Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
*Corso di Laurea in Informatica*

# FURTHEST INSERTION: UNA NUOVA EURISTICA PER IL TSP

**Relatore:** Prof. Giovanni RIGHINI

Tesi di:
Asaf COHEN
Matricola: 975599

Anno Accademico 2023-2024

# Ringraziamenti

*dedicato a DA COMPLETARE...*

# Abstract

Il Problema del Commesso Viaggiatore (TSP) rappresenta una delle sfide più interessanti e rilevanti nell'ambito dell'ottimizzazione combinatoria. Originariamente formulato negli anni '30, il TSP richiede di determinare il percorso più breve per visitare un insieme di città esattamente una volta, ritornando infine alla città di partenza. Nonostante la sua apparente semplicità concettuale, il TSP è noto per la sua complessità computazionale e la sua rilevanza pratica in una vasta gamma di settori, inclusi trasporti, logistica, e progettazione di circuiti.

Il TSP è classificato come un problema NP-hard, il che significa che non esiste un algoritmo efficiente in grado di risolvere tutte le istanze del problema in tempo polinomiale. Di conseguenza, sono state sviluppate numerose euristiche e approcci approssimati per trovare soluzioni accettabili in un tempo ragionevole. Le euristiche sono strategie di ricerca che, pur non garantendo la soluzione ottima, sono in grado di produrre risultati soddisfacenti entro limiti temporali praticabili.

In questa tesi, esploreremo una specifica euristica per il TSP chiamata Furthest Insertion: l'obiettivo principale sarà quello di presentare, analizzare e valutare l'efficacia di questa euristica attraverso run su istanze presenti su TSP-LIB e confronti con altre tecniche note.

La scelta di concentrarsi su un'euristica per il TSP è motivata dalla necessità di affrontare problemi di dimensioni reali in contesti applicativi. Mentre le soluzioni esatte sono desiderabili per la loro precisione, spesso richiedono una potenza di calcolo eccessiva per problemi di grandi dimensioni. Le euristiche offrono un compromesso utile tra precisione e efficienza, consentendo di ottenere soluzioni praticabili che possono guidare decisioni reali.

Questa tesi sarà strutturata nel seguente modo: innanzitutto, forniremo una panoramica del TSP e della sua formulazione matematica. Successivamente, esamineremo le principali categorie di approcci risolutivi, concentrandoci in particolare sulle euristiche basate su inserzione. Presenteremo quindi la nuova euristica Furthest Insertion discutendo la sua implementazione e le scelte progettuali adottate. Infine, concluderemo con un'analisi dei risultati ottenuti, identificando i punti di forza e le limitazioni della nuova euristica.

# Indice

# Capitolo 1

# Il Problema del Commesso Viaggiatore

Il Problema del Commesso Viaggiatore (TSP) è una delle sfide più emblematiche e studiate nell'ambito della ricerca operativa e dell'ottimizzazione combinatoria. Originariamente formulato negli anni '30 da Karl Menger [1], il TSP richiede di determinare il percorso più breve per visitare un insieme di città esattamente una volta, tornando infine alla città di partenza. Nonostante la sua semplice descrizione concettuale, il TSP è noto per la sua complessità computazionale e la sua rilevanza pratica in una vasta gamma di settori.

Le applicazioni del TSP sono diffuse e impattano direttamente su molte attività quotidiane. Ad esempio, nel settore della logistica e della gestione delle catene di distribuzione, il TSP è utilizzato per ottimizzare le rotte dei veicoli di consegna, minimizzando i costi di carburante e il tempo impiegato. In ambito produttivo, il TSP viene impiegato per pianificare i percorsi di ispezione delle fabbriche o per ottimizzare il flusso di lavoro all'interno di un'azienda. Anche nei sistemi di navigazione satellitare e nelle applicazioni GPS, il TSP è alla base dell'ottimizzazione dei percorsi per ridurre il tempo di viaggio.

Storicamente, il TSP ha attratto l'attenzione di numerosi matematici e informatici in quanto è un problema semplice da formulare ma complesso da "risolvere". Il problema è stato formalizzato e reso noto grazie al lavoro di Hassler Whitney nel 1952[1] e successivamente nel 1954 da Merrill Flood. La dimostrazione della sua appartenenza alla classe di complessità NP-hard è stata fondamentale per stimolare lo sviluppo di tecniche approssimate e euristiche. Le sfide legate al TSP sono principalmente dovute alla sua natura combinatoria: per n città, il numero di possibili percorsi da valutare cresce in modo esponenziale con n, rendendo impraticabile un'analisi esaustiva per istanze di grandi dimensioni. Questa complessità ha spinto alla ricerca di approcci

[1] https://www.math.uwaterloo.ca/tsp/uk/history.html

efficienti, come le euristiche, che non garantiscono la soluzione ottimale ma forniscono soluzioni accettabili in tempi ragionevoli.

## 1.1 Il Problema del commesso viaggiatore

Il problema del commesso viaggiatore (TSP) può essere sintetizzato molto semplicemente con la seguente domanda: "Date n città, qual è il percorso più breve che inizia e termina con la stessa città?". Il problema quindi presenta le caratteristiche tipiche di un problema su un grafo, dove il grafo è composto da $n$ vertici (le città) e dove gli archi indicano le distanze euclidee tra le città. La formulazione classica del TSP può essere descritta matematicamente attraverso la programmazione intera lineare. La seguente formulazione fa riferimento alla formulazione MTZ (Miller-Tucker-Zemlin) e alla formulazione DFJ (Dantzig-Fulkerson-Johnson) [2].

## 1.2 Formulazione del problema

**Dati**: Consideriamo un insieme di $n$ città, ogni città $i$ (per $i = 1, 2, ..., n$) rappresenta un punto nello spazio euclideo, ogni città quindi ha coordinate $(x_i, y_i)$. Definiamo la matrice $c$, dove $c_{ij}$ indica la distanza euclidea tra le due città $i$ e $j$.

**Variabili**: La variabile $x_{ij}$ è una variabile binaria, quindi:

$$x_{ij} \in \{0, 1\} \qquad \forall i, j = 1, 2, 3, ..., n$$

La variabile $x$ assume valore 0 se l'arco che collega la città $i$ e $j$ non fa parte del path, 1 se ne fa parte.

**Vincoli**: I vincoli sono i seguenti:

1. Per ogni città $i$ deve essere presente un solo arco uscente corrispondente nel tour, quindi la somma delle variabili $x_{ij}$ deve essere uguale a 1.

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1, 2, 3, ..., n$$

2. Per ogni città $j$ deve essere presente un solo arco entrante corrispondente nel tour, quindi la somma delle variabili $x_{ij}$ deve essere uguale a 1.

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1, 2, 3, ..., n$$

3. Non devono essere presenti cicli all'interno del tour, quindi deve valere:

$$\sum_{i \in Q} \sum_{j \in Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{2, 3, ..., n\}$$

**Funzione Obiettivo**: Si vuole minimizzare il costo totale del tour, quindi:

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

La seguente formulazione permette quindi di identificare la soluzione ottima.

