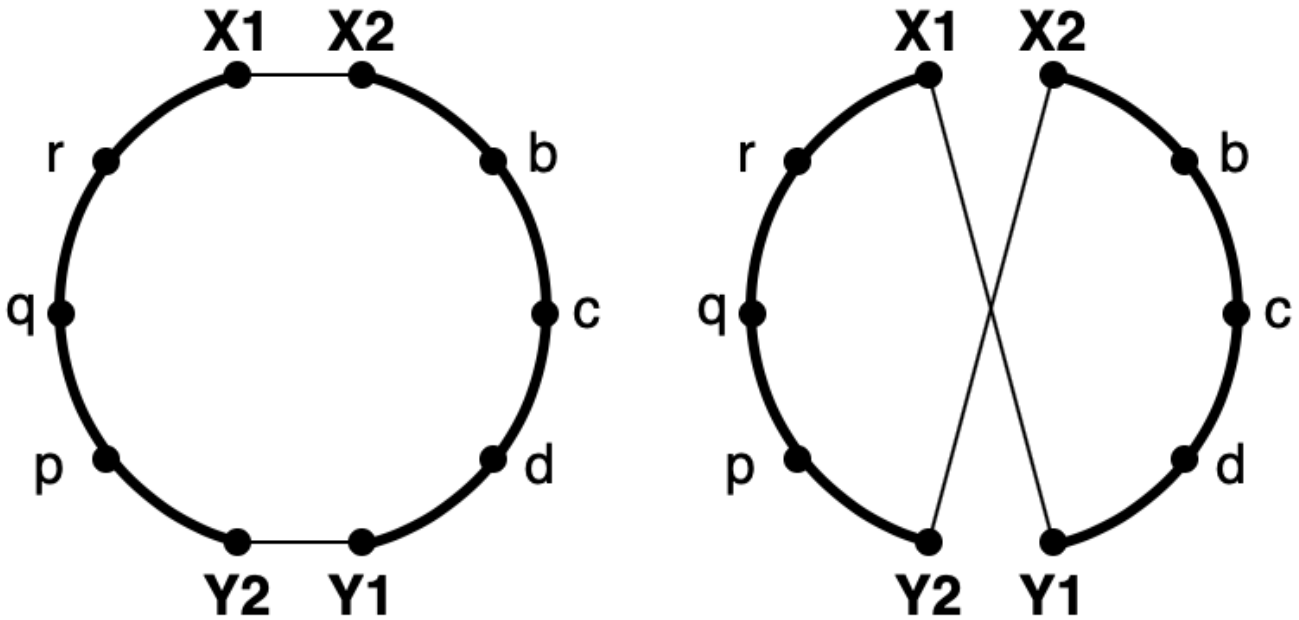
Outline

2-Opt

Il primo intorno sviluppato è **2-Opt**.

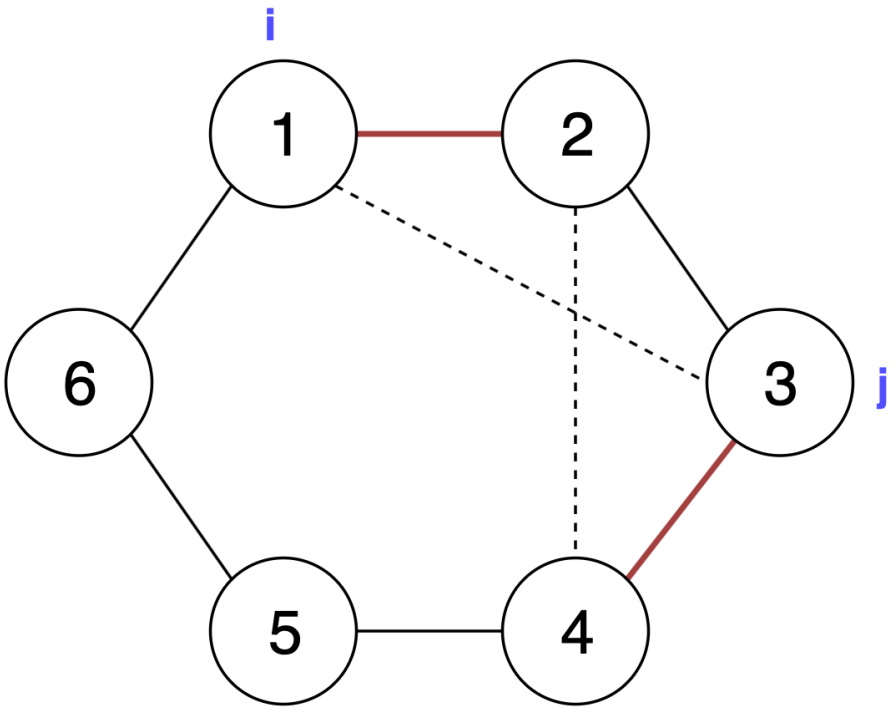
Per ogni ogni **coppia di archi non consecutivi**:

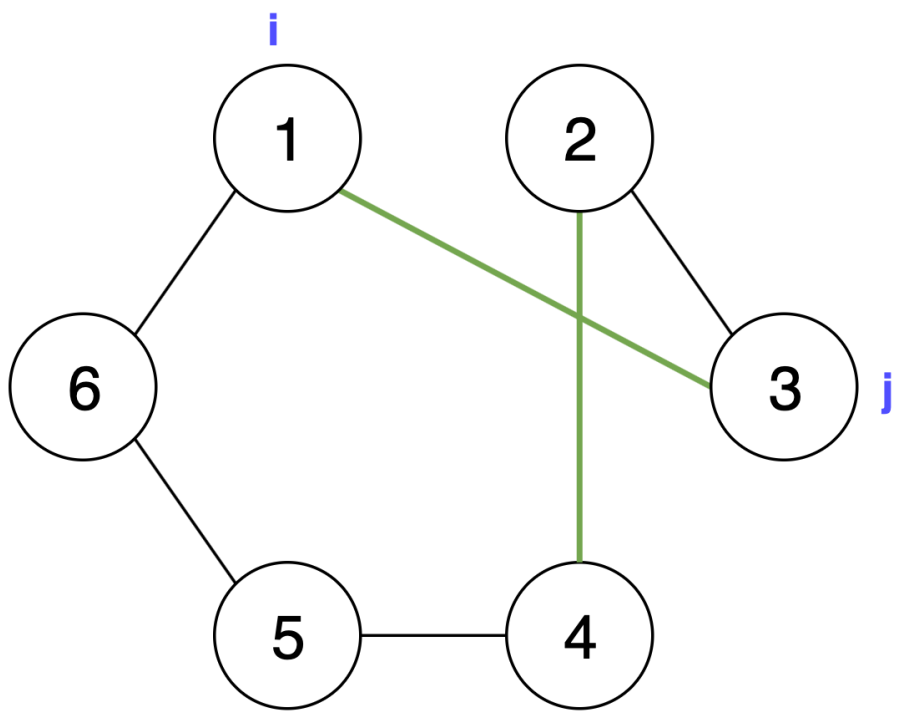
- Si **eliminano** i due archi
- Si **riconnette** il percorso aggiungendo due nuovi archi che permettano di rispettare i vincoli del problema



2-Opt esempio

```
for i in range(0, len(route) - 3):
    for j in range(i + 2, len(route) - 1):
        reverse = route[i + 1:j + 1]
        new_route = route[:i + 1] + reverse[::-1] + route[j + 1:]
```

