



Università  
degli Studi  
di Ferrara

# TSP con pick up and delivery

Progetto del corso di Ricerca Operativa

---

Michele Vaccari - Matricola 121955

1 febbraio 2023

Università degli studi di Ferrara

Corso di laurea magistrale in Ingegneria Informatica e dell'Automazione

AA 2020-2021

## **Introduzione**

---

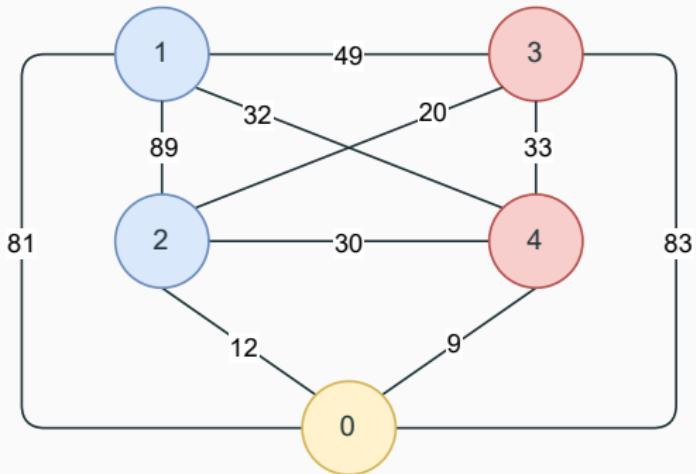
### Progettino 24 - TSP con pick up and delivery (1 persona)

A partire dalla base (nodo 0 del grafo) un corriere deve soddisfare  $n$  richieste di prelievo e consegna di documenti:

- ogni documento è prelevato in un nodo e consegnato in un altro nodo;
- ogni nodo è riferito a una singola richiesta, ma nel tragitto tra punto di prelievo e consegna si possono prelevare/consegnare altri documenti.

Noto il tempo di percorrenza dei singoli archi, si vuole minimizzare la durata del percorso, con partenza e rientro al deposito.

## Un esempio di istanza



**Figura 1:** Istanza con 2 richieste di prelievo e consegna

## **Modello matematico per S-TSPPD**

---

## Notazione i

- Numero di richieste di prelievo e consegna:  $n$
- Nodi di pickup:  $V_P = \{1, \dots, n\}$
- Nodi di delivery:  $V_D = \{n + 1, \dots, 2n\}$
- Una richiesta è identificata con la coppia di nodi  $(i, n + i)$  e  $i$  deve precedere  $n + i$  in un percorso ammissibile
- $V$  è definito come l'unione di  $V_P$  e  $V_D$  con l'aggiunta del nodo di deposito  $\{0\}$
- $E_{PD}$  è l'insieme degli archi che connettono  $V_P \cup V_D$
- $E$  è l'unione di  $E_{PD}$  con tutti gli archi ammissibili che collegano il nodo di deposito

## Notazione ii

- Il grafo  $G = (V, E)$  comprende tutti i nodi e gli archi necessari per descrivere il S-TSPPD (il grafo è completo)

$$V = \{0\} \cup V_P \cup V_D$$

$$E = \{(0, i) | i \in V_P\} \cup \{(0, i) | i \in V_D\} \cup E_{PD}$$

L'insieme degli archi  $E$  è definito in modo tale da includere solo archi ammissibili. In altre parole, non è possibile che un percorso TSPPD inizi con una consegna o finisca con un prelievo.

- Ogni arco  $(i, j) \in E$  è uguale all'arco  $(j, i)$
- Ogni arco ha gli stessi costi e variabili
- $c_{ij}$  è un costo non negativo per ogni arco  $(i, j) \in E$
- $x_{ij} \in \{0, 1\}$  è una variabile decisionale binaria per ogni  $(i, j) \in E$  con il valore  $x_{ij} = 1$  se l'arco  $(i, j)$  è in una soluzione e 0 altrimenti
- $\delta(S) = \{(i, j) \in E | i \in S, j \notin S\}$  è il taglio contenente gli archi che collegano  $S \subset V$  e  $S^* \subset V$
- Per qualsiasi nodo  $i \in V$ ,  $\delta(i) = \delta(\{i\})$

$$\min \sum_{(i,j) \in E} c_i x_{ij}$$

tale che

1.  $x_0 = 1$
2.  $x(\delta(i)) = 2 \quad \forall i \in V$
3.  $x(\delta(i)) \geq 2 \quad \forall S \subset V$
4.  $x(\delta(i)) \geq 4 \quad \forall S \subset V, \{0, n+i\} \subset S, \{0, i\} \subset V \setminus S$
5.  $x_{ij} \in \{0, 1\} \quad (i, j) \in E$

Il vincolo 1 richiede che l'arco che collega il nodo di deposito faccia parte di qualsiasi soluzione ammissibile.

Il vincolo 2 richiede che ogni nodo entri ed esca in tutti i percorsi ammissibili, ma di per sé lascia aperta la possibilità di sotto-tour, formando una rappresentazione completa del TSP.

Il vincolo 3 consente di eliminare i sotto-tour, formando una rappresentazione completa del TSP.

Il vincolo 4 richiede che i ritiri avvengano prima delle rispettive consegne

## **Modello matematico per A-TSPPD**

---

## Notazione i

- Numero di richieste di prelievo e consegna:  $n$
- Nodi di pickup:  $N_P = \{1, \dots, n\}$
- Nodi di delivery:  $N_D = \{n + 1, \dots, 2n\}$
- Una richiesta è identificata con la coppia di nodi  $(i, n + i)$  e  $i$  deve precedere  $n + i$  in un percorso ammissibile
- $N$  è definito come l'unione di  $N_P$  e  $N_D$  con l'aggiunta del nodo di deposito  $\{0\}$
- $A_{PD}$  è l'insieme degli archi *orientati* che connettono  $N_P \cup N_D$
- $A$  è l'unione di  $A_{PD}$  con tutti gli archi *orientati* ammissibili che collegano il nodo di deposito

## Notazione ii

- Il grafo  $G = (N, A)$  comprende tutti i nodi e gli archi necessari per descrivere il A-TSPPD (il grafo è completo)

$$N = \{0\} \cup N_P \cup N_D$$

$$A = \{(0, i) | i \in N_P\}$$

$$\cup \{(i, 0) | i \in N_D\}$$

$$\cup \{(i, j) | i \in N_P, j \in (N_P \cup N_D) \setminus \{i\}\}$$

$$\cup \{(n + i, j) | i \in N_P, j \in (N_P \cup N_D) \setminus \{i, n + i\}\}$$

L'insieme degli archi  $A$  è definito in modo tale da includere solo archi ammissibili. In altre parole, non è possibile che un percorso TSPPD inizi con una consegna o finisca con un prelievo.

- $c_{ij}$  è un costo non negativo per ogni arco  $(i, j) \in A$
- $x_{ij} \in \{0, 1\}$  è una variabile decisionale binaria per ogni  $(i, j) \in A$  con il valore  $x_{ij} = 1$  se l'arco  $(i, j)$  è in una soluzione e 0 altrimenti

- $\delta(S) = \{(i,j) \in A | i \in S, j \notin S\}$  è il taglio contenente gli archi che collegano  $S \subset N$  e  $S^* \subset N$
- Per qualsiasi nodo  $i \in N$ ,  $\delta(i) = \delta(\{i\})$

$$\min \sum_{(i,j) \in A} c_i x_{ij}$$

tale che

1.  $\sum_{(i,j) \in A} x_{ij} = 1 \quad \forall i \in N$
2.  $\sum_{(i,j) \in A} x_{ij} = 1 \quad \forall j \in N$
3.  $x(\delta(S)) \geq 1 \quad \forall S \subset N$
4.  $x(\delta(i)) \geq 4 \quad \forall S \subset N, \{0, n+i\} \subset S, \{0, i\} \subset N \setminus S$
5.  $x_{ij} \in \{0, 1\} \quad (i, j) \in A$

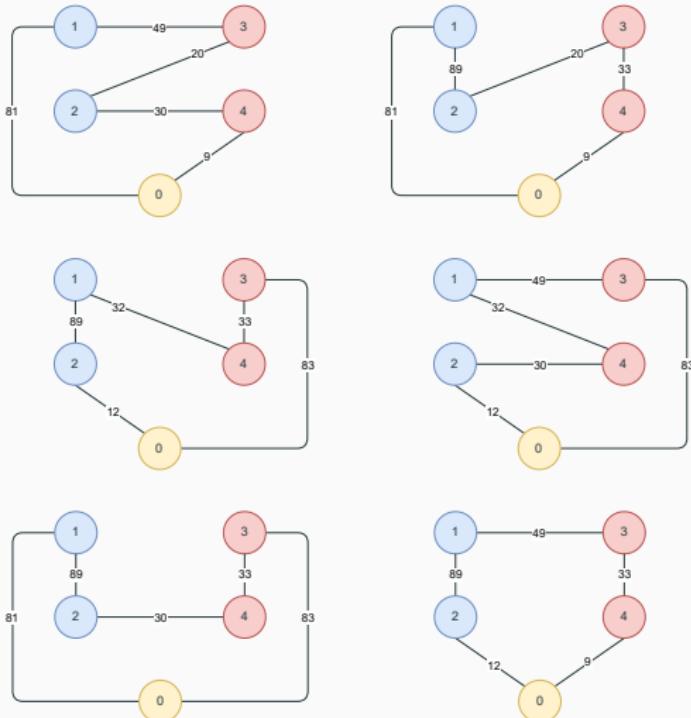
I vincoli 1 e 2 richiedono che ogni nodo preceda e segua direttamente un altro nodo.

Il vincolo 3 consente di eliminare i sotto-tour, formando una rappresentazione completa del TSP.

Il vincolo 4 richiede che i ritiri avvengano prima delle rispettive consegne (è uguale al vincolo 4 del S-TSPPD)

- Di fatto il TSPPD è un TSP con in più i vincoli di precedenza.
- Sia S-TSPPD che A-TSPPD sono formulazioni che pongono dei problemi per il gran numero di vincoli
- Un'istanza S-TSP con  $n$  coppie ha  $\# TSP = \frac{1}{2}(2n - 1)!$  soluzioni distinte [1], mentre un TSPPD della stessa dimensione ha  $\# TSPPD(n) = \frac{2n!}{2^n}$  [3]
- La dimensione dell'insieme ammissibile del TSPPD cresce più lentamente rispetto al numero di coppie di nodi rispetto a quella del TSP
- Ovviamente, l'insieme delle soluzioni ammissibili del TSPPD è un sottoinsieme delle soluzioni fattibili del TSP
- Se si ammettono solo coppie di nodi di ritiro e consegna nel percorso con relazioni di precedenza, la dimensione dell'insieme fattibile del TSP si riduce di  $\frac{1}{2^{n-1}!}$  [3]

## Soluzioni ammissibili per l'istanza d'esempio



**Figura 2:** Soluzioni ammissibili per l'istanza con 2 richieste di prelievo e consegna

## Progetto

---

## Strumenti utilizzati

- Linguaggio Python 3.10 con le seguenti librerie:
  - Click
  - Numpy
  - Pytest
  - Openpyxl
  - Stopwatch
- Visual Studio Code come editor di testo
- Git per il versionamento del codice

### Architettura

L'applicazione è formata da 3 componenti:

- **problem:** è il package che contiene le entità utilizzate per modellare il problema e un generatore di istanze casuali
- **solver:** è il package che contiene gli algoritmi implementati. Utilizza le entità definite nel package problem
- **cli:** è una Command Line Interface che consente di generare istanze del problema e testare i vari algoritmi

### Repository GitHub

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery>

## Automazione

L'esecuzione e la raccolta dei dati durante i benchmark è stata automatizzata sempre utilizzando il linguaggio Python. Si è utilizzata la libreria Openpyxl in modo tale che i dati vengano collezionati in modo ordinato su fogli di calcolo.

## Esecuzione

I benchmark sono stati eseguiti su un PC con:

- **Sistema operativo:** Windows 10
- **Processore:** Intel Core i5-4570 CPU 3.20 GHz
- **RAM:** 16.0 GB

Il tempo di esecuzione complessivo dei benchmark è stato di circa 10 giorni.

Nel dettaglio:

346399.9 *secondi* → 5773.332 *minuti* → 240.5555 *ore* → 10.02315 *giorni*

## Generatore di istanze

È possibile generare un'istanza del problema nel seguente modo:

- **Istanza asimmetrica**

```
python tsppdcli.py generate-instance --requests 2  
--weights-random --output-instance-path  
2-request-weights-random.json
```

- **Istanza simmetrica**

```
python tsppdcli.py generate-instance --requests 2  
--weights-as-euclidean-distance --output-instance-path  
2-request-weights-euclidean-distance.json
```

### Esempio di istanza

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/instances/2-request.json>

## **Algoritmi esatti**

---

## Descrizione dell'algoritmo

Vediamo il codice dell'algoritmo Brute force enumerator:

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/bruteForceEnumerator.py>

## Esempio

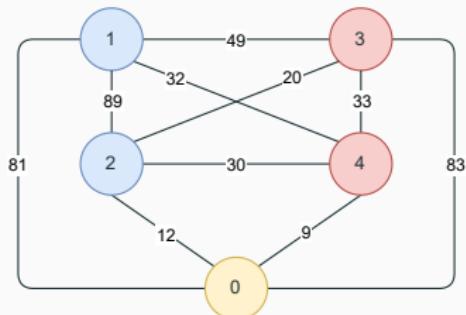


Figura 3: Istanza iniziale

## Esecuzione

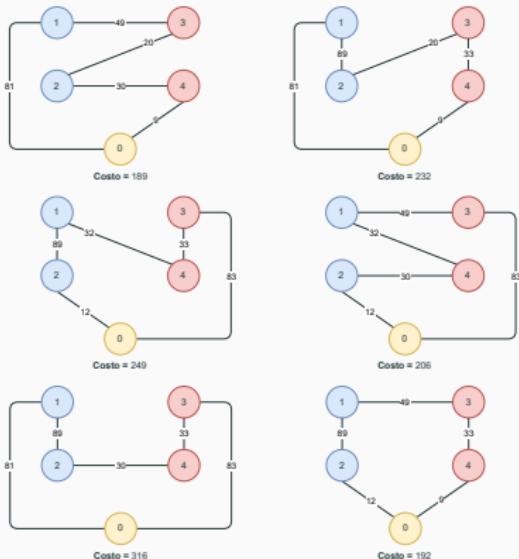
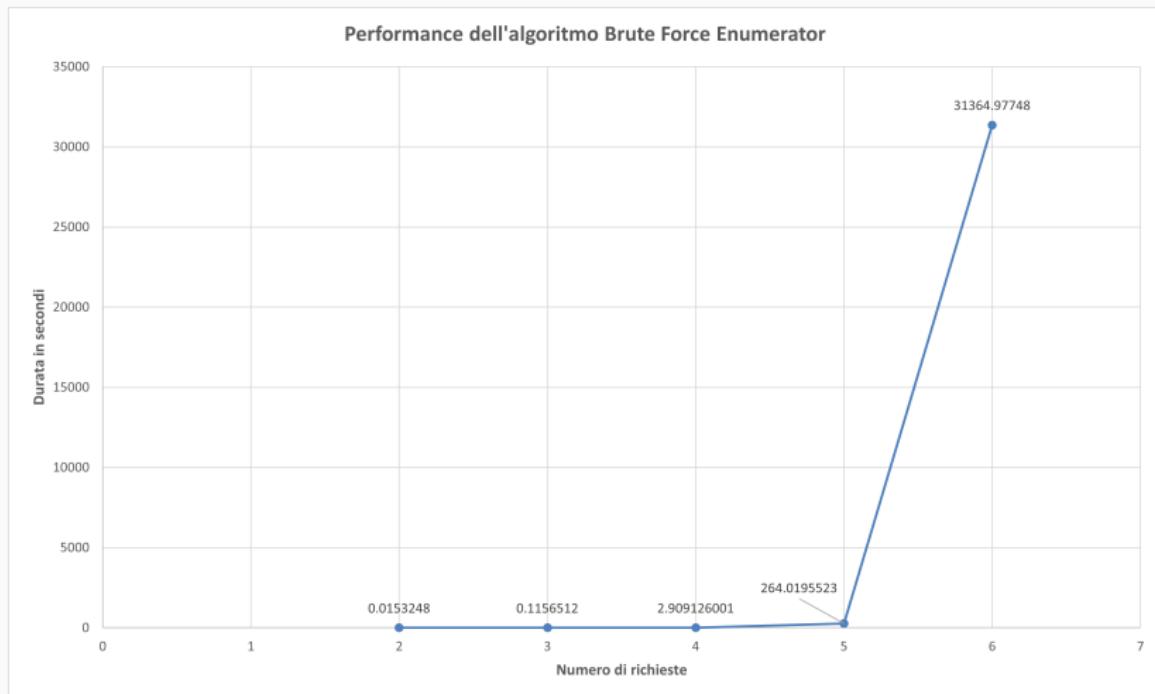