È stato dimostrato che TSP è un problema NP-hard [6], questo implica che attualmente non sono noti algoritmi con complessità polinomiale che risolvono il problema. Se esistesse un algoritmo con complessità polinomiale che risolve il TSP allora si potrebbe dimostrare che vale P = NP e quindi si risolverebbe uno dei più grandi problemi aperti nella teoria della complessità computazionale.

## 1.3 Metodi esatti per il TSP

Esistono vari approcci nella ricerca della soluzione ottima [3]. Un primo approccio "naive" per il problema può essere l'approccio brute-force: consiste nell'enumerare tutti i possibili percorsi e successivamente selezionare il migliore. In questo caso è necessario analizzare $n!$ possibili percorsi (nel peggiore dei casi), per questo motivo tentare di risolvere una istanza TSP con un approccio brute-force implica una complessità computazionale pari a $O(n!)$ e quindi un tempo che risulta rapidamente inaccettabile. Di seguito una tabella che illustra il numero di percorsi da valutare con un approccio basato su ricerca esaustiva al variare del numero n.

| Numero città | Numero percorsi possibili |
|---|---|
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |
| 7 | 5,040 |
| 8 | 40,320 |
| 9 | 362,880 |
| 10 | 3,628,800 |
| 11 | 39,916,800 |
| 12 | 479,001,600 |
| 13 | 6,227,020,800 |
| 14 | 87,178,291,200 |
| 15 | 1,307,674,368,000 |
| 16 | 20,922,789,888,000 |
| 17 | 355,687,428,096,000 |
| 18 | 6,402,373,705,728,000 |
| 19 | 121,645,100,408,832,000 |
| 20 | 2,432,902,008,176,640,000 |
| 21 | 51,090,942,171,709,440,000 |
| 22 | 1,124,000,727,777,607,680,000 |
| 23 | 25,852,016,738,884,976,640,000 |
| 24 | 620,448,401,733,239,439,360,000 |
| 25 | 15,511,210,043,330,985,984,000,000 |

Si può notare come il numero cresce molto rapidamente, anche con istanze relativamente piccole (ad esempio 20 città). Questo approccio risulta quindi inutilizzabile nei problemi reali dove può essere necessario analizzare istanze con migliaia di città.

Esistono altri algoritmi esatti i quali consentono di ridurre il numero di possibili soluzioni (percorsi) da analizzare, ad esempio approcci Branch and Bound [4]. La complessità computazionale nel caso peggiore resta esponenziale in quanto può essere necessario analizzare un numero esponenziale di percorsi e quindi non applicabile su problemi reali.

Oltre all'approccio brute force, esistono in letteratura diversi algoritmi **branch-and-bound**[1][2] che consentono di risolvere in problema in modo più efficiente. L'idea alla base consiste nel suddividere il problema originario in sottoproblemi (più semplici)

[1] https://www.math.cmu.edu/ bkell/21257-2014f/tsp.pdf
[2] https://apps.dtic.mil/sti/tr/pdf/ADA142318.pdf

i quali possono essere a loro volta scomposti in ulteriori sottoproblemi. Il termine "bound" indica il fatto che per ogni sottoproblema viene calcolato il limite inferiore $L_i$ (quindi la soluzione ottima non sarà mai migliore di questo limite inferiore) e si tiene conto della soluzione migliore trovata fino a quel punto $U$: quando $L_i$ è "peggiore" di $U$ allora è possibile scartare a priori l'intero sottoproblema in quanto è già stata trovata una soluzione migliore del limite inferiore del sottoproblema (questo processo prende il nome di "pruning"). Il termine "branch" invece fa riferimento al fatto che i sottoproblemi vengono a loro volta divisi in ulteriori sottoproblemi (come visto prima), l'algoritmo branch-and-bound procede esplorando l'albero dei sottoproblemi generati (detto anche albero branch-and-bound) e scartando i sottoproblemi (pruning) con il criterio visto prima. I vari algoritmi differiscono in base al criterio di branching e di esplorazione dell'albero branch-and-bound.

Lo stato dell'arte degli algoritmi esatti per il TSP non sono i generici branch-and-bound ma sono algoritmi **branch-and-cut**[8]: Branch and Cut è un metodo di ottimizzazione combinatoria simile al branch and bound, ma con alcune differenze fondamentali. Entrambi gli algoritmi mirano a trovare la soluzione ottimale per un problema complesso, dividendolo in sottoproblemi più piccoli e scartando (pruning) i sottoproblemi che in ogni caso non possono portare ad una soluzione parziale migliore di quella già nota. Tuttavia, l'algoritmo Branch and Cut si distingue per l'utilizzo di vincoli di taglio (cutting planes). Di seguito viene illustrato un generico algoritmo branch-and-cut:

1. Aggiungi il problema iniziale $ILP$ alla lista dei problemi attivi $L$.
2. Sia $x^* = null$ e $v^* = -\infty$
3. Finché $L$ non è vuota:
   - 3.1. Seleziona e rimuovi un problema da L
   - 3.2. Risolvi il rilassamento continuo di L
   - 3.3. Se il problema è inammissibile torna al punto 3 (ciclo while). Altrimenti sia $x$ la soluzione trovata e $v$ il valore della funzione obiettivo.
   - 3.4. Se $v \leq v^*$ torna al punto 3
   - 3.5. Se $x$ è intero, aggiorna $v^* \leftarrow v$, $x^* \leftarrow x$ e torna al punto 3.
   - 3.6. Cerca dei piani di taglio che sono violati da $x$, se sono presenti aggiungili al rilassamento continuo di ILP e torna al punto 3.2
   - 3.7. Dividi il problema in sottoproblemi, aggiungi i sottoproblemi alla lista dei problemi attivi $L$, torna al punto 3.
4. Restituisci $x^*$

Come già detto in precedenza, questo approccio è attualmente lo stato dell'arte per risolvere il TSP all'ottimo. In letteratura esistono varie implementazioni di algoritmi branch-and-cut per il TSP[5], sono presenti anche all'interno di risolutori MILP come ad esempio CPLEX. Questo approccio risulta migliore rispetto all'approccio brute-force, ma resta comunque non applicabile su istanze TSP con migliaia di città.

# Capitolo 2

# Euristiche per il TSP

Come discusso nel capitolo 2, i metodi esatti consentono di ottenere soluzioni ottime per il problema del TSP, ma il tempo per ottenere queste soluzioni aumenta esponenzialmente all'aumentare del numero di città presenti nell'istanza. Per affrontare il TSP, sono stati sviluppati numerosi approcci euristici, ovvero metodi che, pur non garantendo la soluzione ottimale, offrono soluzioni di buona qualità in tempi ragionevoli. Le euristiche sono fondamentali per applicazioni pratiche dove la rapidità di calcolo è essenziale. In questo capitolo esploreremo diverse tecniche euristiche, come le euristiche costruttive, che costruiscono una soluzione passo dopo passo a partire da una soluzione parziale, vedremo in particolare sulle euristiche basate su inserzione (Furthest Insertion fa parte di questa categoria) e poi "improvement euristics" ovvero le euristiche basate su meccanismi che consentono di "migliorare" la soluzione parziale trovata modificando il tour.