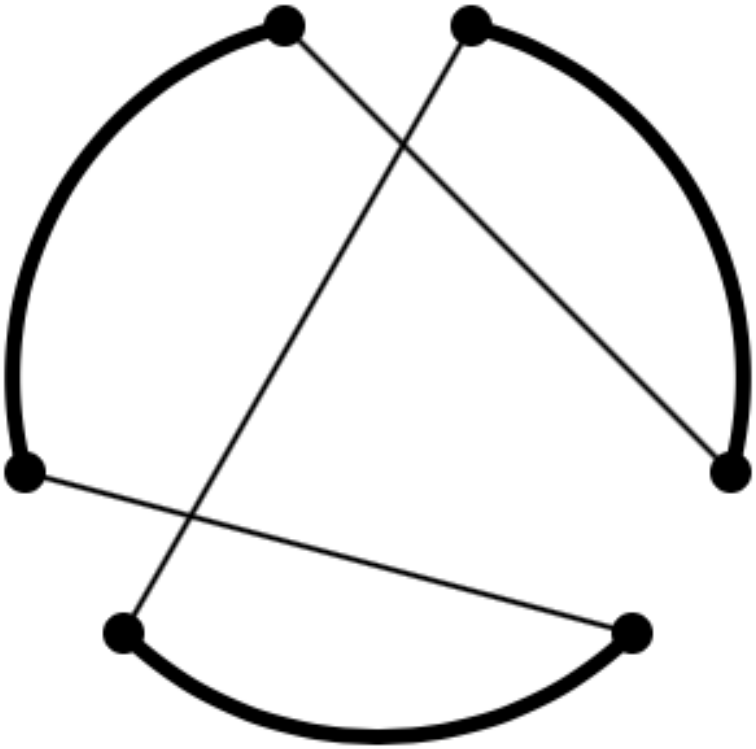




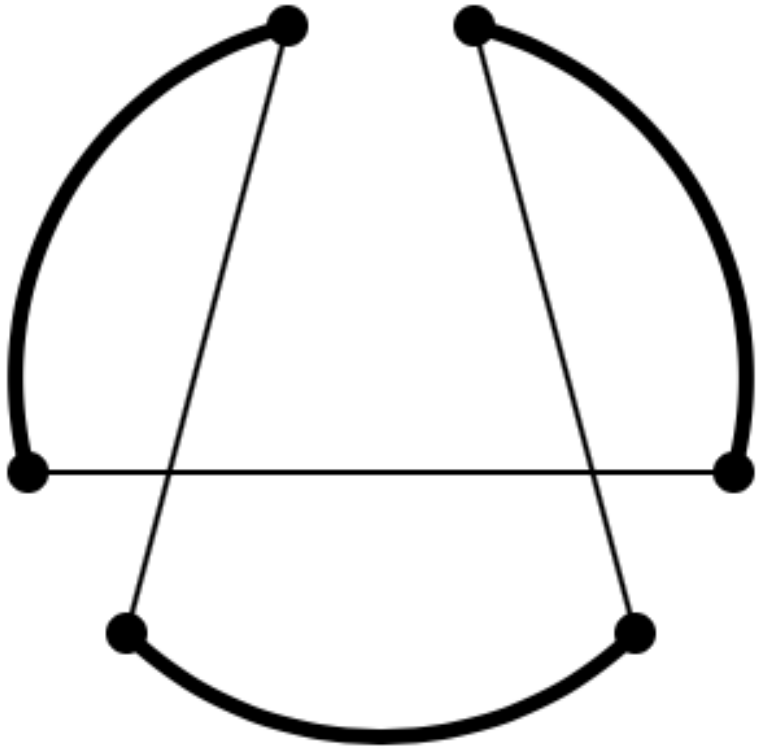
Archi considerati: (1,2) (3,4) Percorso risultante: $[1, 3, 2, 4, 5, 6, 1]$

3-Opt

Per quanto riguarda l'intorno **3-Opt** abbiamo quattro casi possibili:

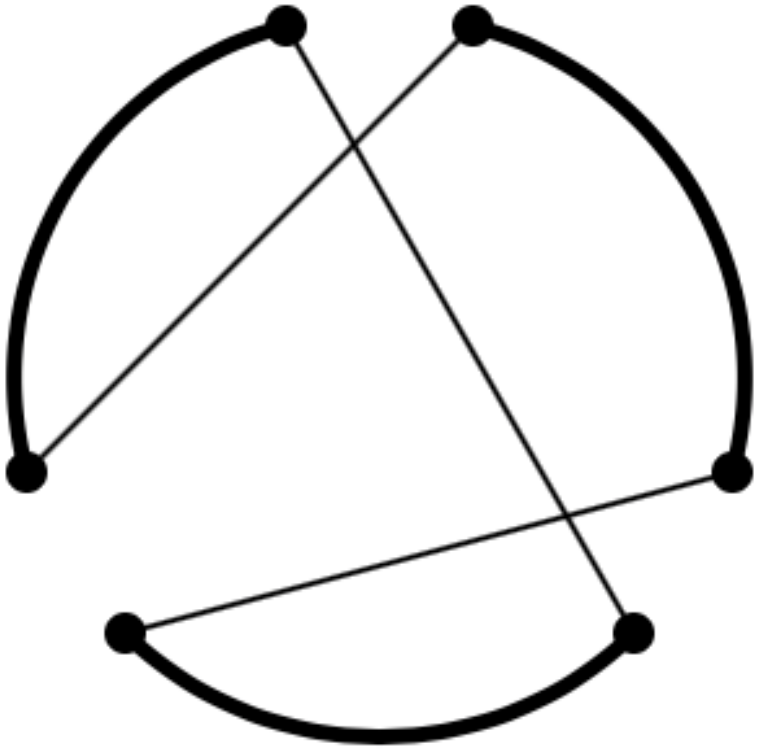


CASO 1

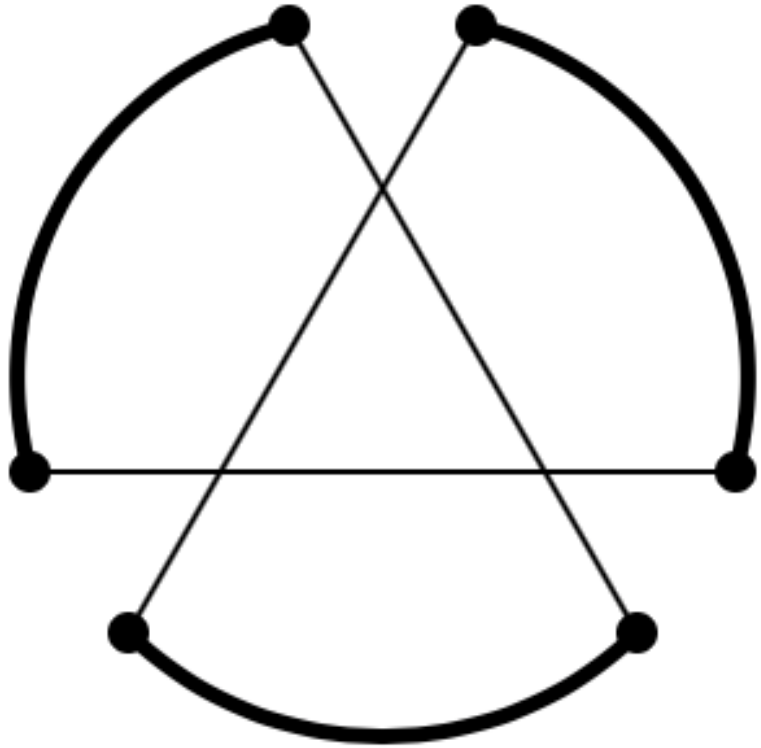


CASO 2

3-Opt



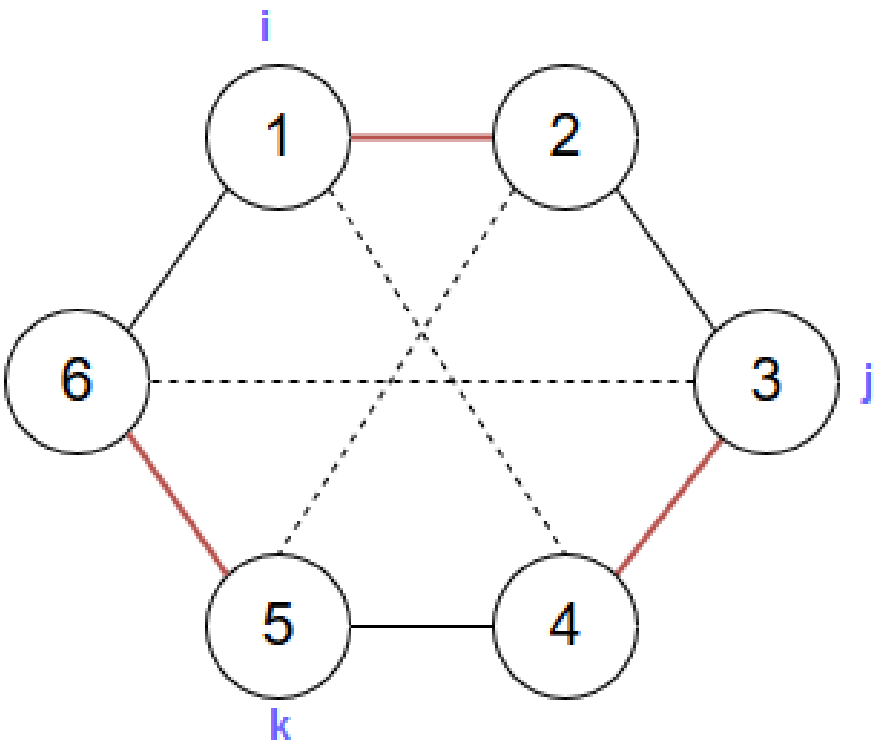
CASO 3