Figura 4: Soluzioni esplorate

## Performance nel tempo



```
function Initialize()
    for all  $i \in nodes$  do
         $arcs(i) \leftarrow ordered\{j | j \in nodes \setminus \{0\}, j \neq i\}$ 
    tour  $\leftarrow (0)$ 
    best  $\leftarrow \emptyset$ 

function Enumerate()
     $n_1 \leftarrow last\ node\ in\ tour$ 
    for all  $n_2 \in arcs(current)$  do
        if  $n_2 \in tour$ 
            continue
        else if  $cost(tour) + cost(n_1, n_2) \geq cost(best)$ 
            continue
        else if  $n_2 \in N_D$  and  $pickup(n_2) \notin tour$ 
            continue

    tour  $\leftarrow tour + next$ 
    if  $|tour| \geq n - 1$  then
         $c \leftarrow cost(tour) + cost(n_2, 0)$ 
        if  $best = \emptyset$  or  $c < cost(best)$ 
            best  $\leftarrow tour + 0$ 
    else
        Enumerate()

    tour  $\leftarrow tour - next$ 

Initialize()
Enumerate()
```

## Descrizione dell'algoritmo

L'algoritmo O'Neil – Hoffman [2] utilizza una funzione ricorsiva per cercare l'insieme ammissibile dei percorsi TSPPD, tenendo traccia della migliore soluzione scoperta in ogni punto dell'albero di ricerca.

L'algoritmo mantiene un nodo corrente e un percorso parziale che inizia con 0 (nodo deposito).

Ogni ricorsione ripercorre gli archi a partire dal nodo corrente e aggiunge ogni arco ammissibile al tour individualmente prima della ricorsione. Se il costo aggiunto da un arco fa sì che il costo di un percorso parziale sia superiore a quello del tour migliore conosciuto, quell'arco viene ignorato e la sua sezione dell'albero di ricerca viene effettivamente esplorata. Se un percorso completo migliora la soluzione migliore conosciuta, viene memorizzato come nuovo ottimo candidato.

La chiave della ricerca efficace e della scoperta precoce di buone soluzioni è la tecnica di ordinamento degli archi. Durante l'inizializzazione, ad ogni nodo viene assegnato un vettore ordinato di nodi successivi. Come funzione di ordinamento utilizziamo il costo ascendente dell'arco, ma altri possono essere facilmente incorporati.

I nodi che non sono fattibili per il percorso parziale corrente vengono saltati.

## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/oneilHoffmanEnumerator.py>

## Esempio

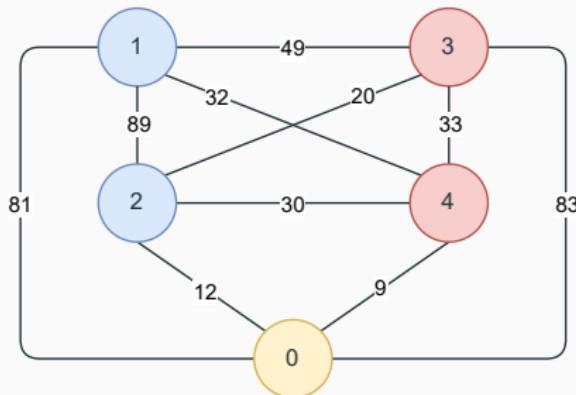
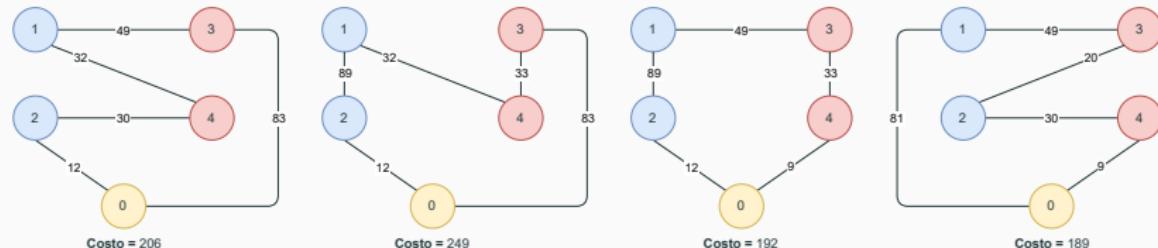


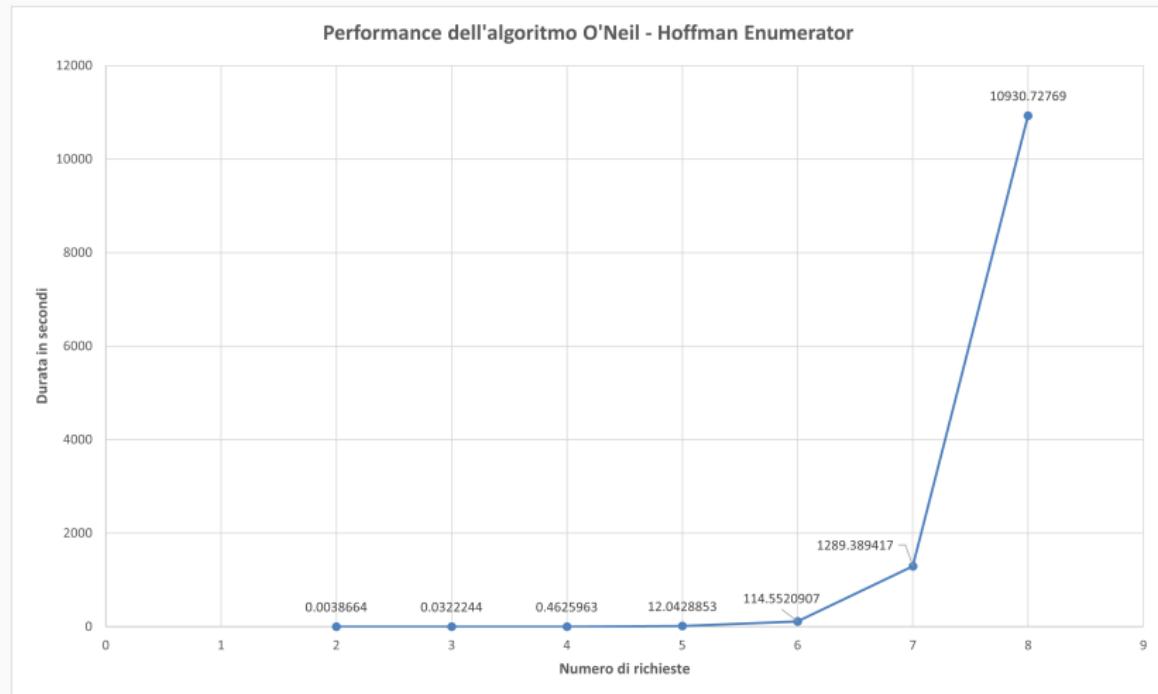
Figura 5: Istanza iniziale

## Esecuzione

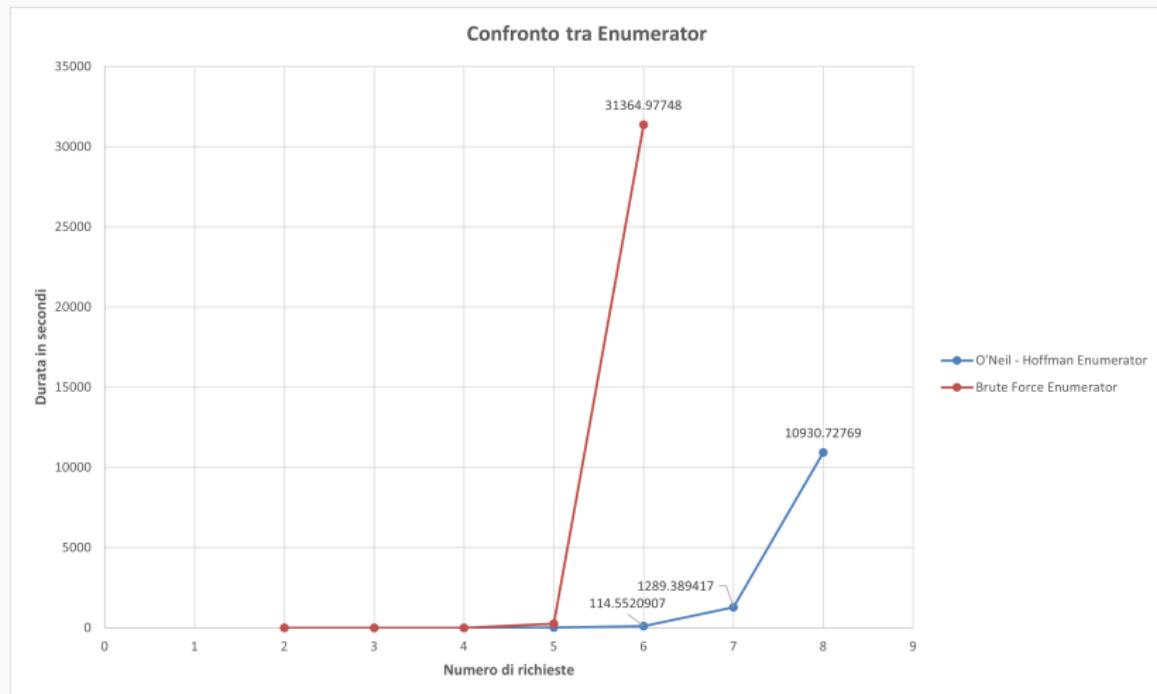


**Figura 6:** Soluzioni esplorate

## Performance nel tempo



## Performance nel tempo



## Confronto ii

### Soluzioni ammissibili esplorate

I due algoritmi differiscono, oltre che per il tempo di esecuzione anche per il numero di soluzioni ammissibili esplorate

Numero di richieste	Brute Force Enumerator		O'Neil - Hoffman Enumerator	
	Durata (in secondi)	Soluzioni ammissibili esplorate	Durata (in secondi)	Soluzioni ammissibili esplorate
2	0.0153248	6	0.0038664	3
3	0.1156512	90	0.0322244	10
4	2.909126001	2520	0.4625963	3
5	264.0195523	113400	12.0428853	23
6	31364.97748	7484400	114.5520907	115
7	N.A.	N.A.	1289.389417	191
8	N.A.	N.A.	10930.72769	24

**Tabella 1:** Numero di soluzioni ammissibili esplorate dagli enumeratori

L'enumerazione può essere una tecnica valida per risolvere piccoli TSPPD. Se guidata da alcune informazioni sul problema, l'enumerazione può essere in grado di scoprire rapidamente percorsi validi senza l'onere della formulazione del modello.

## **Algoritmi euristici costruttivi**

---

Gli algoritmi greedy (voraci) determinano la soluzione attraverso una sequenza di decisioni parziali (localmente ottime), *senza mai modificare le decisioni già prese.*

Sono di facile implementazione e notevole efficienza computazionale, ma, sia pure con alcune eccezioni di notevole rilievo (Matroidi), *non garantiscono l'ottimalità, e a volte neppure l'ammissibilità della soluzione prodotta.*

### Definizione di algoritmo greedy

Sia dato il problema di ottimo associato al sistema di insiemi indipendenti  $(E, F)$  con funzione obiettivo  $c$ , si dice Greedy un algoritmo  $A$  costruttivo iterativo che costruisce l'insieme  $X \in F$ , soluzione ammissibile del problema, partendo dall'insieme vuoto ed inserendovi ad ogni passo l'elemento di  $E$  più "promettente" fra quelli che non violano l'indipendenza dell'insieme.

### Procedure Greedy (template)

- Soluzione iniziale  $S_0 = \emptyset$
- ad ogni step  $k$ 
  - seleziona  $e_k$  come l'elemento di  $E$  più promettente (*criterio best*) tra quelli non ancora esaminati,
  - valuta se la soluzione parziale  $S_k \cup e_k \in F$  (*test di indipendenza* o di ammissibilità delle soluzioni parziali)
- termina avendo esaminato tutto  $E$  o quando  $S_k$  è massimale ( $S_n := S_k$ ).

Per costruzione  $X = S_n$  è massimale. La complessità di "Greedy" dipende dalla complessità delle procedure *Best* e *Ind*, essendo predefinito il numero massimo di iterazioni ( $n$ ). Per costruire un algoritmo greedy è sufficiente definire  $E$  ed  $F$ , e specificare *Best* e *Ind* nell'ambito del template.