## 2.1 Nearest Neighbor

Nearest Neighbor è probabilmente l'euristica costruttiva più semplice per il TSP: si costruisce il tour selezionando sempre la città più vicina all'ultima appena aggiunta al tour. Appartiene alla categoria delle euristiche costruttive in quanto aggiunge via via nuove città al tour (soluzione parziale) senza modificare il tour costruito fino a quel punto. L'algoritmo è il seguente:

1. Seleziona un nodo arbitrario j, sia l = j e L = $\{1, 2, ..., n\} \setminus \{j\}$.
2. Finché $L \neq \emptyset$:
   2.1. Sia $j \in L$ tale che $c_{lj} = min\{c_{li} \mid i \in L\}$.
   2.2. Connetti l a j e rimuovi j da L, quindi $L = L \setminus \{j\}$.
3. Connetti l al primo nodo (era stato selezionato al punto 1) per formare un tour.

Nearest Neighbor è una euristica con complessità temporale pari a $O(n^2)$ in quanto per ogni nodo nel tour (quindi n volte) è necessario cercare tra i restanti nodi, qual è il più vicino (al massimo n volte).

## 2.2 Euristiche basate su Inserzione

Le euristiche basate su inserzione appartengono alla categoria delle euristiche costruttive ma a differenza di Nearest Neighbor la soluzione viene costruita in un modo differente: si inizia da un piccolo tour (che include 2 o 1 città) il quale viene via via esteso includendo i nodi (città) non ancora aggiunti al tour. In questo tipo di euristiche la differenza fondamentale che le distingue sarà il criterio con il quale si aggiungono nuove città al tour e dove queste città devono essere inserite nel tour. Una qualunque euristica basata su inserzione quindi presenta la seguente struttura:

1. Seleziona una città o due città arbitrarie come tour iniziale $T$. Sia $L$ l'insieme delle città che sono fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   2.1. Seleziona un nodo $j \in L$ secondo un certo criterio.
   2.2. Inserisci $j$ nel tour in una determinata posizione.
   2.3. Rimuovi $j$ da L, quindi $L = L \setminus \{j\}$.

Questo schema permette di ottenere un ciclo Hamiltoniano e quindi un percorso valido per il problema del TSP. Le varie euristiche andranno a definire un criterio di scelta del nodo j (punto 2.1) e un criterio di scelta della posizione di inserzione (punto 2.2).

### 2.2.1 Nearest Insertion

Nearest Insertion seleziona come nodo da inserire nel tour il nodo più vicino ad un qualunque nodo già inserito nel tour e lo inserisce nella posizione che minimizza il costo di inserimento. Di seguito l'algoritmo:

1. Sia $T$ il tour iniziale definito dalla coppia delle città più lontane. Sia $L$ l'insieme delle città fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   - 2.1. Sia $d_{min}(v)$ la distanza minima tra il nodo $v$ e il tour $T$, ovvero $d_{min}(v) = min\{c_{vt} | t \in T\}$. Seleziona il nodo $r \notin T$ in modo da minimizzare $d_{min}(r)$.
   - 2.2. Determina la posizione ottima di inserzione $(a, b)$ in modo da minimizzare il costo di inserimento $c_{ar} + c_{rb} - c_{ab}$, inserisci $r$ tra $(a, b)$
   - 2.3. Rimuovi $r$ da L, quindi $L = L \setminus \{r\}$.

Nearest Insertion ha complessità computazionale pari a $O(n^2)$ in quanto ogni nodo (n iterazioni) deve essere inserito nel modo migliore possibile nel tour (fino a n iterazioni).

### 2.2.2 Cheapest Insertion

Cheapest Insertion opera in modo diverso da Nearest Insertion: il criterio di selezione non è più il nodo più vicino al tour, ma diventa il nodo che produce il minor aumento del costo del tour se inserito (quindi minimizza il costo di inserzione). Una volta selezionato, il nodo viene inserito nel modo migliore possibile, quindi minimizzando il costo di inserimento. Di seguito l'algoritmo:

1. Sia $T$ il tour iniziale definito dalla coppia delle città più lontane. Sia $L$ l'insieme delle città fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   - 2.1. $\forall v \notin T$, determina il punto di inserzione $P(v)$ (ovvero l'arco $(a, b) \in T$) in modo da minimizzare $c_{av} + c_{vb} - c_{ab}$, sia $C(v)$ il costo di inserimento del nodo $v$ nel punto di inserzione trovato.
   - 2.2. Tra tutti i nodi fuori dal tour, seleziona il nodo $r$ in modo da minimizzare $C(r)$
   - 2.3. Inserisci il nodo $r$ nel punto di inserzione $P(r)$ trovato prima
   - 2.4. Rimuovi $r$ da L, quindi $L = L \setminus \{r\}$.

La complessità computazionale di questa euristica verrà discussa nella sezione nel capitolo delle implementazioni in quanto verranno presentate più versioni (con complessità differenti).

### 2.2.3 Farthest Insertion

Farthest Insertion opera in analogo a Nearest Insertion in quanto il criterio di selezione del nodo da inserire è basato sulla distanza dal tour, ma differenza di Nearest Insertion però viene scelto il nodo più lontano dal tour al posto del più vicino. Il passo di inserzione avviene invece allo stesso modo rispetto a Nearest Insertion: viene quindi scelta la posizione che minimizza il costo di inserzione. Vediamo l'algoritmo:

1. Sia $T$ il tour iniziale definito dalla coppia delle città più lontane. Sia $L$ l'insieme delle città fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   - 2.1. Sia $d_{min}(v)$ la distanza minima tra il nodo $v$ e il tour $T$, ovvero $d_{min}(v) = min\{c_{vt} | t \in T\}$. Seleziona il nodo $r \notin T$ in modo da massimizzare $d_{min}(r)$.
   - 2.2. Determina la posizione ottima di inserzione $(a, b)$ in modo da minimizzare il costo di inserimento $c_{ar} + c_{rb} - c_{ab}$, inserisci $r$ tra $(a, b)$
   - 2.3. Rimuovi $r$ da L, quindi $L = L \setminus \{r\}$.

Farthest Insertion ha complessità computazionale pari a $O(n^2)$ in modo del tutto analogo a Nearest Insertion.

### 2.2.4 Furthest Insertion

L'obbiettivo della tesi consiste nel proporre un nuovo algoritmo chiamato Furthest Insertion: allo stesso modo in cui Farthest Insertion opera in modo analogo a Nearest Insertion, Furthest Insertion opera in modo analogo a Cheapest Insertion. Il passo di selezione consiste nel selezionare il nodo che massimizza il costo di inserzione, in fase di inserimento però il nodo viene inserito in modo da minimizzare il costo di inserimento. Di seguito l'algoritmo:

1. Sia $T$ il tour iniziale definito dalla coppia delle città più lontane. Sia $L$ l'insieme delle città fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   - 2.1. $\forall v \notin T$, determina il punto di inserzione $P(v)$ (ovvero l'arco $(a, b) \in T$) in modo da minimizzare $c_{av} + c_{vb} - c_{ab}$, sia $C(v)$ il costo di inserimento del nodo $v$ nel punto di inserzione trovato.
   - 2.2. Tra tutti i nodi fuori dal tour, seleziona il nodo $r$ in modo da massimizzare $C(r)$
   - 2.3. Inserisci il nodo $r$ nel punto di inserzione $P(r)$ trovato prima
   - 2.4. Rimuovi $r$ da L, quindi $L = L \setminus \{r\}$.