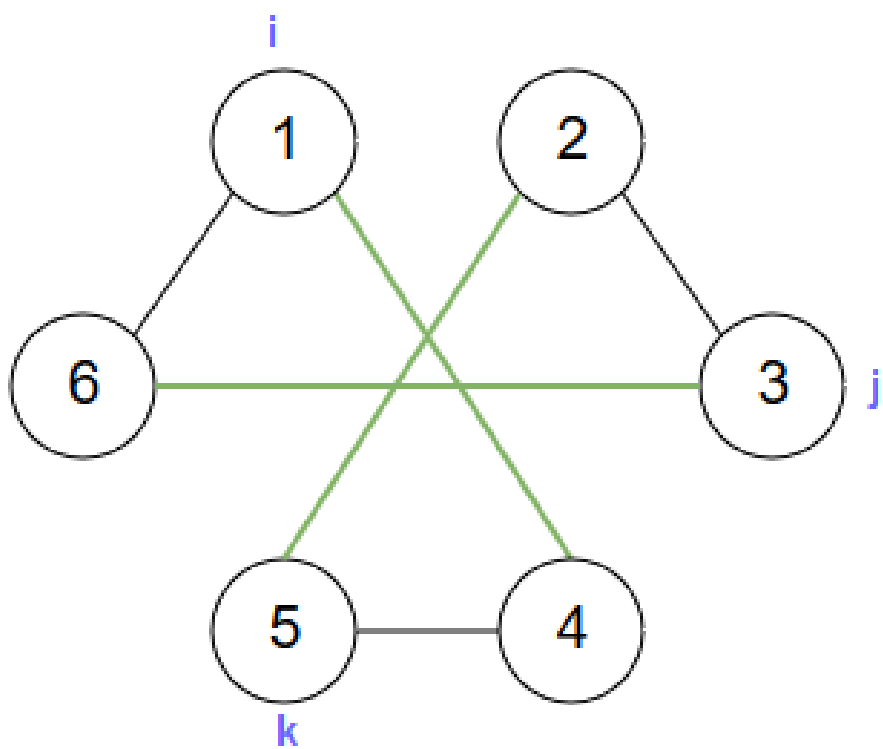


CASO 4

3-Opt esempio caso 4

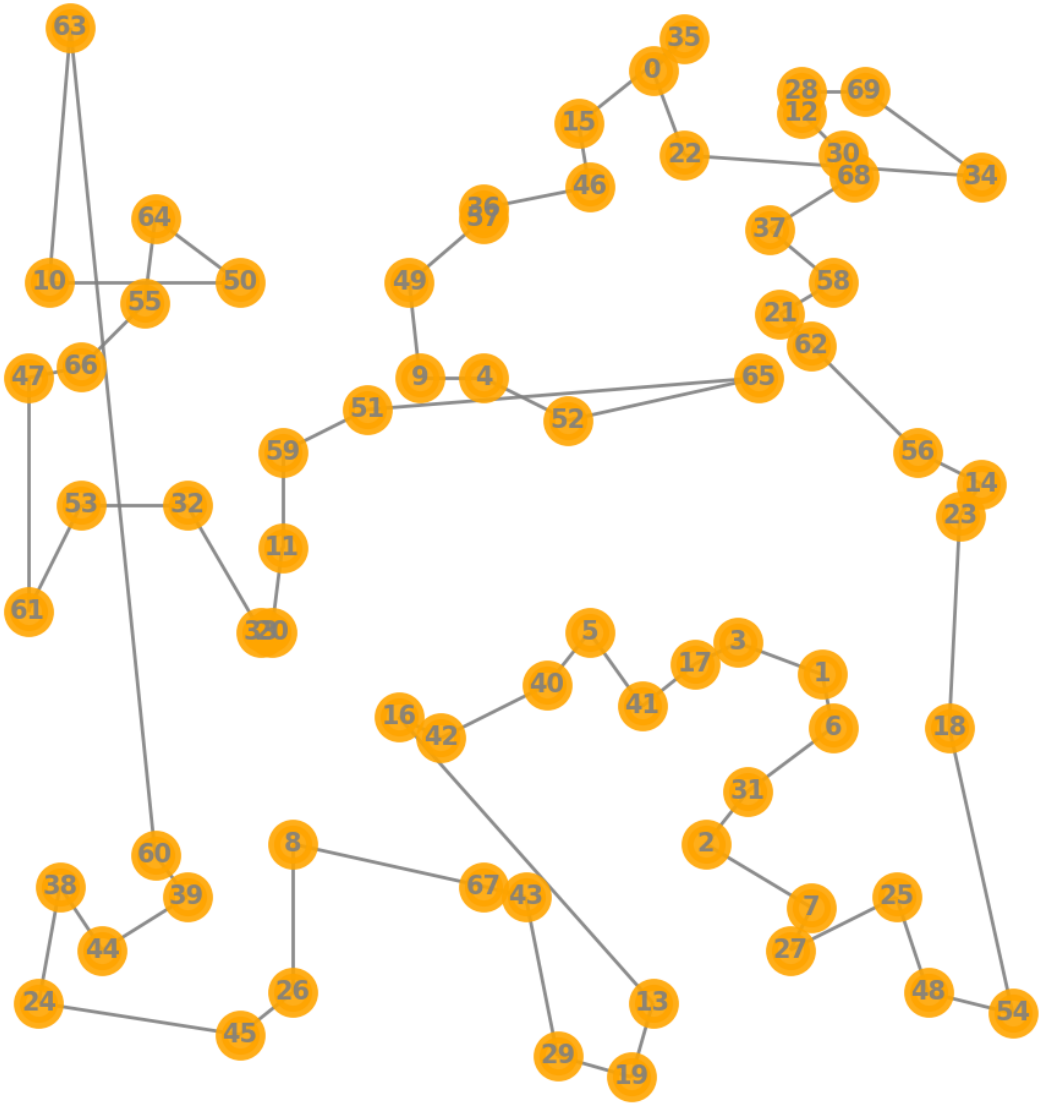
```
for i in range(0, len(route) - 5):
    for j in range(i + 2, len(route) - 3):
        for k in range(j + 2, len(route) - 1):
            new_route = route[:i + 1] + route[j + 1:k + 1] +
                        route[i + 1:j + 1] + route[k + 1:]
```