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyTemplate.py>

### Definizione del criterio best e test di ammissibilità

**criterio best** seleziono prima i nodi pick-up in ordine di richiesta, poi i nodi di delivery in ordine di richiesta

**test di ammissibilità** verifico che le precedenze tra i nodi siano sempre rispettate (lo sono per costruzione)

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyPickupFirst.py>

### Esempio

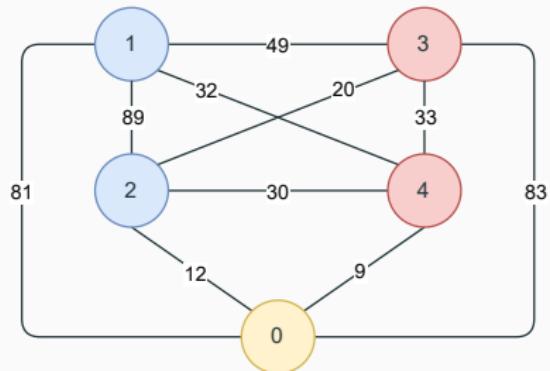


Figura 7: Istanza iniziale

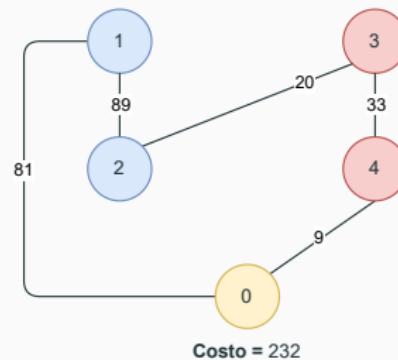
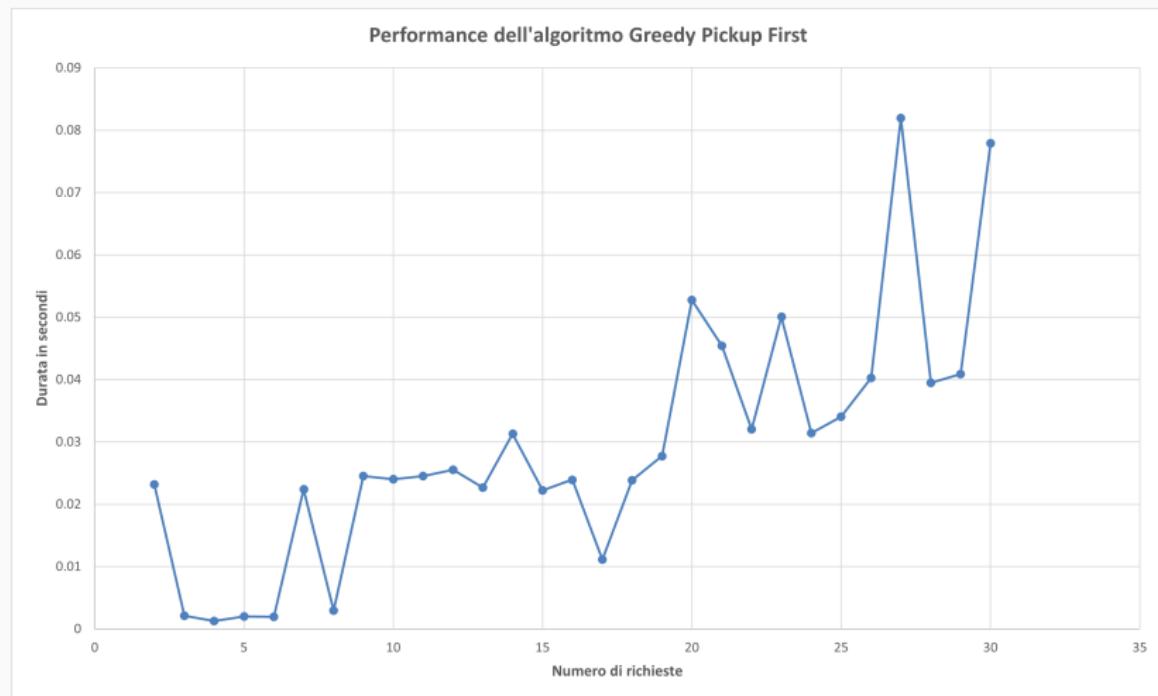


Figura 8: Soluzione generata

## Performance nel tempo



### Definizione del criterio best e test di ammissibilità

**criterio best** seleziono prima i nodi pick-up e di delivery  
in ordine di richiesta

**test di ammissibilità** verifico che le precedenze tra i nodi siano  
sempre rispettate (lo sono per costruzione)

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyRequestOrder.py>

## Greedy Request Order ii

### Esempio

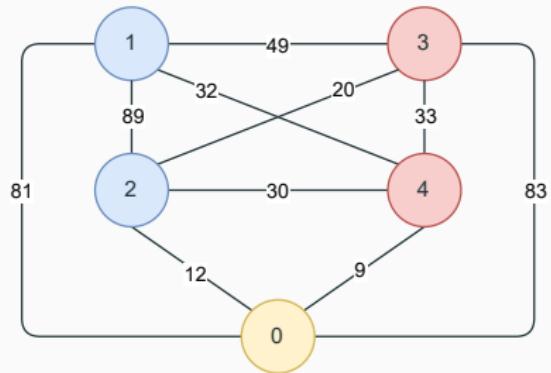


Figura 9: Istanza iniziale

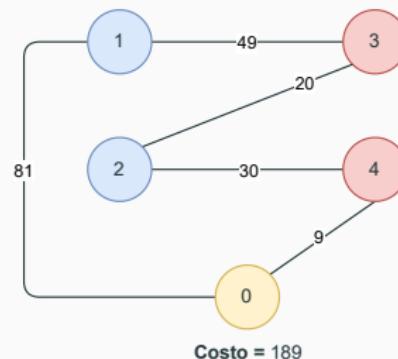
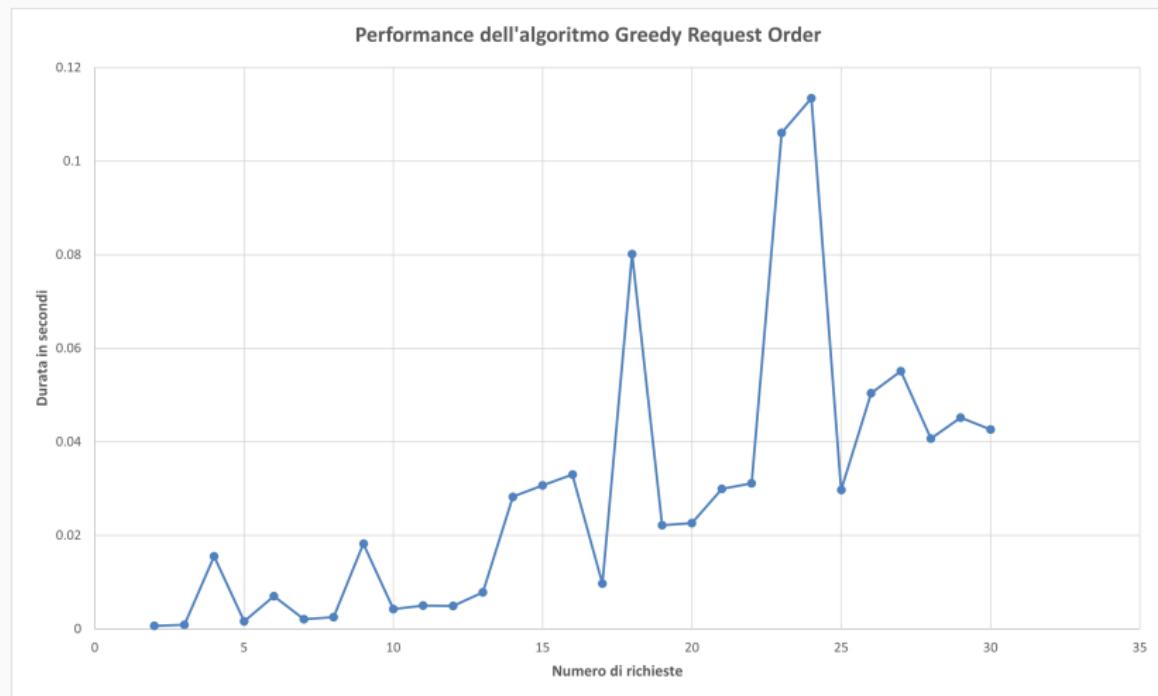


Figura 10: Soluzione generata

**N.B.** In questo caso la soluzione generata è la soluzione ottima

## Performance nel tempo



### Definizione del criterio best e test di ammissibilità

**criterio best** seleziono il nodo successivo più vicino  
rispetto al nodo corrente

**test di ammissibilità** verifico che le precedenze tra i nodi siano  
sempre rispettate

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyNearestNeighbor.py>

## Greedy Nearest Neighbor ii

### Esempio

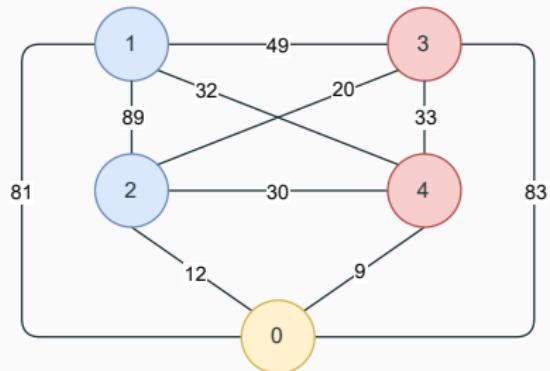


Figura 11: Istanza iniziale

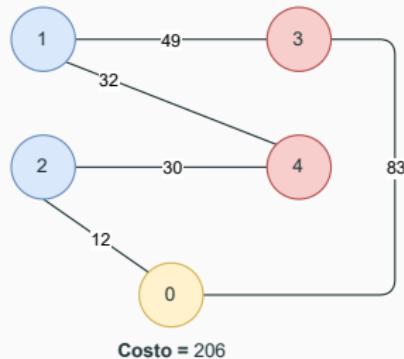
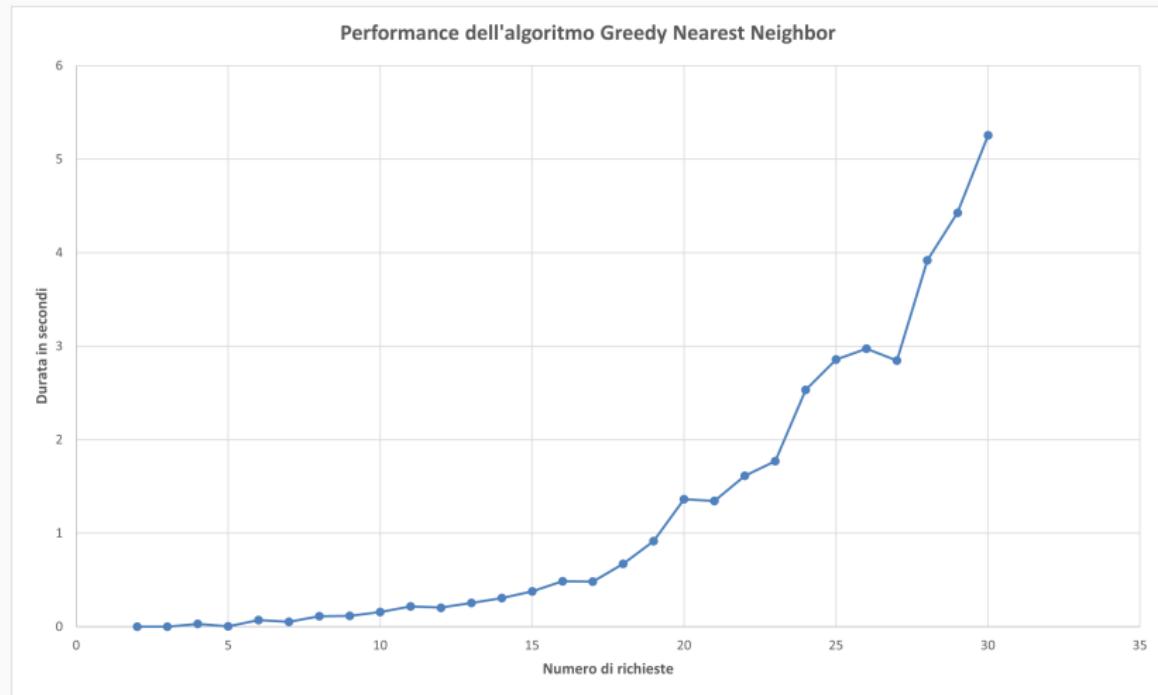


Figura 12: Soluzione generata

## Performance nel tempo



### Definizione del criterio best e test di ammissibilità

**criterio best** seleziono il nodo successivo a caso

**test di ammissibilità** verifico che le precedenze tra i nodi siano  
sempre rispettate

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyRandom.py>

### Esempio

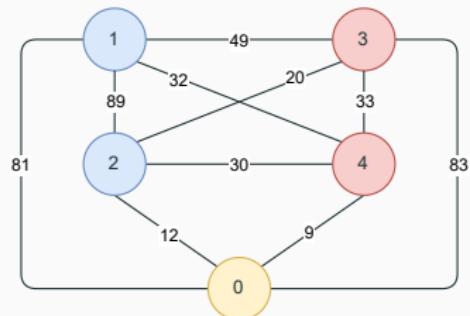


Figura 13: Istanza iniziale

## Esecuzione

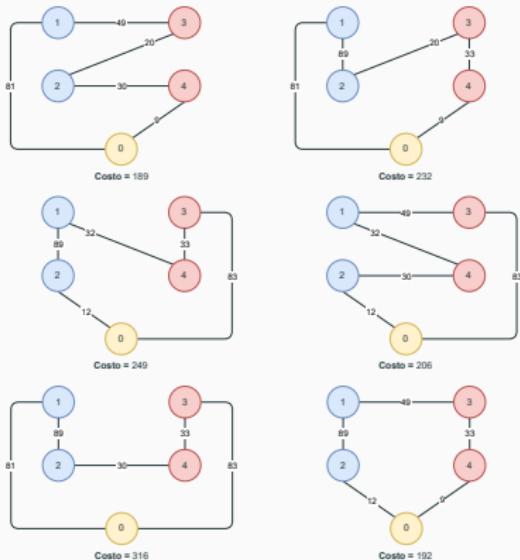
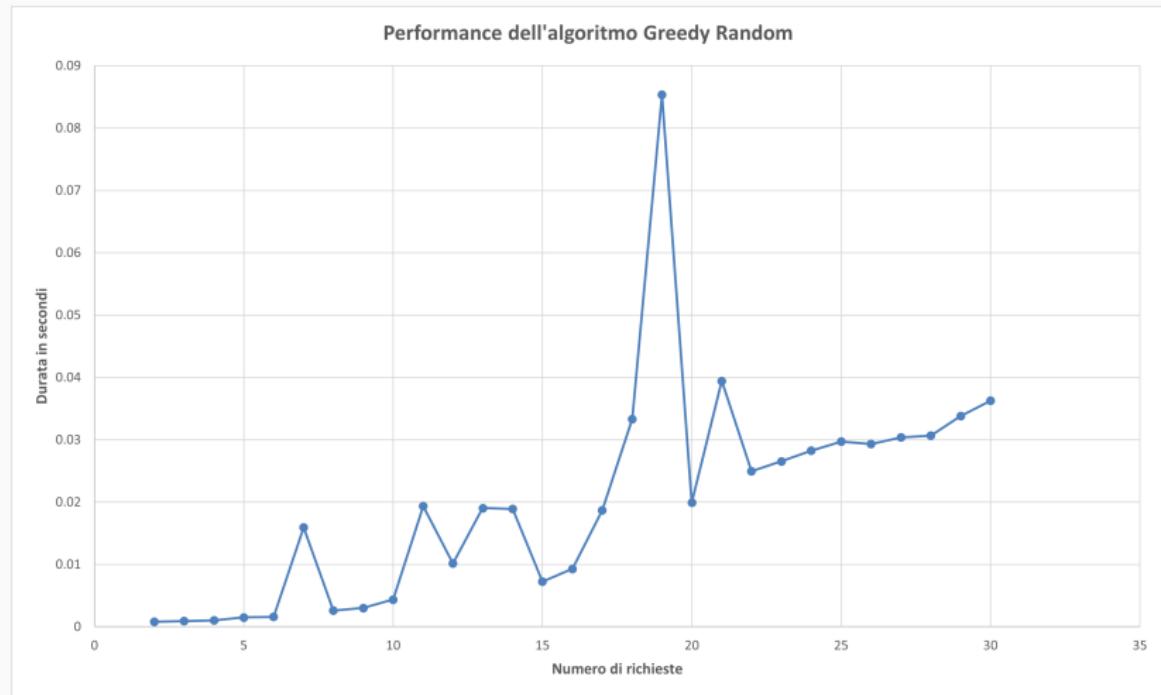