La complessità computazionale di questa euristica verrà discussa nella sezione nel capitolo delle implementazioni (come per Cheapest Insertion) in quanto verranno presentate più versioni (con complessità differenti).

### 2.2.5 Random Insertion

Random Insertion seleziona casualmente il nodo da inserire nel tour, successivamente però sceglie il punto di inserimento in modo da minimizzare il costo di inserimento (esattamente come per Nearest Insertion). Di seguito l'algoritmo:

1. Sia $T$ il tour iniziale definito da una città casuale. Sia $L$ l'insieme delle città fuori dal tour, quindi: $L = \{1, 2, ..., n\} \setminus T$.
2. Finché $L \neq \emptyset$:
   2.1. Seleziona un nodo $r \in L$ casualmente.
   2.2. Seleziona l'arco $(i, j)$ in $T$ tale che $c_{ir} + c_{rj} - c_{ij}$ risulti minimo.
   2.3. Inserisci $r$ tra $i$ e $j$.
   2.4. Rimuovi $r$ da L, quindi $L = L \setminus \{r\}$.

Random Insertion ha complessità computazionale pari a $O(n^2)$ in quanto n volte viene selezionato un nodo casuale, dopodiché è necessario iterare n volte alla ricerca della posizione migliore dove inserire il nodo.

## 2.3 Ricerca Locale

Le euristiche basate su ricerca locale[9] sono euristiche che implementano un meccanismo che consente di migliorare un tour (anche parziale), il meccanismo può ad esempio scambiare due nodi nel tour (vedremo nel dettaglio una euristica in particolare), rimuovere un nodo e inserirlo in un punto del tour migliore oppure implementare meccanismi più sofisticati.

### 2.3.1 Node Insertion

Questa euristica permette di migliorare un tour rimuovendo un nodo e inserendolo nel tour nel modo migliore possibile. Di seguito la procedura:

1. Sia $T$ il tour (anche parziale).
2. Ripeti finché il tour $T$ resta invariato:
   2.1. Per ogni nodo $i = 1, 2, ..., n$: è possibile ridurre la lunghezza del tour rimuovendo il nodo i da una posizione $p_1$ e inserendo il nodo $i$ in un'altra posizione $p_2$? Se si, aggiorna $T$ spostando il nodo $i$ da $p_1$ a $p_2$.

La procedura richiede tempo $O(n^2)$ in quanto per ogni nodo (n volte) è necessario ispezionare il possibile scambio di posizione (n volte).

### 2.3.2 Edge Insertion

In modo analogo a Node Insertion, questa euristica permette di migliorare un tour rimuovendo un arco (a differenza del nodo di prima) e inserendolo nel tour nel modo migliore possibile. Di seguito la procedura:

1. Sia $T$ il tour (anche parziale).
2. Ripeti finché il tour $T$ resta invariato:
   2.1. Per ogni nodo $i = 1, 2, ..., n$, considera l'arco che collega $i$ con il nodo successivo nel tour, sia $(i, j)$ questo arco: è possibile ridurre la lunghezza del tour rimuovendo l'arco $(i, j)$ da una posizione $p_1$ e inserendo $(i, j)$ in un'altra posizione $p_2$? Se si, aggiorna $T$ spostando l'arco da $p_1$ a $p_2$.

La procedura richiede tempo $O(n^2)$ in modo del tutto analogo a Node Insertion.

### 2.3.3 Euristica 2-Opt

L'algoritmo 2-Opt[10] consente di eliminare due archi e di riconnetterli in un modo diverso, andando quindi a modificare il tour. Questa euristica deriva dall'osservazione empirica per cui due archi che si incrociano possono essere sempre riorganizzati in modo da non incrociarsi e questo produce sempre una riduzione della lunghezza totale del tour. La procedura consente di ridurre la lunghezza del tour riorganizzando una coppia di archi anche quando non si incrociano. Si nota che è sempre possibile ricombinarli in un unico modo diverso. Di seguito la procedura 2-Opt:

1. Sia $T$ il tour (anche parziale).
2. Ripeti finché il tour $T$ resta invariato:
   2.1. Per ogni nodo $i = 1, 2, ..., n$, considera tutti gli scambi 2-Opt possibili con l'arco $(i, j)$ dove $j$ è il nodo successivo ad $i$ nel tour. Se è possibile ridurre la lunghezza del tour applicando lo scambio 2-Opt, aggiorna T.

### 2.3.4 Euristica 3-Opt

L'euristica 3-Opt è un'estensione dell'euristica vista precedentemente: al posto di eliminare due archi per ricombinarli in modo diverso, in questa euristica sono 3 gli archi che l'euristica rimuove e ricombina in modo diverso. A differenza di 2-Opt sono 8 i modi diversi in cui possono essere ricombinati questi archi[1] (includendo anche la configurazione in cui gli archi restano immutati). Vediamo l'algoritmo:

1. Sia $T$ il tour.
2. Per ogni nodo $i \in V$, sia $N(i)$ un insieme di nodi.
3. Ripeti finché il tour $T$ resta invariato:
   - 3.1. Per ogni nodo $i = 1, 2, ..., n$: considera tutti gli scambi 3-Opt possibili che includa $i$ e un nodo nell'insieme $N(i)$. Se è possibile ridurre la lunghezza del tour in questo modo, aggiorna T.

La scelta di ridurre l'insieme dei nodi $V$ a $N(i)$ è necessario in quanto analizzare tutti i possibili scambi che includano 3 nodi implica un tempo $O(n^3)$. L'insieme $N(i)$ può essere generato a partire a partire dal nodo $i$ ad esempio selezionando i 10 nodi più vicini ad $i$.

### 2.3.5 Euristica Lin-Kernighan

L'algoritmo Lin-Kernighan (LK)[11] è una delle migliori euristiche per risolvere il TSP. Appartiene alla classe degli algoritmi di ricerca locale, che prendono come input un tour (ciclo hamiltoniano) e tentano di migliorarlo cercando nella sua vicinanza un tour più corto, e una volta trovato ripetono il processo da quel nuovo tour, fino a incontrare un minimo locale. L'algoritmo LK è stato sviluppato da Serge Lin e Brian Kernighan nel 1971 ed è stato dimostrato essere molto efficace per risolvere istanze di TSP di medie e grandi dimensioni. È spesso utilizzato come componente di algoritmi ibridi per il TSP, combinandolo con altre tecniche di ricerca per ottenere risultati ancora migliori. Di seguito una descrizione ad alto livello dell'algoritmo:

[1] http://tsp-basics.blogspot.com/2017/03/3-opt-move.html

1. Sia $T$ il tour iniziale, può essere generato casualmente oppure tramite altre euristiche come Nearest Neighbor. Sia $S$ la soluzione migliore fino a questo momento.
2. Ripeti finché il tour $T$ resta invariato (e quindi è stato trovato un minimo locale):
   2.1. Applica l'algoritmo k-Opt. Inizialmente $k$ deve essere un valore piccolo (2 o 3), aumenta man mano il valore $k$.
3. Una volta trovato un minimo locale, se è migliore di $S$ allora aggiorna $S$. Torna al punto 1 generando un tour diverso da quello di partenza, oppure l'algoritmo termina e S è la soluzione trovata.