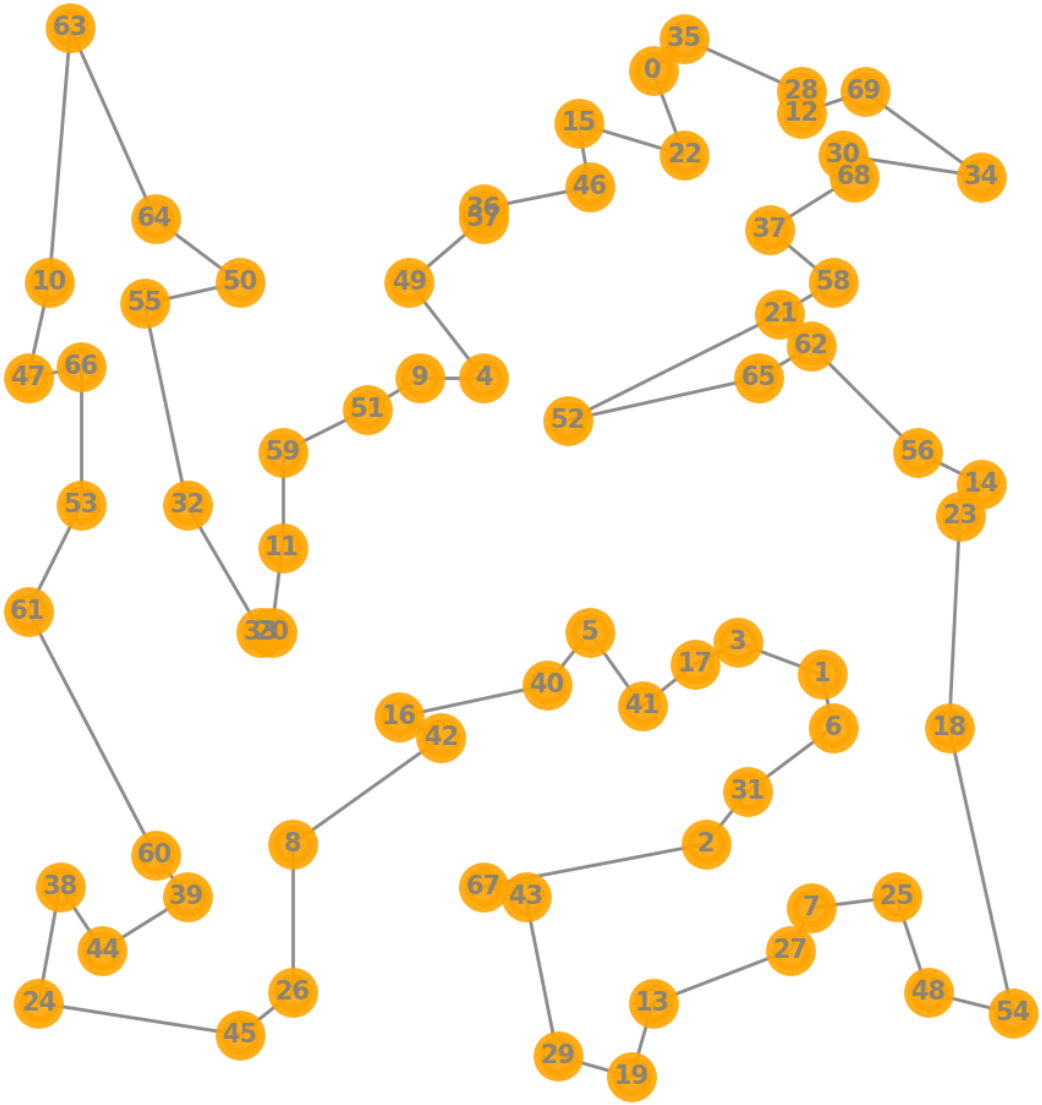


Archi considerati: (1,2) (3,4) (5,6) Percorso risultante: $[1, 4, 5, 2, 3, 6, 1]$