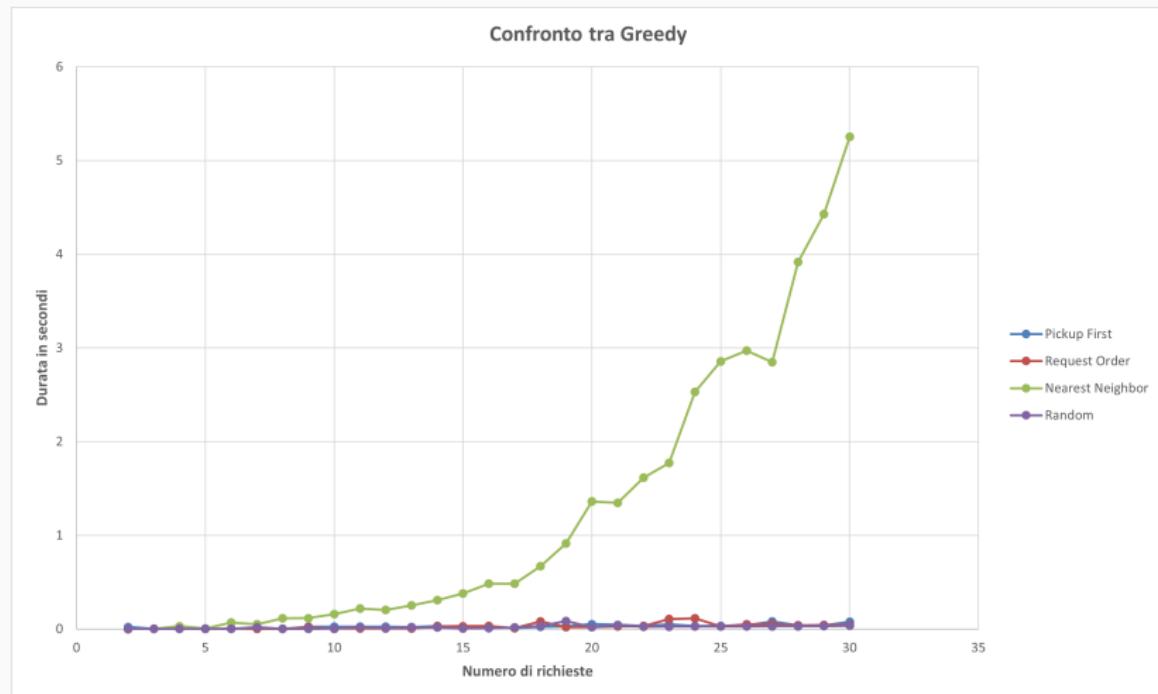
Figura 14: Possibili soluzioni generate

## Performance nel tempo



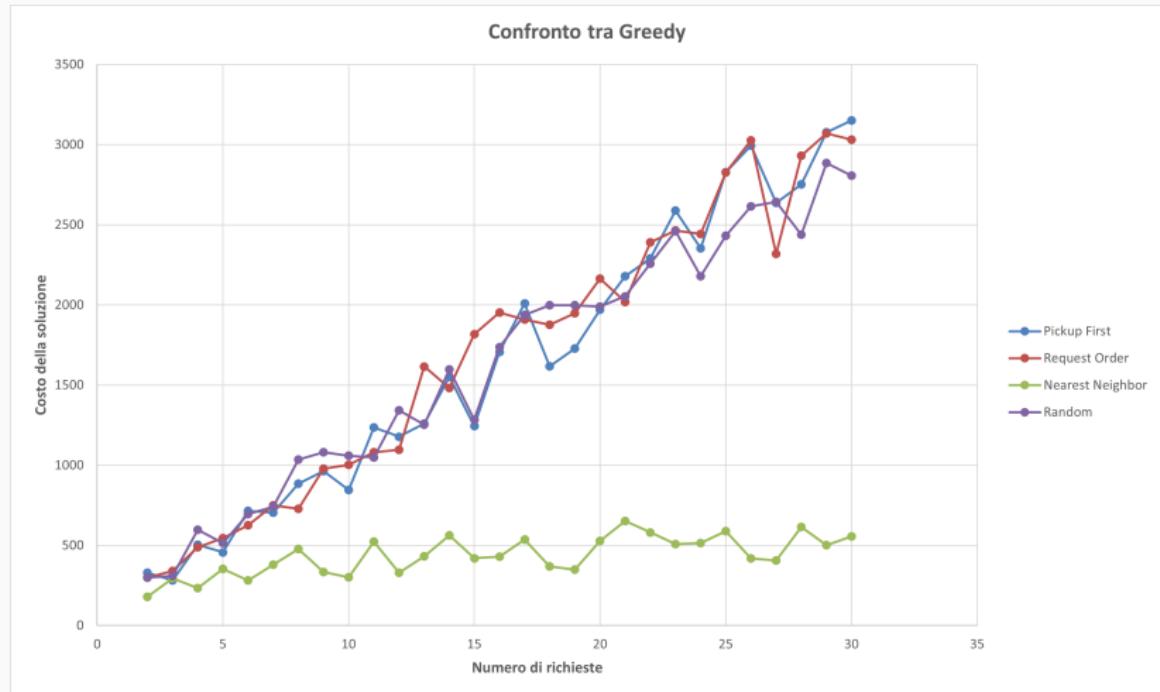
# Confronto i

## Performance nel tempo



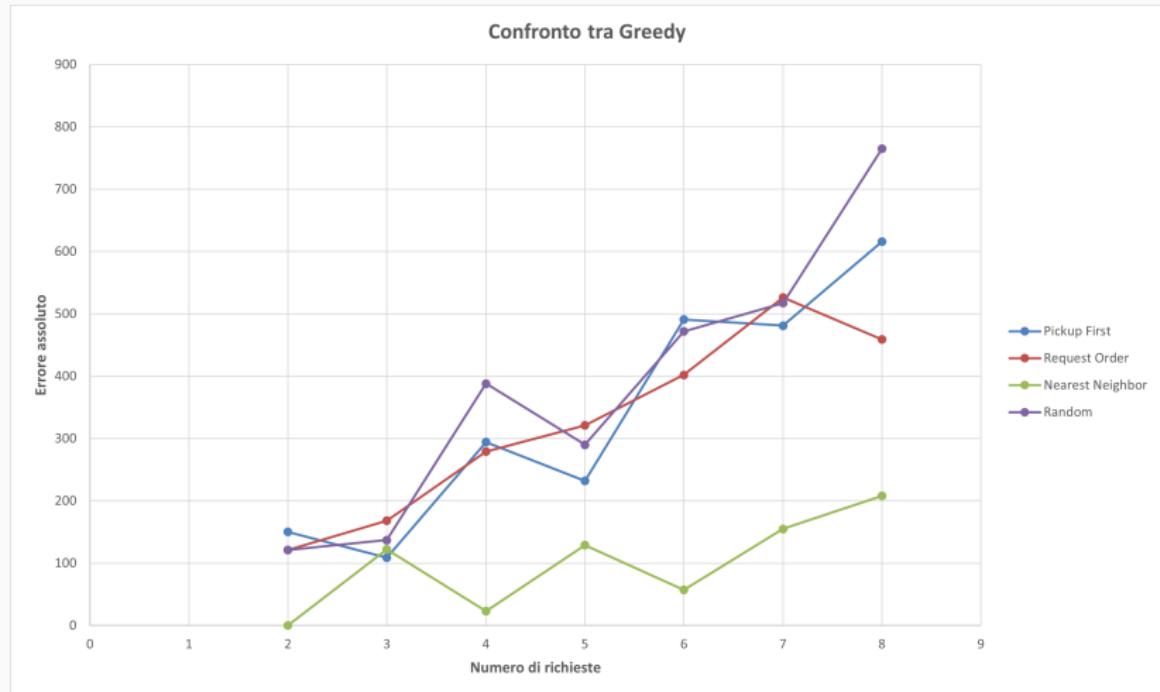
# Confronto ii

## Performance costo soluzione



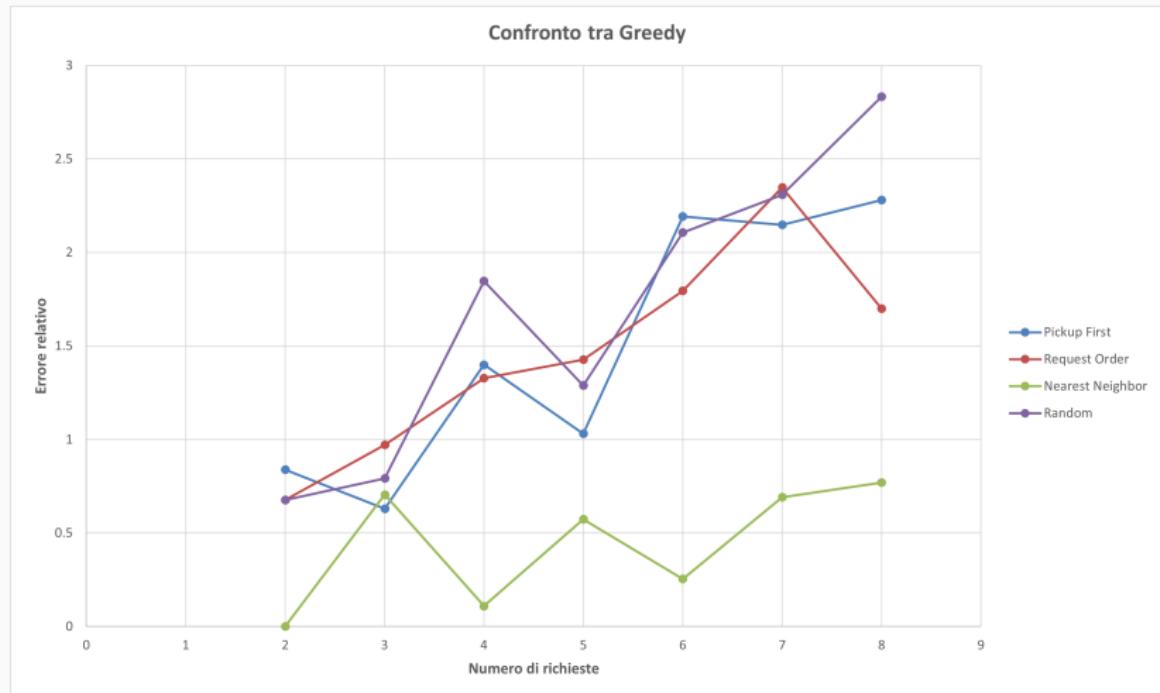
## Confronto iii

### Performance errore assoluto



## Confronto iv

### Performance errore relativo



## **Algoritmi euristici di miglioramento**

**Local search neighborhood based**

---

- La ricerca locale è l'algoritmo di riferimento per tutte le (meta)euristiche basate sul concetto di intorno.
- Data una soluzione  $x$ , l'intorno  $N(x^k)$  è dato attraverso la MOSSA che descrive operativamente cosa modifico di  $x$  per generare tutte le soluzioni in  $N(x^k)$ .
- Se l'intorno gode della proprietà di raggiungibilità, cioè per ogni coppia di soluzioni esiste una successione di soluzioni che le collega, tale per cui ogni soluzione appartiene all'intorno della soluzione precedente, allora la ricerca è in grado di esplorare tutta la regione ammissibile  $F$ .
- Se esplorando l'intorno non trovo una soluzione migliore, allora vuol dire che sono in un ottimo locale per quell'intorno.

- Esistono diverse tecniche secondo cui esplorare l'intorno. Per tutte le euristiche implementate si è scelto l'approccio *best improvement*, ovvero si risolve in modo esatto il sottoproblema trovando la soluzione migliore nell'intorno.
- Come soluzione di partenza viene utilizzata la migliore delle greedy (la soluzione che ha costo più basso).
- Consultando i file excel generati durante i benchmark si può valutare anche l'approccio *first improvement* (si restituisce la prima soluzione che migliora l'ottimo corrente) e *h-improvement* (si restituisce la migliore delle h soluzioni che migliorano l'ottimo corrente) senza dover rieseguire i benchmark, in quanto si è memorizzato l'andamento delle migliori soluzioni durante l'esplorazione dell'intorno.

# City swap i

## Definizione dell'algoritmo

*City swap* è l'intorno ottenuto dallo scambio di 2 nodi (anzichè 2 archi) nel ciclo.

## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/citySwap.py>

## Esempio

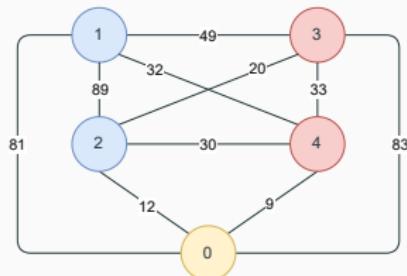


Figura 15: Istanza iniziale

# City swap ii

## Esecuzione

La soluzione di partenza è la migliore delle greedy  $\rightarrow s_0 = [0, 1, 3, 2, 4] \quad c(s_0) = 189$

## Passi dell'algoritmo:

1. Scambio il 3 con il 2  $\rightarrow s_1 = [0, 1, 2, 3, 4, 0] \quad c(s_1) = 232$
2. Scambio il 2 con il 3  $\rightarrow s_2 = [0, 1, 2, 3, 4, 0] \quad c(s_2) = 232$

N.B. La tabu search mi impedirebbe il punto 2

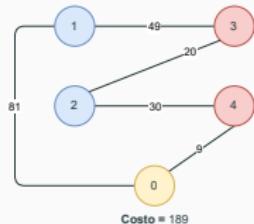


Figura 16: Soluzione  $s_0$

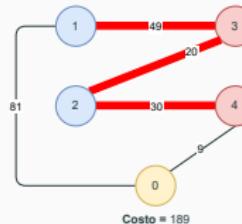


Figura 17: Archi eliminati al passo 2

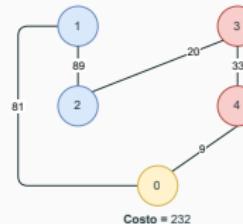
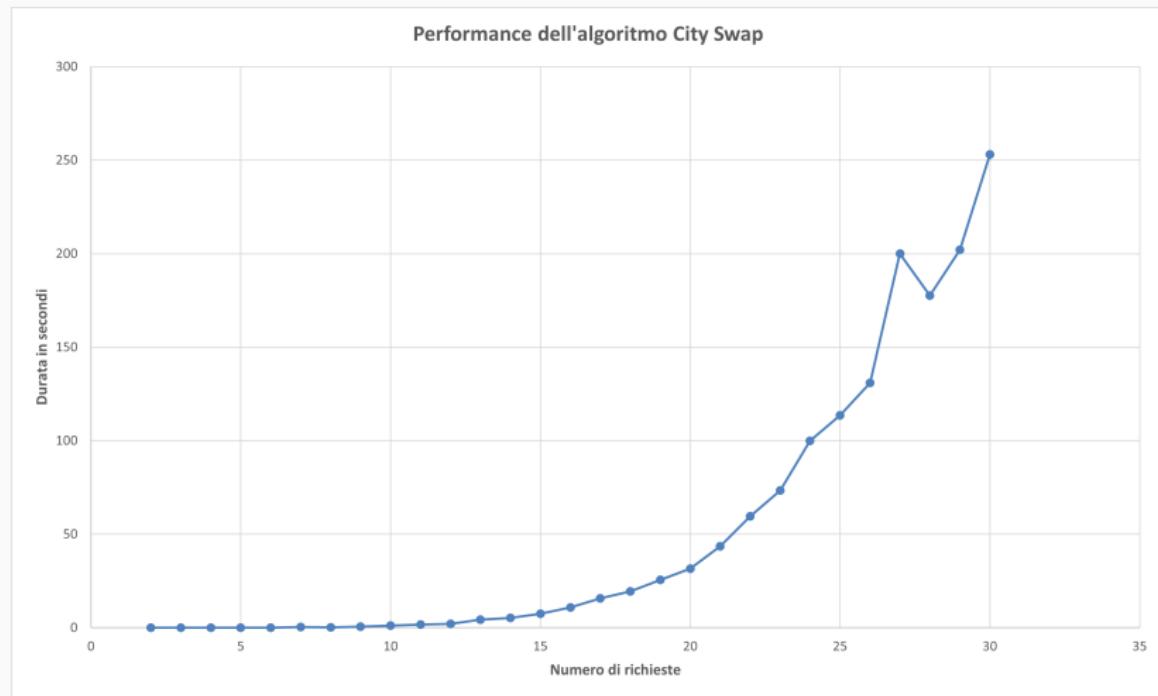


Figura 18: Soluzione  $s_1$

Il passo 2 è un 3 opt limitato agli archi incidenti su questi nodi

## Performance nel tempo



## Definizione dell'algoritmo

*City insert* è l'intorno ottenuto dalla cancellazione di 1 nodo nel ciclo e dal suo reinserimento in un altro punto.

## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/citySwap.py>

## Esempio

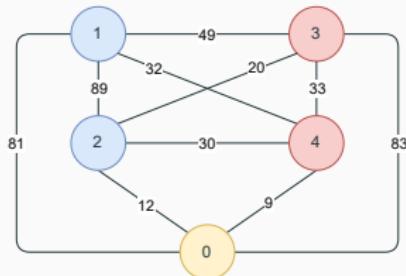


Figura 19: Istanza iniziale

# City insert ii

## Esecuzione

La soluzione di partenza è la migliore delle greedy  $\rightarrow s_0 = [0, 1, 3, 2, 4] \quad c(s_0) = 189$

## Passi dell'algoritmo:

1. Cancello il 3 e lo reinserisco dopo il 2  $\rightarrow s_1 = [0, 1, 2, 3, 4] \quad c(s_1) = 232$
2. Cancello il 3 e lo reinserisco dopo il 4  $\rightarrow s_2 = [0, 1, 2, 4, 3] \quad c(s_2) = 316$
3. Cancello l'1 e lo reinserisco dopo il 2  $\rightarrow s_3 = [0, 2, 1, 3, 4] \quad c(s_3) = 192$
4. Cancello il 2 e lo reinserisco dopo l'1  $\rightarrow s_4 = [0, 1, 2, 3, 4] \quad c(s_4) = 232$

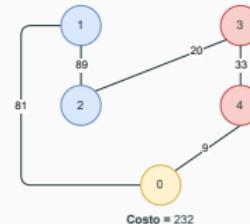
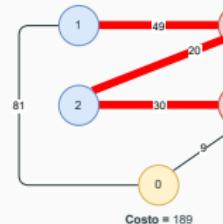
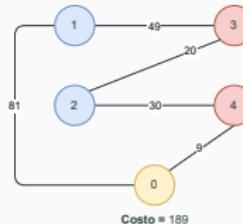


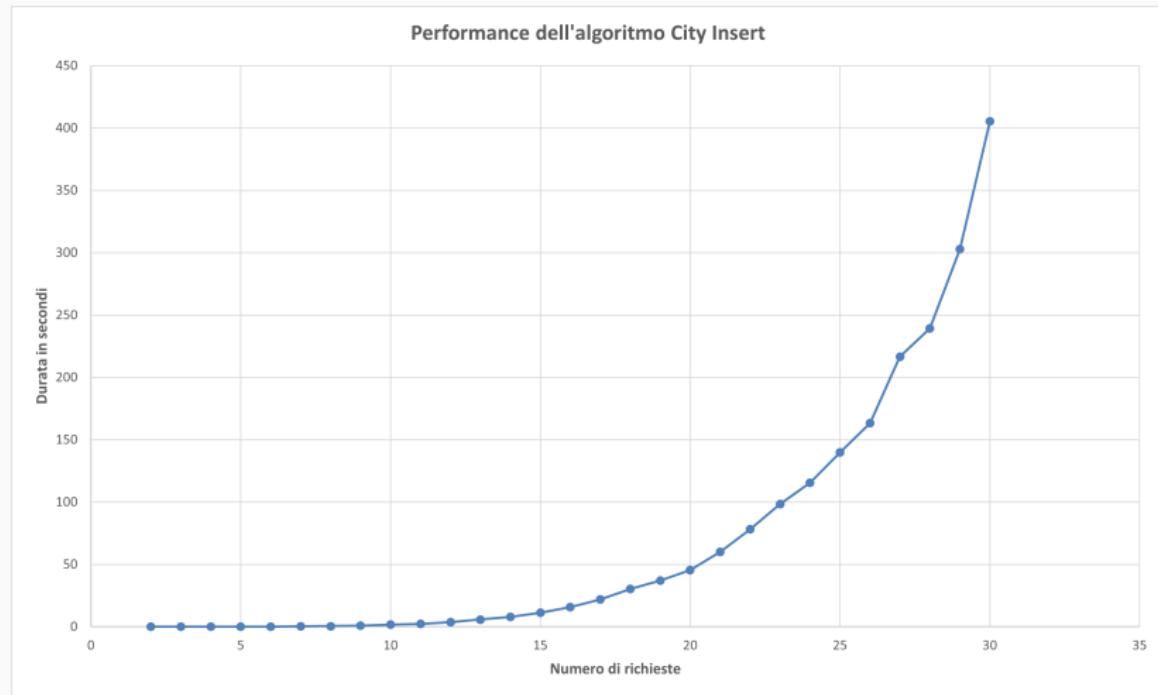
Figura 20: Soluzione  $s_0$

Figura 21: Archi eliminati al passo 1

Figura 22: Soluzione  $s_1$

Il passo 1 è un 3 opt limitato agli archi incidenti su questi nodi

## Performance nel tempo



- Gli algoritmi Node-based Neighborhood City Swap e City Insert portano ad un ottimo locale
- Ad ogni iterazione si tratta di risolvere un problema di ottimizzazione. Quindi la procedura restituisce, nel caso di best improvement, la migliore soluzione dell'intorno
- Resta intrappolata negli ottimi locali
- La soluzione prodotta è determinata dalla scelta dell'intorno, dalla strategia di esplorazione e dal punto iniziale, che determina come risultato l'ottimo locale del bacino di attrazione su cui è posizionato il punto iniziale (se si segue una strategia best improvement)

## **Algoritmi euristici di miglioramento**

Oltre la local search

---

Per uscire dal bacino di attrazione degli ottimi locali si devono accettare anche dei peggioramenti.

Per evitare di tornare su soluzioni già visitate e andare in loop è possibile:

- Accettare peggioramenti con probabilità decrescente (Simulated Annealing)
- Memorizzare le  $N$  ultime mosse eseguite (Tabu Search)
- Allargare la visione strettamente locale della local search

Annealing è il processo di raffreddamento secondo cui un solido raggiunge uno stato di energia minima che gli conferisce stabilità.

Il metodo converge all'ottimo globale con opportuni parametri.

### Descrizione dell'algoritmo

Ad ogni iterazione una soluzione è scelta a caso dall'intorno della soluzione corrente: l'intorno non viene esplorato, bensì campionato.

Se la mossa applicata alla soluzione porta ad un miglioramento, viene sempre accettata.

Altrimenti, la decisione se spostarsi ad una soluzione peggiore dipende da una funzione casuale  $e^{-\Delta/T}$  dove:

- $\Delta = \text{costo}(\text{sol}_{\text{ottenuta dal campionamento dell'intorno}}) - \text{costo}(\text{sol}_{\text{migliore corrente}})$
- $T$  = parametro temperatura che diminuisce durante l'euristica

### Parametri utilizzati nell'algoritmo

- Soluzione iniziale: la migliore soluzione ottenuta dale greedy
- Intorno utilizzato: city swap
- Temperatura iniziale: 900
- Tasso di raffreddamento: 0.92
- Temperatura finale: 0.1
- Iterazioni per temperatura: 5

### Codice

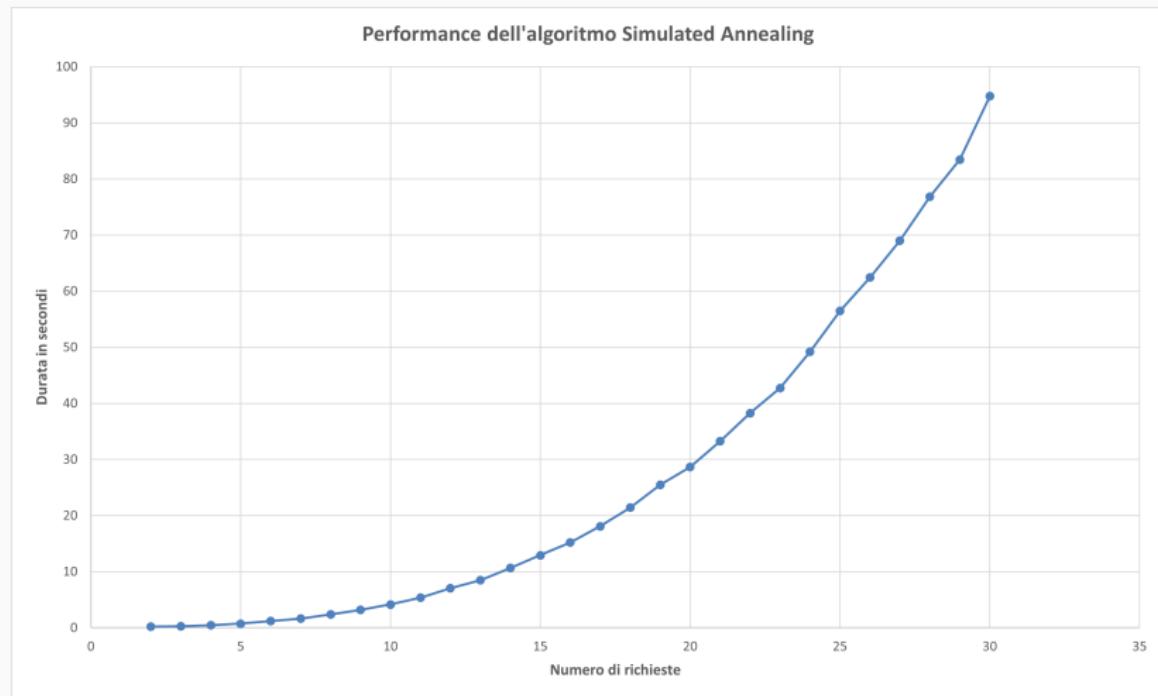
<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/simulatedAnnealing.py>

## Schema dell'algoritmo

```
Simulated Annealing(problem) return a solution
```

```
T ← determine a starting temperature
current ← generate an initial solution
best ← current
while (not yet frozen) do // finchè  $T > T_{frozen}$ 
    while not yet at equilibrium for this temperature do
        next ← a random solution selected from Neighbor(current)
         $\Delta E$  ← f(next) – f(current)
        if  $\Delta E < 0$  then current ← next // miglioramento
            if f(next) < f(best) then best ← next // aggiorno best
        else
            choose a random number r uniformly in [0, 1]
            if r <  $e^{-\Delta E/T}$  then current ← next // accetta peggioramento con prob  $e^{-\Delta E/T} > 0$ 
    end while
    lower the temperature T // equilibrio raggiunto per T
end while // processo di raffreddamento terminato
return best
```

## Performance nel tempo



### Descrizione dell'algoritmo

Ad ogni iterazione si seleziona la miglior soluzione dell'intorno, diversa dalla soluzione corrente, anche peggiore, così da poter sfuggire al bacino di attrazione dell'ottimo locale in cui mi trovo (mantenendo comunque in memoria le soluzioni migliori visitate).

Per evitare loop, e ritornare nello stesso bacino di attrazione utilizzo la *Tabu List* di dimensione fissata.

L'algoritmo termina (stop condition) nel caso in cui il massimo numero di iterazioni sia stato raggiunto oppure sia stato raggiunto il massimo numero di passi senza miglioramenti (stalllo).

**Eccezioni:** il criterio di aspirazione permette di visitare soluzioni tabù con valore di funzione obiettivo migliore di quella dell'ottimo candidato (sono sicuramente soluzioni mai visitate).

## Note

Nella lista tabù memorizzo le mosse proibite. Se durante la ricerca scambio due nodi (29 e 39) allora la mossa da memorizzare è [39,29]. Prima di scegliere una nuova soluzione dovrò sempre controllare che i due nodi selezionati per lo scambio non siano già presenti nella Tabu List.

## Parametri utilizzati nell'algoritmo

- Soluzione iniziale: la migliore soluzione ottenuta dale greedy
- Intorno utilizzato: city swap
- Dimensione della tabu list: 20
- Numero massimo di iterazioni: 20
- Stallo massimo: 15

## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/tabuSearch.py>

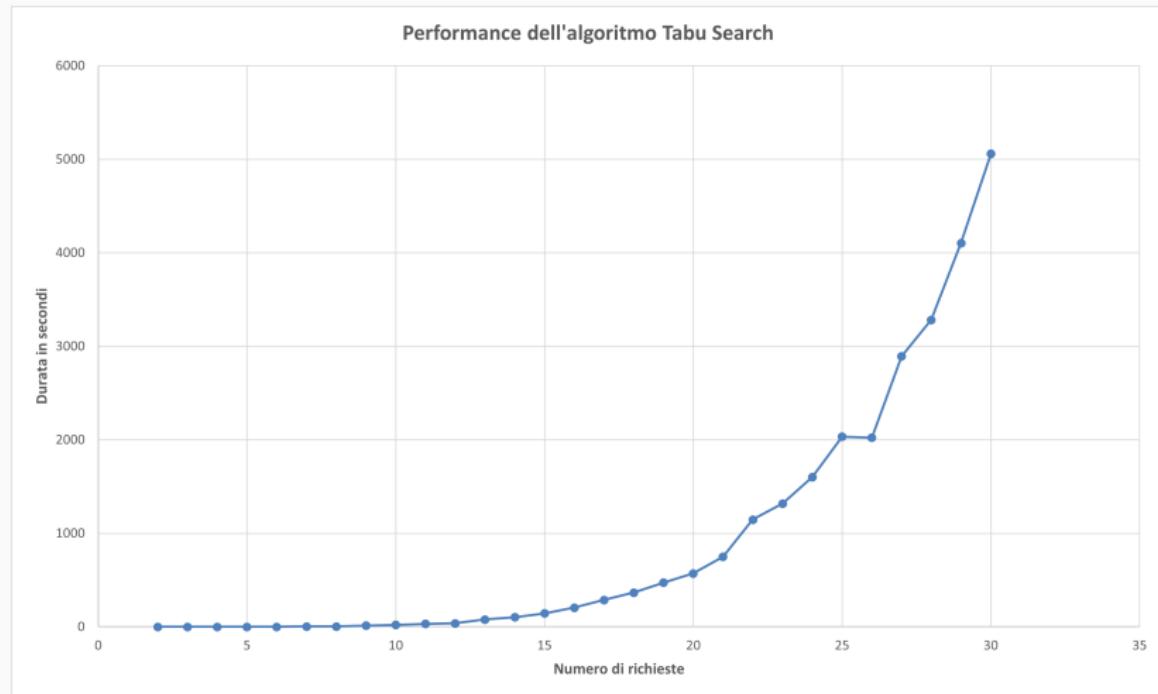
## Schema dell'algoritmo

```
// Step 1 (init)
k = 1
Build an initial solution  $S_1$ 
 $S_{best} = S_1$ 

// Step 2 (explore neighborhood)
At iter k
    select  $S_c \neq S_k \in Neighborhood(S_k)$  // possibly the best in  $N(S_k)$ 
    if  $f(S_c) < f(S_{best})$  then
         $S_{best} := S_c$ ,  $S_{k+1} := S_c$ 
        goto Step 3 // aspiration criteria
    if the move that transforms  $S_k \rightarrow S_c \in tabu - list$  then
        goto Step 2 // skip  $S_c$ 
    else // the move  $S_k \leftarrow S_c$  is not forbidden
         $S_{k+1} := S_c$  // update current solution
        insert the inverted move ( $S_c \rightarrow S_k$ ) in the tabu list
        remove the last tabu move from the tabu - list // FIFO policy
    goto Step 3

// Step 3 (stop or loop)
k = k + 1
if (stopping condition = true) then STOP
    return( $S_{best}$ )
else
    goto Step 2
```