L'euristica Lin-Kernighan è considerata una delle migliori euristiche per generare soluzioni ottime o quasi-ottime per il problema del TSP[12].

# Capitolo 3

# Implementazione delle euristiche

Questo capitolo descrive l'implementazione di una classe Python chiamata "TSP" tramite la quale deve essere possibile:

1. Poter generare istanze della classe TSP che rappresenti una particolare istanza del TSP, con la matrice di adiacenza e le altre strutture dati necessarie per la risoluzione del problema

2. Poter importare i dati di una determinata istanza del TSP (coordinate dei punti, matrice di adiacenza...)

3. Poter ottenere una soluzione valida per il problema tramite l'invocazione di un metodo che implementa una euristica tra quelle descritte precedentemente

4. Ottenere i dati relativi alla qualità delle soluzioni e tempi dell'euristica

Per il progetto è stata scelta la libreria TSP-LIB[1] come fonte di istanze per il TSP in quanto in letteratura è ampiamente utilizzata per testare e confrontare algoritmi e euristiche progettate per risolvere il TSP. Nel progetto, le istanze sono state tutte collocate nella directory "ALL-TSP" dove sono presenti due tipi di file: "NOME.tsp" e "NOME.opt.tour" dove la prima contiene i dati relativi all'istanza chiamata "NOME", mentre la seconda contiene la soluzione ottima sempre per quella istanza. Di seguito la struttura dei file di input:

berlin52.tsp

```
1 NAME: berlin52
2 TYPE: TSP
3 COMMENT: 52 locations in Berlin (Groetschel)
4 DIMENSION: 52
```

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

```
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
4 945.0 685.0
5 845.0 655.0
6 880.0 660.0
7 25.0 230.0
8 525.0 1000.0
9 580.0 1175.0
10 650.0 1130.0
11 1605.0 620.0
12 1220.0 580.0
13 1465.0 200.0
...
50 595.0 360.0
51 1340.0 725.0
52 1740.0 245.0
EOF
```

Dove, come specificato nella documentazione di TSP-LIB[2], gli elementi importanti sono:

1. "NAME": ovvero il nome dell'istanza TSP
2. "TYPE": ovvero il tipo di problema, nel nostro caso ci limitiamo alle istanze TSP simmetriche (in TSP LIB sono presenti anche istanze ATSP)
3. "EDGE_WEIGHT_TYPE": ovvero il tipo di distanze tra i punti, ai fini del progetto per semplicità mi limito al caso euclideo in due dimensioni
4. "NODE_COORD_SECTION": da questa sezione in poi sono presenti 3 numeri per ogni riga: numero del nodo (città), coordinata x e coordinata y: queste coordinate saranno utili per calcolare le distanze euclidee tra i nodi.

Per ogni istanza TSP, non solo è presente il relativo file descritto sopra (quindi NOME.tsp), ma anche il file contenente la soluzione ottima "NOME.opt.tour". Di seguito un esempio sempre dall'istanza berlin52:

berlin52.opt.tsp

[2]http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf

```
NAME : berlin52.opt.tour
TYPE : TOUR
DIMENSION : 52
TOUR_SECTION
1
49
32
45
19
41
8
9
10
43
...
30
2
7
42
21
17
3
18
31
22
-1
EOF
```

In questa tipologia di file è presente la soluzione in forma di lista dei nodi che formano il tour migliore. È importante notare come i nodi sono numerati da 1 compreso fino a n compreso, nella implementazione della classe (e quindi nelle euristiche) ho deciso di trasporre gli indici nel range [0, n - 1] (compresi) in modo coerente con la numerazione presente nei metodi standard delle liste presenti nel linguaggio Python (quindi ad esempio la città 3 in questo esempio corrisponderà in questa implementazione alla città 2).

## 3.1 Implementazione del costruttore della classe

Come descritto precedentemente, il costruttore della classe deve consentire di importare i dati relativi ad una istanza specifica del TSP contenuta nella cartella "TSP-ALL",

di seguito l'implementazione:

```
def __init__(self, name, ignoraOpt=False):
  self.ignoraOpt = ignoraOpt
  self.openTSP(name)
  self.name = name
  self.optimalSolution = soluzioneOttima[name]
```

Dove: `self.openTSP()` è il metodo che effettivamente importa i dati dal file relativo, inoltre sono presenti gli attributi `self.name` il quale semplicemente memorizza il nome dell'istanza TSP e l'attributo `self.optimalSolution` che memorizza il valore della soluzione ottima dal file "NOME.opt.tour".
Di seguito l'analisi del metodo `self.openTSP(name)`:

```
def openTSP(self, name):
  f = open(f"ALL-TSP/{name}.tsp")
  self.numCity = -1
  i = 0
  data = False
  self.coord = []
  for line in f:
    if "EOF" in line:
      break
    if line == "":
      continue
    if data:
      line = line.replace("  ", " ")\
               .replace("  ", " ")\
               .replace("\n", "").strip()
      if "." in line.split(" ")[1]:
          self.coord.append([int(float(line.split(" ")[1])), \
                             int(float(line.split(" ")[2]))])
      else:
          self.coord.append([int(line.split(" ")[1]), \
                             int(line.split(" ")[2])])
    if "DIMENSION" in line:
      self.numCity = int(line.split(": ")[1])
    if "NODE_COORD_SECTION" in line:
      data = True
  f.close()
```

Questa prima parte del metodo permette di aprire il file relativo all'istanza TSP e importa nella lista `self.coord` tutte le coppie $(x, y)$ che rappresentano le coordinate cartesiane dei punti.

```
def openTSP(self, name):
  ...
  self.adj = [[0 for _ in range(self.numCity)] for _ in range(self.
                                      numCity)]
```

```
  for i in range(self.numCity):
    for j in range(i):
      if i == j:
        continue
      else:
        self.adj[i][j] = self.adj[j][i] = \
           self.distance(self.coord[i], self.coord[j])
```

In questa seconda parte del metodo viene calcolata la matrice di adiacenza dell'istanza TSP e memorizzata nell'istanza `self.adj` (il metodo `self.distance` verrà descritto nella prossima sezione).

```
def openTSP(self, name):
  ...
  if not self.ignoraOpt:
      # leggo il persorso ottimale
      f = open(f"ALL-TSP/{name}.opt.tour")
      self.optTour = [-1 for _ in range(self.numCity)]
      self.tour = [-1 for _ in range(self.numCity)]
      data = False
      i = 0
      for line in f:
        if "-1" in line:
          break
        if data:
          self.optTour[i] = int(line) - 1
          i += 1
        if "TOUR_SECTION" in line:
          data = True
      f.close()
  else:
    self.optTour = [-1 for _ in range(self.numCity)]
    self.tour = [-1 for _ in range(self.numCity)]
```

L'ultima parte del metodo consente di inizializzare le liste `self.optTour` e `self.tour` dove la prima contiene la soluzione ottima contenuta nel file "NOME.opt.tour" e la seconda sarà invece utilizzata dalle euristiche costruttive per "costruire" man mano la soluzione.