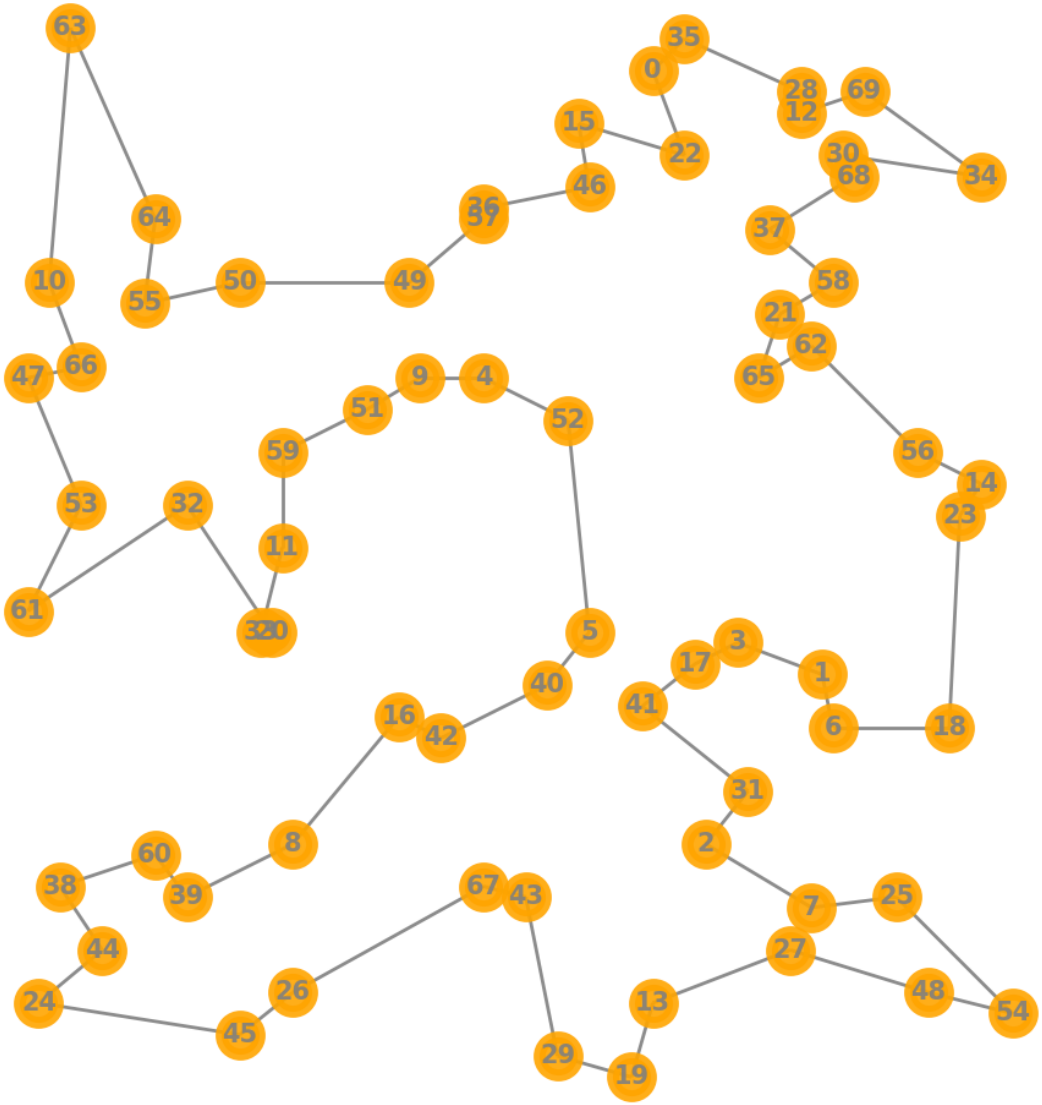
Esempio - st70 - soluzione iniziale NNv2



Esempio - st70 - soluzione finale



Esempio - st70 - soluzione ottima



Outline

Random Tour

L'analisi dei risultati è concentrata sullo studio di come la **soluzione iniziale possa influenzare il calcolo della soluzione finale**.

Inizialmente abbiamo voluto verificare l'adeguatezza o meno di usa **soluzione iniziale** generata in modo totalmente **casuale**.

I risultati sono i seguenti:

- Generalmente le prestazioni sono peggiori rispetto a quelle generate con nearest neighbor
- Nel **2%** delle esecuzioni viene raggiunto l'ottimo

Nearest Neighbor

Abbiamo eseguito diversi test partendo da NNv1 ed NNv2 giungendo ai seguenti risultati:

- NNv1 permette di ottenere più velocemente una soluzione iniziale ma richiede l'**esplorazione di più intorni**
- NNv2 ha un costo computazionale più alto ma permette di **ridurre il numero di intorni da esplorare**

Risultati

Problema	Approccio	Tempo	2-opt	3-opt	Soluzione
pr76	NNv1	67s	20	6	**111512**
pr76	NNv2	23s	14	2	113265
st70	NNv1	9s	14	1	738
st70	NNv2	24s	13	3	708
st70	Random	33s	61	4	**703**
berlin52	NNv1	6s	12	2	7799
berlin52	NNv2	3s	6	1	**7711**
kroA100	NNv1	114s	24	4	21708
kroA100	NNv2	31s	18	1	**21389**
eil101	NNv1	89s	23	3	**635**
eil101	NNv2	61s	22	2	638
eil101	Random	116s	98	4	668

GRAZIE!