## Performance nel tempo



- Per sfuggire agli ottimi locali con strategie che restano nell'ambito della filosofia Local Search è possibile utilizzare intorni di grandi dimensioni
- Maggiore è l'ampiezza dell'intorno e migliore la qualità dell'ottimo locale determinato dalla Local Search
- Esistono diverse strategie da adottare per l'esplorazione di intorni di grandi dimensioni, tipicamente si tratta di strategie non esatte
- Se la dimensione dell'intorno è vasta, non è possibile operare una esplorazione esaustiva esplicita
- Una possibilità è individuare un sottoinsieme dell'intorno in modo stocastico campionando casualmente le soluzioni nell'intorno secondo una legge probabilistica

- Le strategie di LNS appartengono alla famiglia delle VLNS e sono caratterizzate dal fatto che le soluzioni dell'intorno sono ottenute applicando una mossa di *destroy* e una di *repair*
- *Destroy* distrugge una parte della soluzione corrente, scelta secondo una procedura non deterministica in modo che la parte modificata vari ad ogni iterazione
- La procedura di *Repair* deve assegnare valore alle variabili “cancellate” in modo da ripristinare l’ammissibilità (e possibilmente nel modo migliore rispetto alla funzione obiettivo).
- L’intorno è implicitamente definito dall’insieme di tutte le soluzioni ottenibili applicando le procedure *destroy & repair*
- Il tasso di distruzione (% delle variabili rimesso in discussione) caratterizza l’intorno che generalmente è esponenziale rispetto alle dimensioni dell’istanza

## Template

Procedure *Large Neighborhood Search*

```
x0 = GenerateInitialSolution
k = 0
x* = xk
repeat
    x' = rep (dest(xk))
    if accept(x', xk) then xk = x', k = k + 1
        if c(xk) < c(x*) then x* = xk
until StopCondition
return (x*)
end
```

## Descrizione dell'algoritmo

La procedura mantiene 3 variabili:

- $x^*$  la migliore soluzione trovata (ottimo candidato)
- $x'$  la nuova soluzione tentativo
- $x^k$  la soluzione corrente all'iterazione k

*dest(.)* e *rep(.)* sono gli operatori di destroy e repair.  
La funzione *accept* può essere implementata in vari modi, il più semplice secondo il criterio di improvement rispetto alla soluzione corrente (accetto solo se miglioro).  
LNS non esplora l'intorno ma lo campiona, come nel *Simulated Annealing*.

## Parametri utilizzati nell'algoritmo

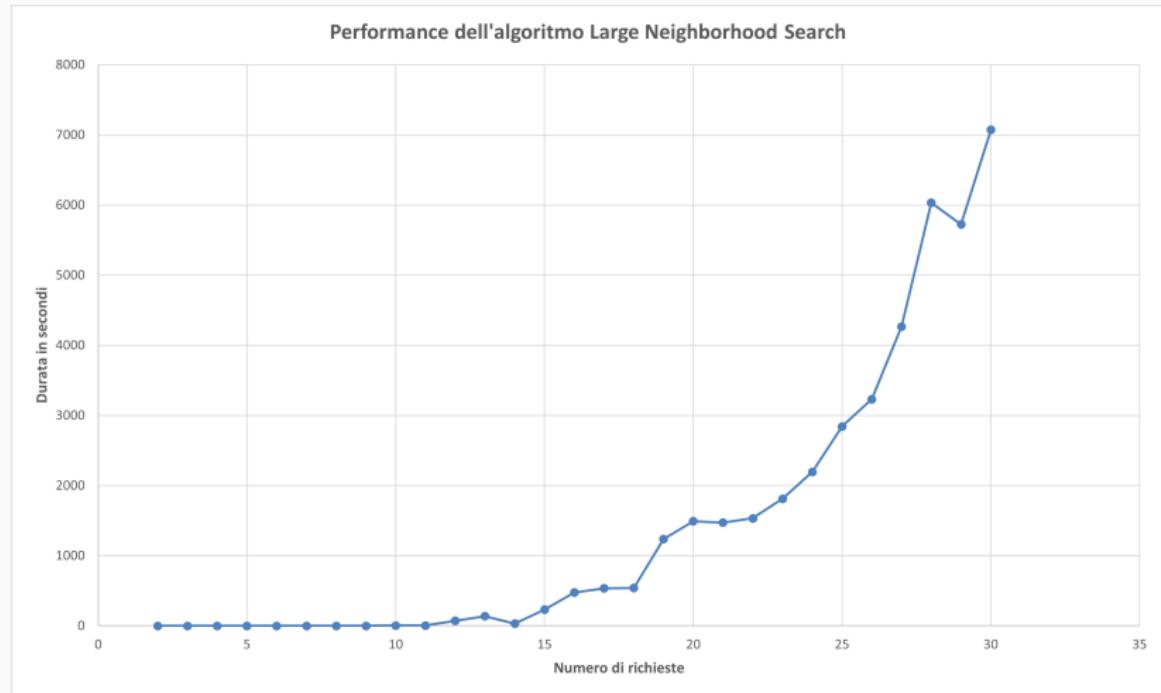
- Soluzione iniziale: la migliore soluzione ottenuta dale greedy
- Operatore destroy: rimuovo dalla soluzione un numero di nodi pari al tasso di distruzione (non rimuovo il primo e l'ultimo nodo, è il nodo deposito)
- Tasso di distruzione: 0.3
- Operatore repair: procedura “hand made”. Prima ricolloco i nodi di pickup e, ogni volta che reinserisco un nodo nella soluzione cerco di reinserirlo nel modo ottimale (in modo che il costo della soluzione sia minimo). Poi ricolloco i nodi di delivery e, ogni volta che reinserisco un nodo nella soluzione cerco di reinserirlo nel modo ottimale (in modo che il costo della soluzione sia minimo).
- Criterio di stop: ho raggiunto 2 miglioramenti oppure ho raggiunto 100 peggioramenti

## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/largeNeighborhoodSearch.py>

# Very large size neighborhood v

## Performance nel tempo



- Per sfuggire agli ottimi locali con strategie che restano nell'ambito della filosofia Local Search è possibile *agire sul punto di arrivo uscendo dall'ottimo locale corrente* attraverso una mossa di *diversificazione* che sposti il punto corrente, da cui riprendere la ricerca, nel bacino di attrazione di un altro ottimo locale
- Studi sperimentali hanno comprovato che per problemi di ottimizzazione combinatoria numerosi ottimi locali sono di bassa qualità
- Quindi è ragionevole cercare una strategia per continuare la ricerca anche dopo avere determinate un ottimo locale
- Ad esempio iterando il procedimento a partire da un altro punto iniziale
- *Occorre una strategia per il restart* che sfrutti il fatto che statisticamente le buone soluzioni sono clusterizzate
- Quindi piuttosto che ripartire da un punto casuale è meglio perturbare l'ottimo locale corrente

### Definizione dell'algoritmo

Itero più volte una ricerca locale, ogni volta partendo da un punto iniziale diverso

### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/multiStartLocalSearch.py>

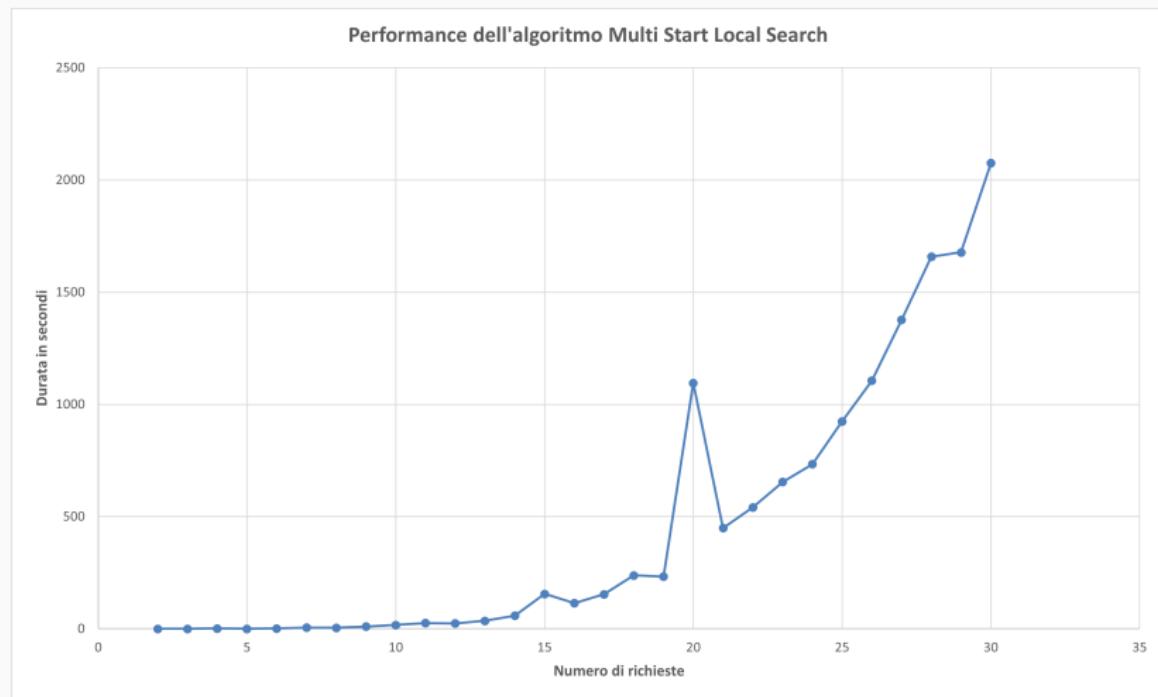
### Parametri utilizzati nell'algoritmo

- Iterazioni massime: 10
- Strategia per partire da un punto diverso: Greedy random (random restart / blind search)
- Local Search utilizzata: Large Neighborhood Search
- Criterio di stop: numero massimo di iterazioni raggiunto

### Rischi

- Se due soluzioni iniziali appartengono allo stesso bacino di attrazione, la ricerca converge allo stesso ottimo locale
- La generazione totalmente random del punto iniziale non permette un controllo a priori

## Performance nel tempo



- Per sfuggire agli ottimi locali con strategie che restano nell'ambito della filosofia Local Search è possibile agire *moltiplicando i punti di partenza* e riapplicare una nuova LS a partire da ciascuno di essi
- È naturale pensare a un algoritmo greedy per la costruzione della soluzione ammissibile di partenza di una ricerca locale
- Con le sole greedy e ricerca locale possiamo già creare delle meta-euristiche che forniscono risultati migliori dei singoli algoritmi
- Le *Meta-euristiche* sono metodi risolutivi che orchestrano una interazione tra procedure locali di miglioramento e strategie di livello superiore che permettono di sfuggire agli ottimi locali, e operare una esplorazione il più possibile robusta (indipendente dal punto di partenza) della regione ammissibile
- La Multi Start Local Search applica ripetutamente una ricerca locale LS a partire da punti iniziali diversi ottenuti in modo casuale. La generazione random dei punti iniziali ha i seguenti svantaggi:

- nessuna garanzia sulla *qualità* della *soluzione iniziale*, il che richiede molto sforzo computazionale per la fase iniziale della ricerca per migliorare una soluzione mediocre
  - spesso sono punti che appartengono al bacino di attrazione di ottimi locali (*soluzione finale di ogni LS*) non molto buoni
  - potenzialmente si converge più volte allo stesso ottimo locale (quando più punti iniziali appartengono allo stesso bacino di attrazione)
- La Greedy Randomized Adaptive Search Procedure è un'evoluzione della Multi Start Local Search. Intervenendo sulla generazione dei punti iniziali della Multi Start Local Search in modo tale da avere contemporaneamente le seguenti condizioni:
    - garantire una sufficiente diversità delle soluzioni prodotte all'iterare della procedura, tale che le soluzioni appartengano a bacini di attrazione di ottimi locali diversi, per non convergere più volte allo stesso ottimo locale
    - spesso sono punti che appartengono al bacino di attrazione di ottimi locali (*soluzione finale di ogni LS*) non molto buoni
    - non perdere troppo nella qualità della soluzione iniziale per non investire troppo nella prima fase di ogni LS

## Definizione dell'algoritmo

### Passo 1: Greedy & Random

Ad ogni passo della greedy seleziono a caso un elemento tra i migliori  $k$  (dimensione della restricted candidate list)

### Passo 2: Greedy & Random & Local Search

Applico una LS a ciascuna soluzione prodotta dalla greedy randomizzata, e ripetendo  $k$  volte la procedura.

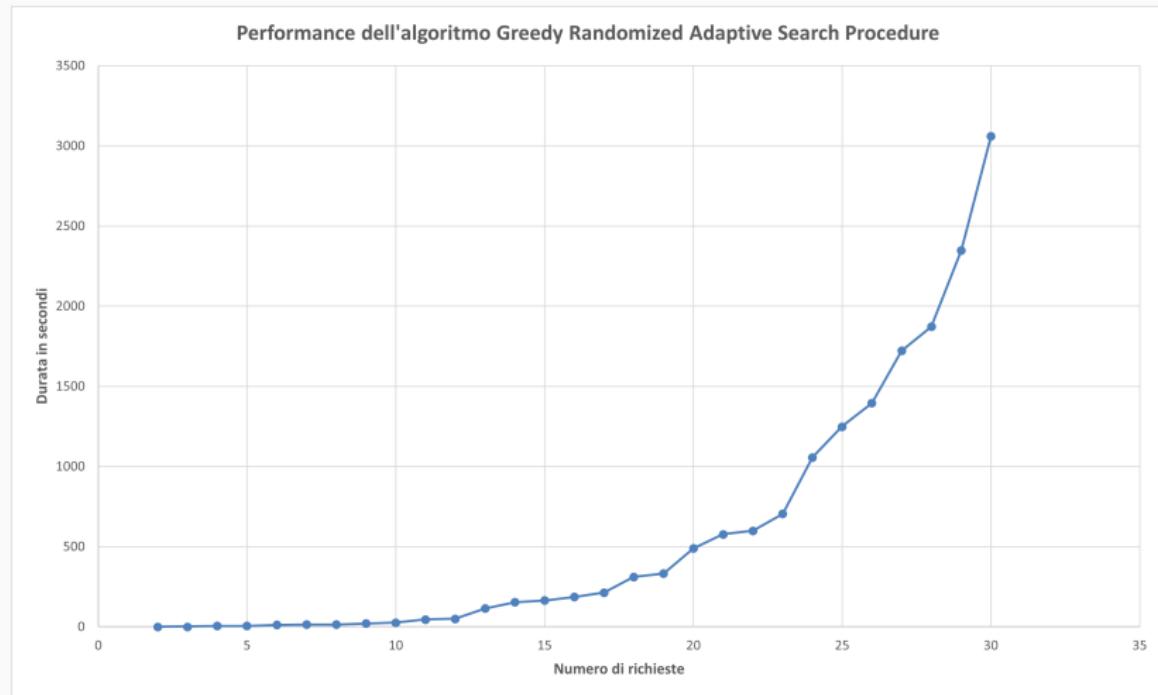
## Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyRandomizedAdaptiveSearchProcedure.py>

## Parametri utilizzati nell'algoritmo

- Iterazioni massime: 10
- Dimensione della restricted candidate list: 20
- Strategia per partire da un punto diverso: scelgo a caso un elemento dalla restricted candidate list
- Local Search utilizzata: Large Neighborhood Search
- Criterio di stop: numero massimo di iterazioni raggiunto

## Performance nel tempo



Un limite è dato dalla mancanza di memoria della procedura. Ogni restart è indipendente dai precedenti.

Guardando al profilo della funzione obiettivo su  $F$ , spesso si osserva che gli ottimi locali sono concentrati in una regione limitata (Proximate Optimality Principle).