## 3.2 Implementazione di metodi ausiliari per le euristiche

In questa sezione analizzo l'implementazione di metodi ausiliari e di supporto. Questi metodi fattorizzano il codice utilizzato frequentemente da diverse euristiche, migliorando la modularità, la leggibilità e la manutenibilità del codice e includono altri

metodi utili per la manipolazione di istanze del TSP.

```
1   def distance(self, a, b):
2     return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)
```

Questo metodo (come visto prima) semplicemente consente di calcolare la distanza euclidea tra due punti a e b. Come descritto precedentemente, ci limitiamo alle istanze del TSP con coordinate in due dimensioni.

```
1  def openRandomTSP(self, n):
2    self.name = "random"
3    self.numCity = n
4    self.adj = [[0 for _ in range(self.numCity)] \
5                   for _ in range(self.numCity)]
6    for i in range(self.numCity):
7      for j in range(i):
8        if i == j:
9          continue
10       else:
11         self.adj[i][j] = self.adj[j][i] = random.randint(0, 1000)
```

Questo metodo consente di generare una istanza casuale del TSP dato come parametro un numero intero n (il numero di città). Dato che il progetto si limita al TSP simmetrico, alla riga 10 si assegna lo stesso valore sia all'arco $(i, j)$ che all'arco $(j, i)$.

```
1  def calculateCost(self):
2    self.cost = 0
3    for i in range(self.numCity - 1):
4      self.cost += self.adj[self.tour[i]][self.tour[i + 1]]
5    self.cost += self.adj[self.tour[self.numCity - 1]][self.tour[0]]
```

Il metodo `calculateCost()` consente di assegnare all'attributo `self.cost` il costo del tour, che si ottiene sommando i costi degli archi $(i, j)$ per ogni coppia di archi contenuti nel tour. Essendo un ciclo hamiltoniano, alla fine del ciclo for si aggiunge anche il costo per tornare al punto di partenza, quindi il peso dell'arco che collega l'ultima città presente nel tour con la prima.

```
1  def calculateOptimalCost(self):
2    self.cost = 0
3    for i in range(self.numCity - 1):
4      self.cost += self.adj[self.optTour[i]][self.optTour[i + 1]]
5    self.cost += self.adj[self.optTour[self.numCity - 1]][self.optTour
                                                          [0]]
```

Il metodo `calculateOptimalCost()` consente, come nel caso di `calculateCost()`, di calcolare il costo del tour, con la differenza che si considera il tour ottimo (attributo `self.optTour`) al posto del tour del metodo precedente.

```
def verifyTour(self):
  for i in range(self.numCity):
    if not i in self.tour:
      print(f"ERRORE, manca {i}")
      return False
    if self.tour.count(i) != 1:
      print(f"ERRORE, il numero {i} risulta esserci \
            {self.tour.count(i)} volte")
      return False
  return True
```

Il metodo `verifyTour()` consente di testare la correttezza del tour trovato dall'euristica, è quindi pensato per essere verificato (tramite un `assert`). Il metodo si limita a verificare che ogni nodo deve essere presente nel tour esattamente una volta sola, il metodo chiaramente non garantisce la presenza di altri errori nell'implementazione dell'euristica. Questo metodo quindi consente di verificare se il tour trovato è almeno una soluzione valida per il problema.

## 3.3 Implementazione delle euristiche

In questo capitolo, descriveremo l'implementazione delle euristiche per la risoluzione del TSP, in particolare verranno implementate le euristiche costruttive presentate nei capitoli 2.1 e 2.2.

### 3.3.1 Nearest Neighbor

L'euristica Nearest Neighbor (NN) costruisce un tour partendo da una città arbitraria, dopodiché sceglie di aggiungere al tour la città più vicina all'ultima città aggiunta al tour. Di seguito viene riportato il codice per questa euristica.

```
def nearestNeighbor(self):
  self.tour = [0]
  visited = set([0])
  notVisited = set(range(1, self.numCity))
  while len(self.tour) < self.numCity:
    cost = np.inf
    j = -1
    for j2 in notVisited:
      if self.adj[self.tour[-1]][j2] < cost:
        cost = self.adj[self.tour[-1]][j2]
        j = j2
    self.tour.append(j)
    visited.add(j)
    notVisited.remove(j)
```

```
15    self.calculateCost()
```

Per semplicità ho scelto la prima città (nodo 0) come città iniziale arbitraria, dopodiché finché sono presenti nodi da aggiungere al tour, si cerca un nodo j non ancora visitato che sia il più vicino possibile all'ultimo nodo aggiunto al tour (`self.tour[-1]`), inoltre sono presenti gli insiemi (set) visited e nonVisited che tengono traccia dei nodi presenti o non presenti nel tour. Alla fine viene calcolato il costo del tour trovato tramite il metodo `self.calculateCost()`.
Dal punto di vista della complessità temporale, l'implementazione ha tempo pari a $O(n^2)$ in quanto tutte le istruzioni hanno un tempo pari a $O(1)$ tranne le istruzioni nelle righe 12, 13 e 14 per le quali il tempo medio corrisponde a $O(1)$ ma nel caso peggiore il tempo corrisponde a $O(n)$[3]. Di conseguenza, considerando i due cicli for alla riga 5 e 8, la complessità generale è pari a $O(n^2)$.

### 3.3.2 Nearest Insertion

Nelle prossime sezioni verrà discussa l'implementazione delle euristiche costruttive basate su inserzione, a partire da Nearest Insertion.

```
1  def nearestInsertion(self):
2    n = self.numCity
3    distances = np.array(self.adj)
4    path = [0, 0]
5
6    minDist = np.inf
7    for i in range(n):
8      for j in range(0, i):
9        if distances[i][j] < minDist:
10         path[0], path[1] = i, j
11         minDist = distances[i][j]
12   in_path = {path[0], path[1]}
```

Inizialmente l'euristica inizializza il tour con la coppia di città più vicine presenti nell'istanza TSP, viene quindi cercato l'arco $(i, j)$ con costo più basso e inizializzato il path (posizione 0 e 1) con questi due nodi.

```
1  def nearestInsertion(self):
2    ...
3    h = []
4    for i in set(range(n)) - in_path:
5      h.append((min(distances[path[0], i], distances[path[1], i]), i))
6    heapq.heapify(h)
```