## Definizione dell'algoritmo

### Passo 3: Greedy & Local Search & Path Relinking

Si raccolgono le soluzioni elite, e si opera una ricerca nello spazio ristretto alle soluzioni simili ad esse, secondo una tecnica, detta *PATH RELINKING*, che esplora le soluzioni che si ottengono operando delle mosse a partire da una soluzione elite di partenza  $p$  avvicinandosi a una soluzione elite target  $t$

### Path Relinking

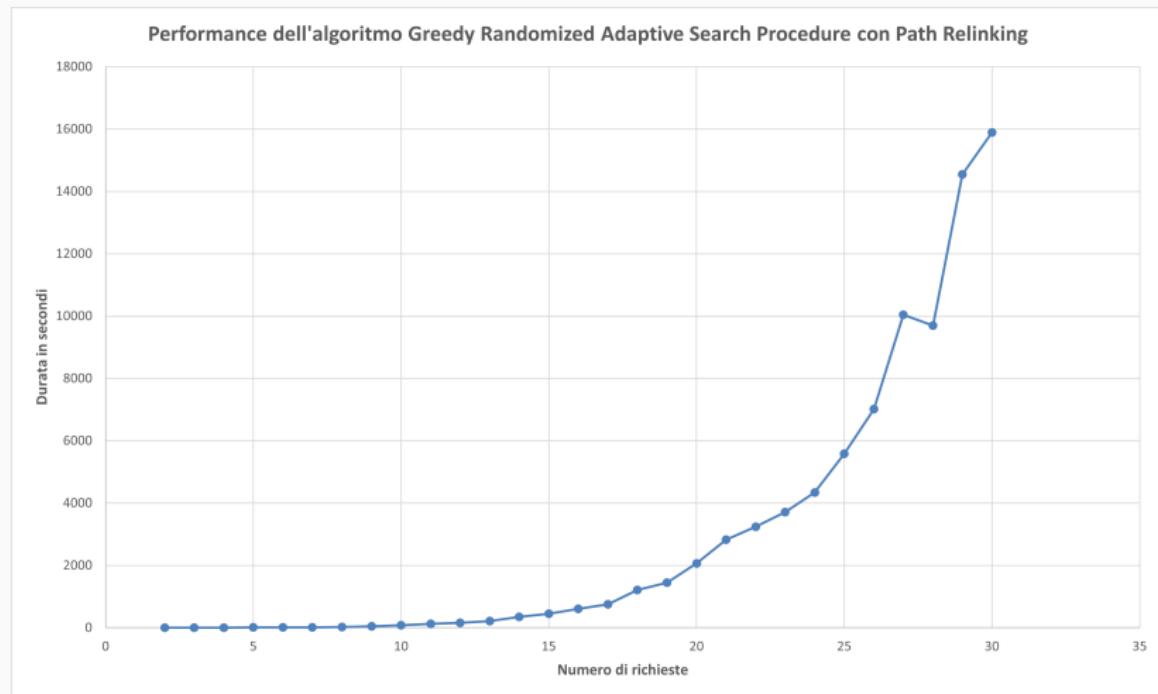
Partendo da una soluzione iniziale ( $s_i$ ), con la ricerca locale si ottiene una soluzione target  $s^*$ . Opero in successione delle mosse di scambio di componenti per avvicinarmi alla soluzione target. In pratica:

- Ad ogni passo, la mossa potenziale consiste nel modificare una componente di  $s_i$  sostituendola con una del target  $s^*$ , per ciascuna delle componenti per cui  $s_i$  e  $s^*$  differiscono. Si procede selezionando la mossa più conveniente
- Il valore della funzione lungo il percorso non è monotono, ma spesso lungo il percorso si individuano soluzioni migliori sia di  $s_i$  che di  $s^*$

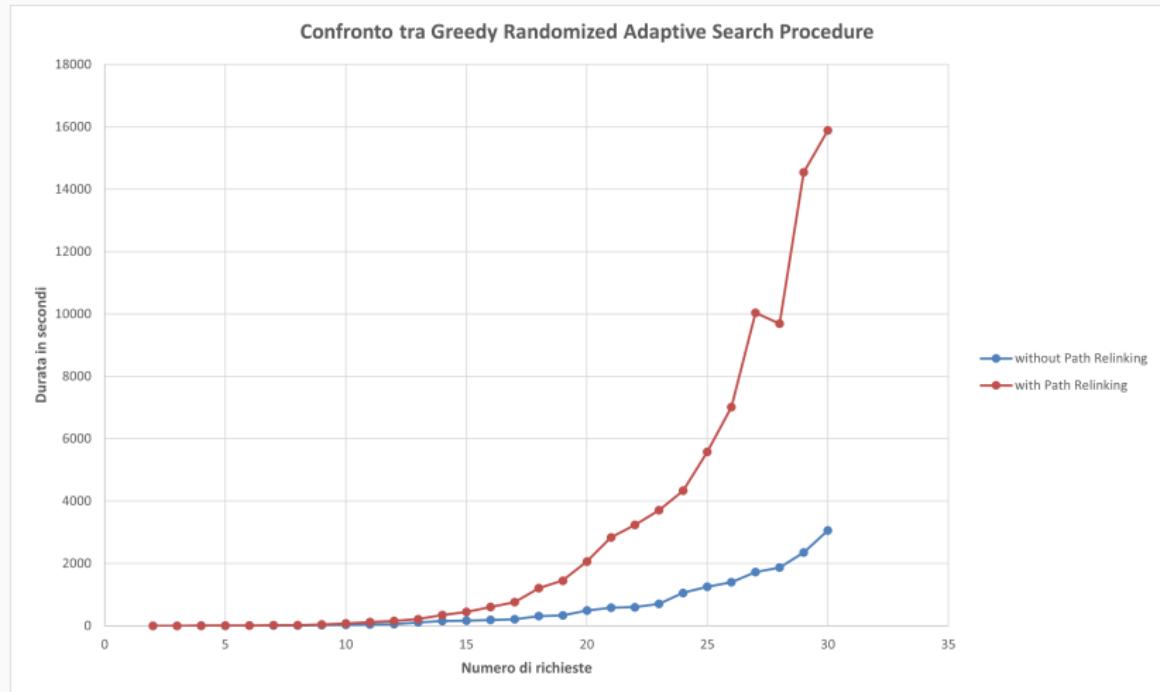
### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/greedyRandomizedAdaptiveSearchProcedure.py>

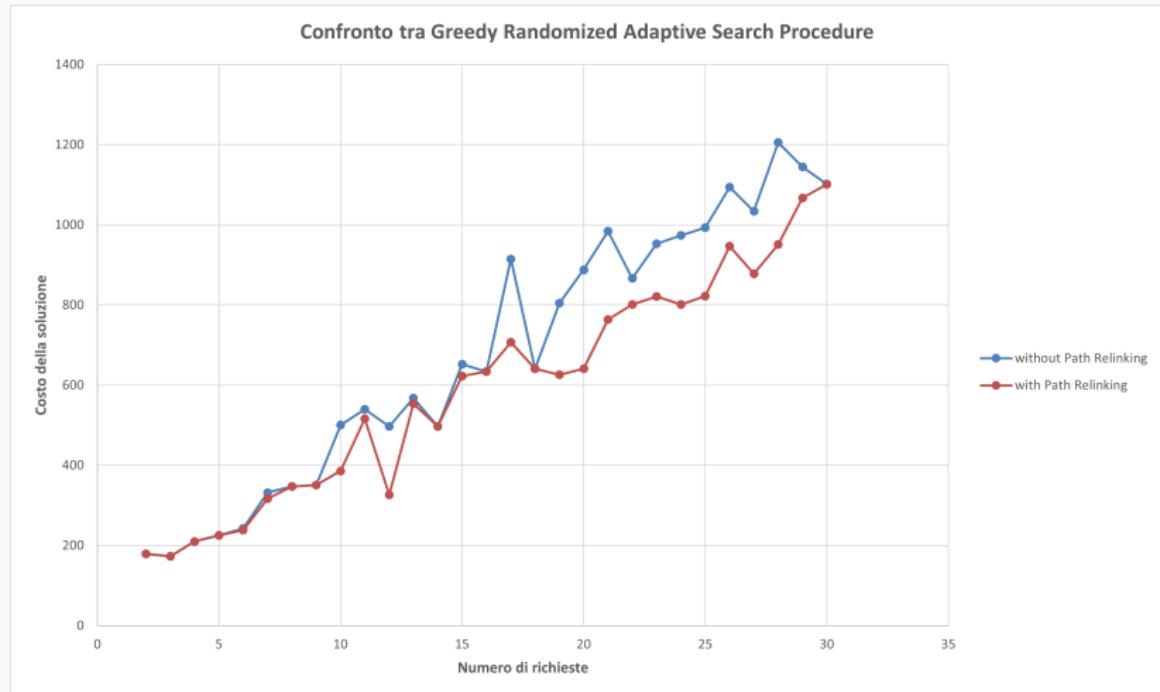
## Performance nel tempo



## Performance nel tempo



## Performance costo soluzione

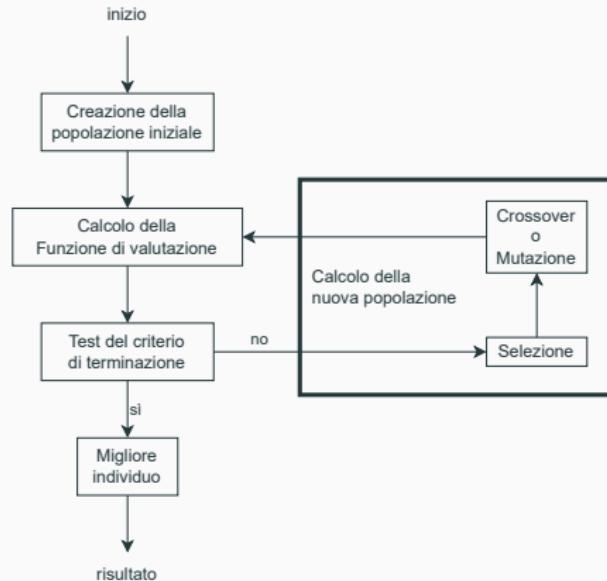


## **Algoritmi euristici population based**

---

# Genetic Algorithm i

Algoritmi basati sulla teoria dell'evoluzione di Darwin



**Figura 23:** Diagramma a blocchi del Genetic Algorithm

### Parametri utilizzati nell'algoritmo

- Numero elementi della popolazione iniziale:  $\# \text{ nodi}^2$  se  $\leq 200$ , altrimenti 200
- Numero di iterazioni massime:  $2 \times \# \text{ popolazione iniziale}$  se  $\leq 200$ , altrimenti 200
- Popolazione: generata utilizzando la *Greedy Random*
- Codifica: lista in ordine di visita dei nodi
- Metodo di scelta dei genitori: *Roulette Wheel*
- Crossover: custom (vedi slide 81)
- Elitismo: il migliore individuo passa alla generazione successiva sovrascrivendo il peggiore
- Tasso di mutazione: 0.3
- Condizione di stop: numero iterazioni massime raggiunto

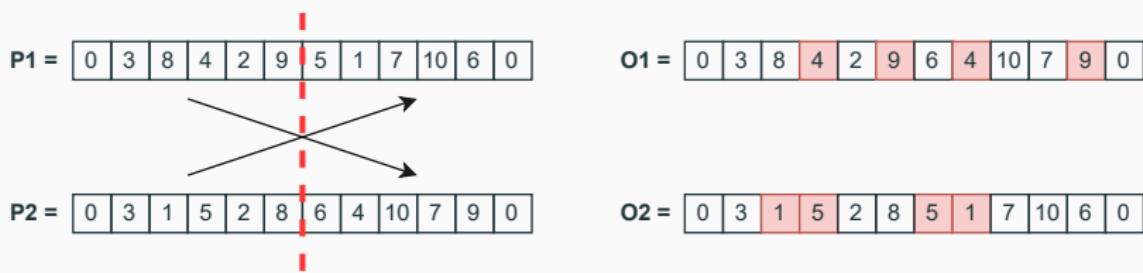
### Codice

<https://github.com/michele-vaccari/TSP-con-pick-up-and-delivery/blob/main/src/tsppd/solver/geneticAlgorithm.py>

## Crossover

Un individuo è un tour.

Il normale cross over a 1 punto non funziona.



I nodi 4, 9, sono visitati 2 volte in O1.

I nodi 1, 5, sono visitati 2 volte in O2.

## Genetic Algorithm iv



I nodi 8, 9, sono visitati 2 volte in O3.

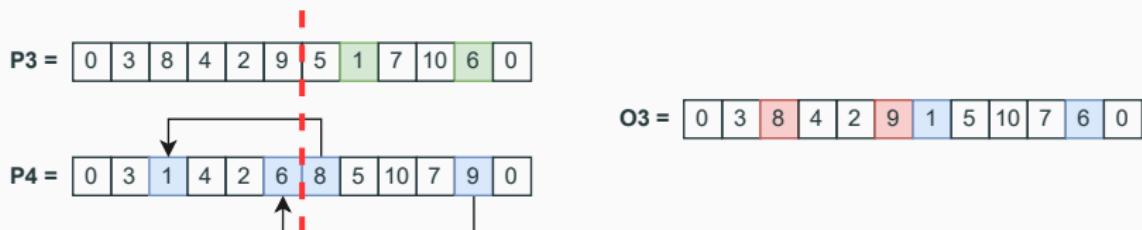
I nodi 1, 6, sono visitati 2 volte in O4.

I nodi visitati 2 volte possono essere o di pickup o di delivery.

## Caso 1: tutti i nodi visitati 2 volte sono delivery (vedi O3)

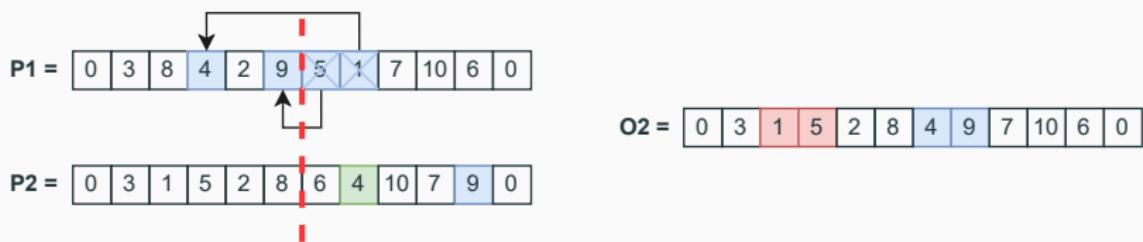
Si fa ereditare al figlio, dal secondo genitore, solo la posizione relativa dei nodi e non quella assoluta  $\Rightarrow$  nella prima parte del figlio si mantiene la posizione assoluta dei geni del primo genitore, e solo quella relativa dell'altro genitore nella seconda parte.

Ruolo asimmetrico dei genitori in ciascun figlio dovuto alla asimmetria nell'ereditare le parti della sequenza (nessun figlio eredita la posizione assoluta nella seconda parte della sequenza di alcun genitore, quell'informazione viene persa).