[3]https://wiki.python.org/moin/TimeComplexity

Successivamente vengono inizializzate le distanze minime e le città più vicine per ogni città non nel percorso. Dato che nearest insertion dovrà determinare il nodo fuori dal ciclo con distanza minore rispetto ad un nodo qualunque nel ciclo, ho deciso di utilizzare uno heap tramite il modulo `heapq` con il quale sarà possibile tenere traccia dei nodi fuori dal ciclo e la loro distanza minima con il tour.

```
1  def nearestInsertion(self):
2    ...
3    while len(path) < n:
4      _, to_insert = heapq.heappop(h)
5
6      best_increase = np.inf
7      best_position = None
8      for i in range(len(path)):
9        next_i = (i + 1) % len(path)
10       increase = distances[path[i], to_insert] +\
11                  distances[to_insert, path[next_i]] - \
12                  distances[path[i], path[next_i]]
13       if increase < best_increase:
14         best_increase = increase
15         best_position = i + 1
16
17     path.insert(best_position, to_insert)
18     in_path.add(to_insert)
19
20     for i, (cost, node) in enumerate(h):
21       if node not in in_path and distances[to_insert, node] < cost:
22         h[i] = (distances[to_insert, node], node)
23     heapq.heapify(h)
24
25   self.tour = path
26   self.calculateCost()
```

Questo è il ciclo principale dell'euristica: prima di tutto tramite l'operazione `heappop` viene estratto il nodo `to_insert` come nodo candidato da inserire nel ciclo (in quanto ha distanza minima dal tour), successivamente viene effettuata una ricerca nel tour della posizione di inserimento migliore possibile. La posizione migliore deve minimizzare $c_{ir} + c_{rj} - c_{ij}$, quindi la variabile `increase` definita alla riga 10. La scelta di memorizzare nella variabile `best_position` il valore `i + 1` e non `i` è motivata dal fatto che il metodo `insert(pos, n)` (con parametro pos e n) delle liste python sposta in avanti tutti gli elementi dalla posizione pos compresa per fare spazio alla variabile da inserire (n), di conseguenza una volta trovata la posizione migliore (tra le posizioni i e i+1), sarà necessario indicare la posizione i+1 per ottenere l'inserimento del nuovo arco tra la posizione i e la posizione i + 1[4].

[4]https://docs.python.org/3/tutorial/datastructures.html

Successivamente (dalla riga 20) viene aggiornato lo heap in quanto alla riga 17 viene modificato il path mentre lo heap fa ancora riferimento al path prima dell'inserimento. L'aggiornamento procede come segue: dato che lo heap contiene, per ogni nodo fuori dal tour, la distanza con il nodo più vicino all'interno del tour (quindi per ogni nodo fuori dal tour, mantiene la distanza minima con il tour), dato che è stato aggiunto un nuovo nodo nel tour, è necessario verificare per ogni elemento nello heap se il nuovo nodo aggiunto rappresenta un caso migliore rispetto a quanto precedentemente memorizzato nello heap, per questo motivo alla riga 21, nel caso in cui la distanza tra il nodo appena aggiunto e il nodo fuori dal path `node` è migliore di quanto precedentemente memorizzato nello heap (`cost`), allora è necessario aggiornare quel valore nello heap.
Successivamente è necessario effettuare l'operazione `heapq.heapify(h)` in quanto l'aggiornamento potrebbe aver modificato la struttura dello heap. Alla fine dell'implementazione di Nearest Insertion è presente (come prima) l'aggiornamento della variabile self.tour (che memorizza il tour finale) e viene eseguito il metodo `self.calculateCost()` come per le altre euristiche per ottenere il costo del percorso nella variabile `self.cost` come visto precedentemente nel capitolo dei metodi ausiliari.
La complessità temporale dell'implementazione è $O(n^2)$ dato che sono presenti solo due cicli for che iterano al massimo n volte, tutte le istruzioni nel ciclo interno hanno complessità pari a $O(1)$ e le istruzioni nel ciclo esterno hanno al massimo complessità $O(n)$.

### 3.3.3 Cheapest Insertion: versione 1

Nelle prossime sezioni discutiamo l'implementazione di Cheapest Insertion che, come abbiamo visto nel secondo capitolo, seleziona i nodi in modo da minimizzare il costo di inserimento (al posto di selezionare i nodi più vicini al tour come visto con nearest insertion). Sono state implementate 3 versioni di questa euristica, iniziamo dalla versione peggiore dal punto di vista della complessità computazionale.

```
1  def cheapestInsertionOn3(self):
2    n = self.numCity
3    adj = np.array(self.adj)
4    path = [0, 0]
5
6    minDist = np.inf
7    for i in range(n):
8      for j in range(0, i):
9        if adj[i][j] < minDist:
10         path[0], path[1] = i, j
11         minDist = adj[i][j]
12   in_path = {path[0], path[1]}
```

Per prima cosa il path viene inizializzato con la coppia di città più vicine presenti nell'istanza TSP.

```
1  def cheapestInsertionOn3(self):
2    ...
3    while len(path) < n:
4      ln = []
5      for v in set(range(n)) - in_path:
6        record = [np.inf]
7        for i in range(len(path)):
8          l = path[i]
9          r = path[(i + 1) % len(path)]
10         cost = adj[l][v] + adj[v][r] - adj[l][r]
11         if cost < record[0]:
12           record = [cost, v, l, r]
13       ln.append(record)
14     [_, to_ins, l, r] = min(ln)
15     path.insert(path.index(r), to_ins)
16     in_path.add(to_ins)
17   self.tour = path
18   self.calculateCost()
```

Dopodiché inizia il ciclo principale dell'euristica: finché il path non include tutti i nodi, si determina per ogni nodo v fuori dal tour il punto di inserimento ottimo $(i, j)$ fuori dal tour in modo da minimizzare il costo di inserimento, quindi si crea per ogni nodo v un record `[cost, v, l, r]` che contiene le informazioni: costo di inserimento, nodo da inserire `v` tra i due nodi `l` ed `r`. Successivamente dalla lista `l` di record viene selezionato il record con costo minore e si procede all'inserimento.
Questa implementazione di Cheapest Insertion risulta essere molto semplice, ma è inefficiente dal punto di vista della complessità temporale. Infatti sono presenti 3 cicli for che iterano al massimo n volte (riga 3, 5 e 7), alla riga 13 il metodo append alla lista `ln` (lista dei nodi) che ha complessità pari a $O(1)$ e le istruzioni `path.index(r)` e `path.insert()` che hanno complessità pari a $O(n)$. Di conseguenza la complessità temporale è $O(n^3)$.
In particolare l'inefficienza di questa implementazione, consiste nel fatto che ad ogni iterazione vengono ricalcolati i costi di inserimento ottimi per gli stessi nodi, infatti ad ogni iterazione solo un nodo viene inserito nel path, per tutti gli altri nodi fuori dal tour (che sono rimasti fuori dal tour) si ricalcolano esattamente gli stessi costi, con la differenza che un arco è stato rimosso (il punto di inserzione) e sono comparsi due nuovi archi. Per questo motivo nella prossima versione vedremo come memorizzare (e aggiornare) tutte queste informazioni al posto di ricalcolarle ogni volta.