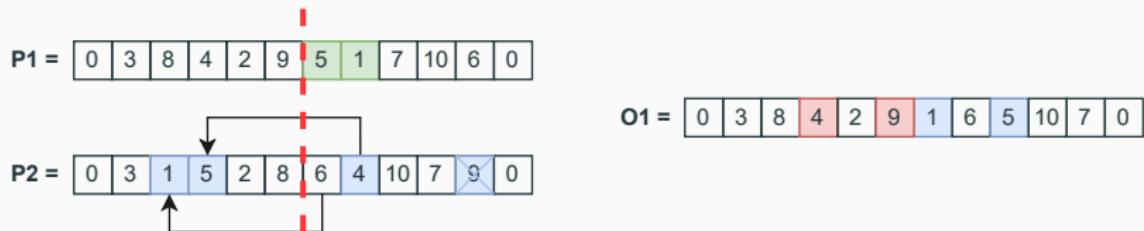


### Caso 2: tutti i nodi visitati 2 volte sono pickup (vedi O2)

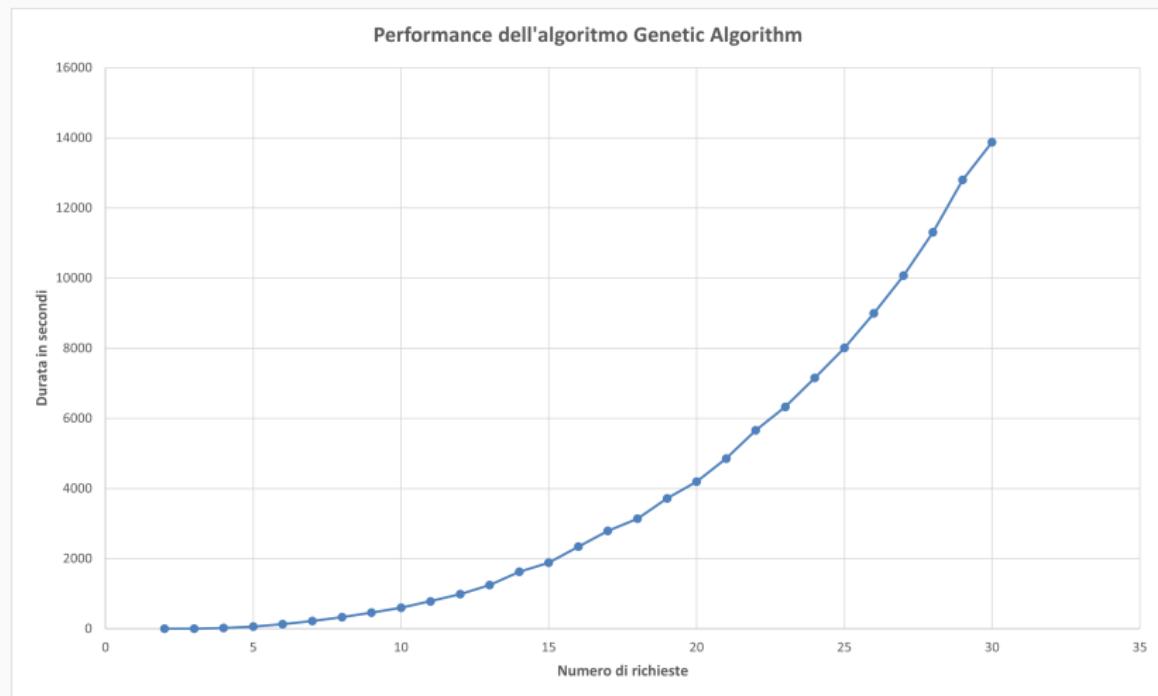
Nella prima parte del figlio si mantiene la posizione assoluta dei geni del primo genitore, nella seconda parte si cancellano i nodi duplicati e si inseriscono i nodi di pickup nella prima parte, rispettando l'ordine relativo.



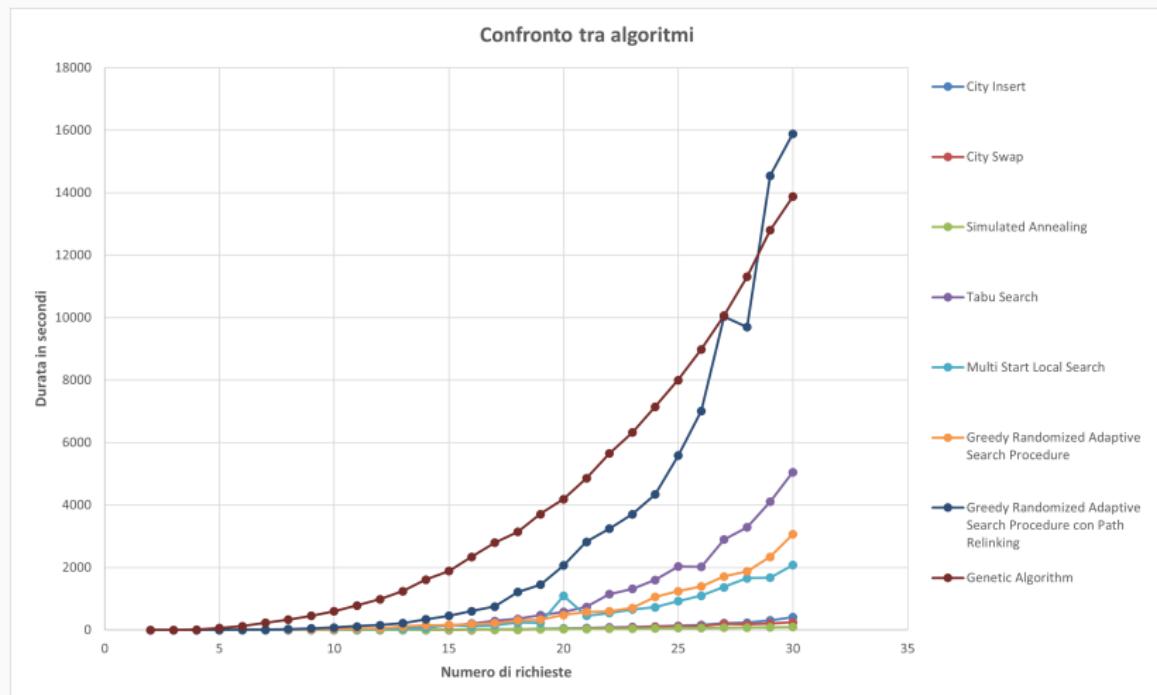
**Caso 3: i nodi visitati 2 volte sono sia pickup che delivery (vedi O1 e O4)**  
Tratto i nodi di delivery come il *Caso 1* mentre i nodi di pickup come il *Caso 2*



## Performance nel tempo

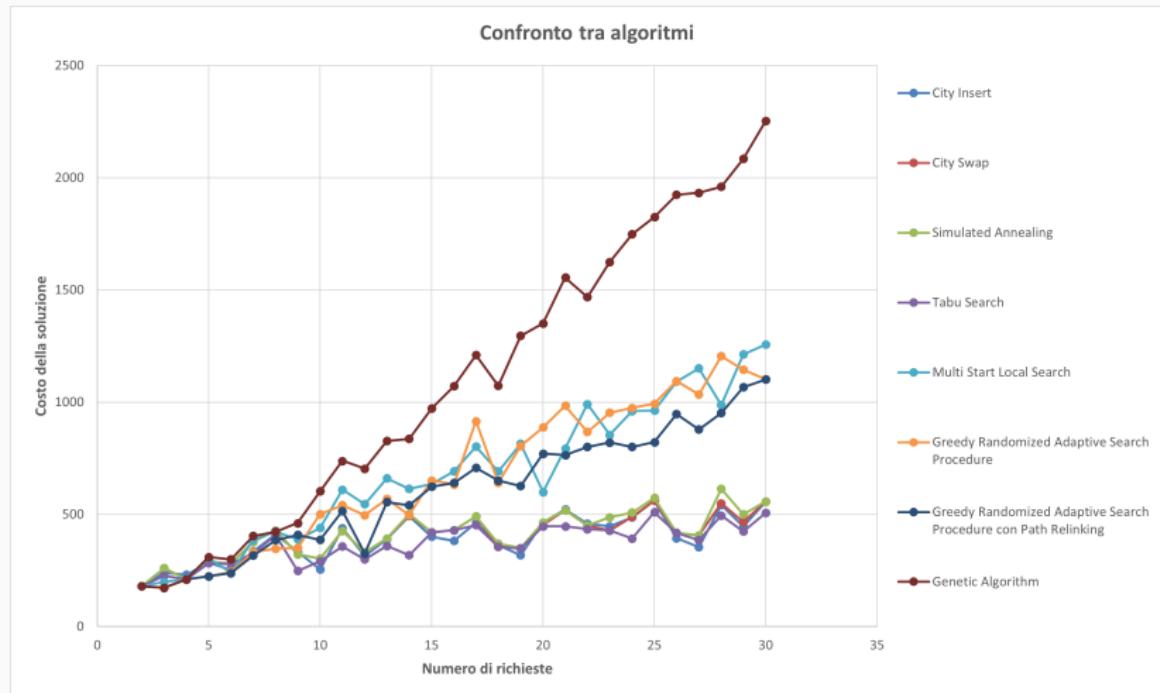


## Performance nel tempo



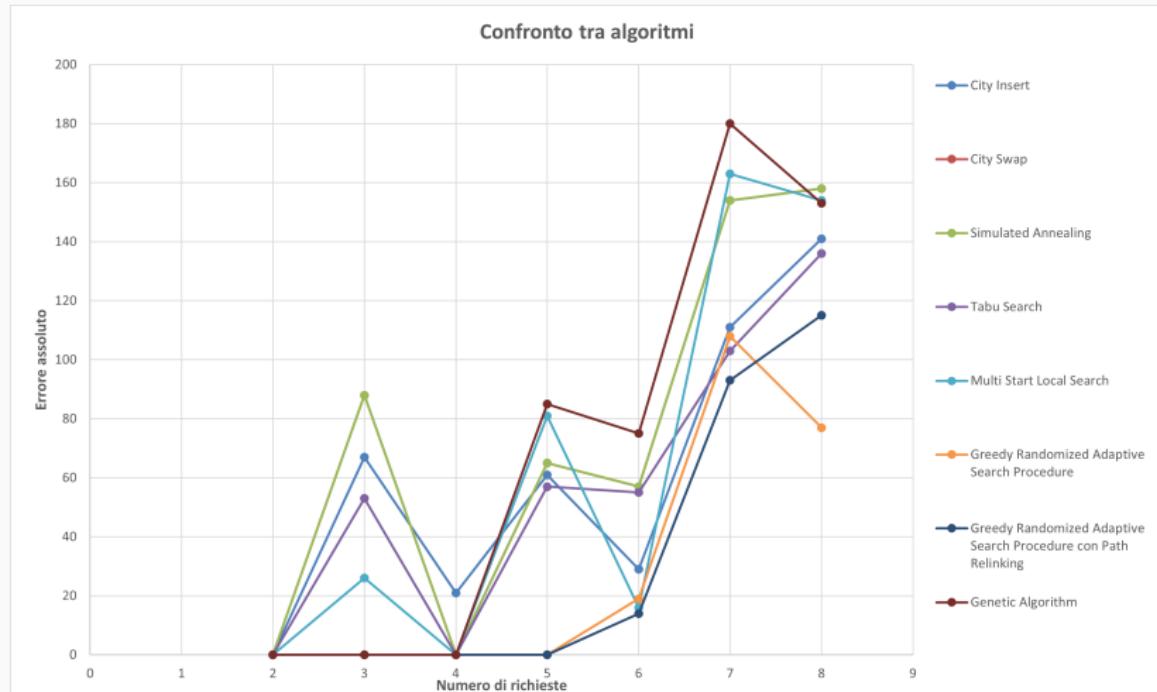
# Confronto ii

## Performance costo soluzione



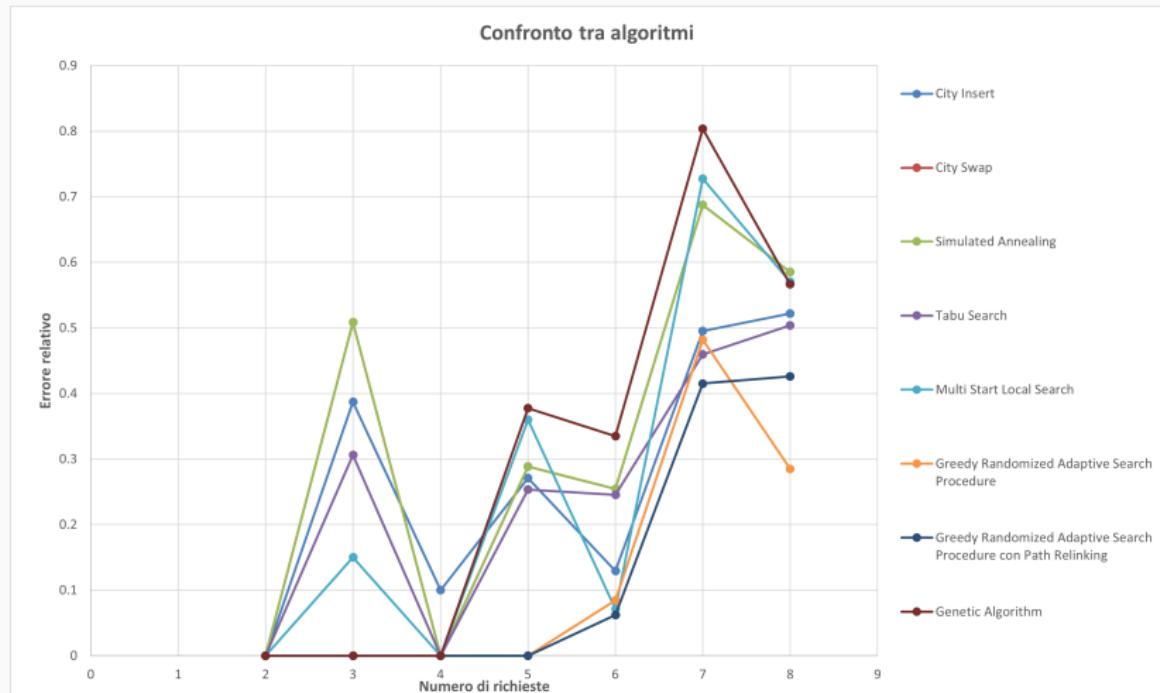
# Confronto iii

## Performance errore assoluto



## Confronto iv

### Performance errore relativo



- Per istanze del problema piccole (fino  $n = 6$ ) è possibile utilizzare il metodo enumerativo avendo la certezza di trovare un ottimo globale
- Le procedure euristiche di tipo Greedy, producono velocemente i risultati ma in generale non trovano soluzioni ottimali
- Si può osservare che nella maggior parte dei casi si è sempre riusciti a migliorare la soluzione iniziale trovata tramite le 3 greedy utilizzando i diversi algoritmi di miglioramento
- Tra gli algoritmi di miglioramento implementati il più veloce è il Simulated Annealing mentre l'algoritmo che da la miglior soluzione (con le istanze testate) è il Tabu Search
- In generale non c'è dominanza tra le varie euristiche. Con un tuning accurato è possibile migliorare ulteriormente i vari algoritmi sia in termini di soluzioni prodotte per una classe di istanze che in termini di velocità di esecuzione
- I vari algoritmi restituiscono soluzioni applicabili anche a contesti reali (è possibile integrare il codice sviluppato in piattaforme/applicativi reali)

-  S. Lin.  
**Computer solutions of the traveling salesman problem.**  
*Bell System Technical Journal*, 44(10):2245–2269, 1965.
-  R. J. O'Neil and K. Hoffman.  
**Exact methods for solving traveling salesman problems with pickup and delivery in real time.**  
*Optimization-Online. org*: [http://www.optimization-online.org/DB\\_HTML/2017/12/6370.html](http://www.optimization-online.org/DB_HTML/2017/12/6370.html). Accessed, 9, 2018.
-  K. Ruland and E. Rodin.  
**The pickup and delivery problem: Faces and branch-and-cut algorithm.**  
*Computers & mathematics with applications*, 33(12):1–13, 1997.

**Grazie per l'attenzione**