### 3.3.4 Cheapest Insertion: versione 2

In questa versione di Cheapest Insertion (a differenza della versione precedente) si vuole utilizzare una struttura dati che possa mantenere, per ogni nodo fuori dal tour, il costo e il punto di inserzione ottimi. Per fare questo si può operare in modo analogo a Nearest Insertion, ovvero con uno heap: lo heap andrà a conservare le informazioni dette prima e ci consentirà di ottenere il record con costo minore in modo efficiente, con la differenza, però, che l'aggiornamento dello heap è indubbiamento più complesso: nel caso di Nearest Insertion bastava confrontare il vecchio nodo più vicino con il nuovo nodo aggiunto, in questo caso la struttura del path cambia con un inserimento e quindi, nel caso peggiore, sarà necessario ricalcolare completamente il punto di inserimento ottimo. Vediamo l'implementazione:

```
def cheapestInsertion(self):
  n = self.numCity
  adj = np.array(self.adj)
  path = [0, 0]

  minDist = np.inf
  for i in range(n):
      for j in range(0, i):
          if adj[i][j] < minDist:
              path[0], path[1] = i, j
              minDist = adj[i][j]
  in_path = {path[0], path[1]}
```

La parte iniziale dell'implementazione è identica a Nearest Insertion: viene inizializzato il tour con la coppia di nodi più vicini presenti nell'istanza.

```
def cheapestInsertion(self):
  ...
  h = []
  for i in set(range(n)) - in_path:
    cost = adj[path[0]][i] + adj[i][path[1]] - adj[path[0]][path[1]]
    h.append((cost, i, path[0], path[1]))
  heapq.heapify(h)
```

Come descritto prima, in questa versione viene inizializzato uno heap in modo analogo a Nearest Insertion, ma con una differenza: la chiave associata ad ogni nodo non è più la distanza minima con il tour ma il costo di inserimento (se inserito nel modo migliore possibile). In questo caso il costo di inserimento è semplice da ottenere in quanto nel tour sono presenti solo 2 nodi: dati gli unici due nodi presenti nel tour $a$ e $b$, il costo di inserimento del nodo $i$ sarà $c_{ai} + c_{ib} - c_{ab}$. Lo heap userà come valore chiave il costo (prima variabile della tupla) permettendo quindi di estrarre il nodo `i` con costo minore. Nello heap vengono memorizzate anche i due nodi tra i quali

dovrebbe avvenire l'inserimento (`path[0]` e `path[1]`) in quanto sono informazioni che saranno utili nella seconda parte dell'implementazione dell'euristica.

```
def cheapestInsertion(self):
  ...
  while len(path) < n:
    (_, to_ins, _, r) = heapq.heappop(h)
    best_pos = path.index(r)
    path.insert(best_pos, to_ins)
    in_path.add(to_ins)
    ...
```

Questo è il ciclo for (while) principale dell'euristica: viene selezionato dallo heap il nodo `to_ins` e il punto di inserzione `r`. Abbiamo visto come nello heap sono presenti le seguenti informazioni: `(costo, nodo, l, r)` dove `cost` indica il costo di inserzione del nodo `nodo` se inserito nel modo migliore possibile e `l` e `r` sono i due nodi (presenti nel tour) tra i quali è necessario effettuare l'inserzione. Come visto nell'implementazione di nearest insertion, il metodo `l.insert(pos, element)` sposta avanti tutti gli elementi dalla posizione `pos` per fare spazio all'elemento da inserire, per questo motivo dallo heap è necessario ottenere solo il nodo `r` (e non `l`) in quanto basta sapere la posizione del nodo a destra (rispetto al punto di inserzione) da poter passare come argomento al metodo `insert()`.

```
def cheapestInsertion(self):
  ...
  while len(path) < n:
    ...
    for i, (cost, node, nodeLeft, nodeRight) in enumerate(h):
      # costo di inserimento di node tra
      # (path[(best_pos - 1) % len(path)], to_ins)
      left_cost = adj[path[(best_pos - 1) % len(path)]][node] + \
                  adj[node][to_ins] - \
                  adj[path[(best_pos - 1) % len(path)]][to_ins]
      # costo di inserimento di node tra
      # (to_ins, path[(best_pos + 1) % len(path)])
      right_cost = adj[to_ins][node] + \
                   adj[node][path[(best_pos + 1) % (len(path))]] - \
                   adj[to_ins][path[(best_pos + 1) % (len(path))]]

      # se ho inserito to_ins proprio tra nodeLeft e nodeRight:
      # ricalcolo completamente il costo migliore
      if node not in in_path and nodeLeft == path[(best_pos - 1) % \
        len(path)] and nodeRight == path[(best_pos+1) % len(path)]:
        best_cost = np.inf
        posL, posR = -1, -1
        for i2 in range(len(path)):
          next_i = (i2 + 1) % len(path)
          insertion_cost = adj[path[i2]][node] + \
```

```
26                          adj[node][path[next_i]] - \
27                          adj[path[i2]][path[next_i]]
28          if best_cost > insertion_cost:
29            best_cost, posL, posR = insertion_cost, i2, next_i
30        h[i] = (best_cost, node, path[posL], path[posR])
31        (cost, node, nodeLeft, nodeRight) = h[i]
32      # path[best_pos - 1] -- node -- to_ins
33      # se il nuovo arco a sinistra
34      # permette un inserimento migliore di cost, quindi:
35      if node not in in_path and left_cost < cost:
36        new_cost = left_cost
37        h[i] = (new_cost, node, \
38                path[(best_pos - 1) % len(path)], to_ins)
39        (cost, node, nodeLeft, nodeRight) = h[i]
40
41      # to_ins -- node -- path[best_pos + 1]
42      # se il nuovo arco a destra
43      # permette un inserimento migliore di cost, quindi:
44      if node not in in_path and right_cost < cost:
45        new_cost = right_cost
46        h[i] = (new_cost, node, to_ins, \
47                path[(best_pos + 1) % (len(path))])
48    heapq.heapify(h)
49  self.tour = path
50  self.calculateCost()
```

Dalla riga 19 in poi avviene l'aggiornamento dello heap (in modo analogo a quanto visto con Nearest Insertion): dato che l'inserimento del nuovo nodo nel tour ha generato una modifica del tour (e quindi la cancellazione di un arco e l'inserimento di nuovi due archi) è necessario verificare che i valori presenti nello heap siano ancora rappresentativi del nuovo tour e, in caso contrario, aggiornare lo heap. Per ogni elemento nello heap, dati i valori `node` (nodo da inserire), `nodeLeft` e `nodeRight` (nodi tra i quali andava inserito `node`), i casi da considerare sono:

1. Nel caso in cui `nodeLeft` e `nodeRight` corrispondono esattamente ai due nodi tra i quali è stato inserito il nodo `to_ins` (nella prima parte del ciclo principale dell'algoritmo), allora è necessario ricalcolare per il nodo `node` il costo e il punto di inserimento migliore, in quanto il vecchio arco è stato cancellato. (Riga 17-31).

2. In ogni altro caso il vecchio arco (tra `nodeLeft` e `nodeRight`) è rimasto intatto. A questo punto è necessario semplicemente verificare che i due nuovi archi generati dopo l'inserimento (quindi gli archi `path[best_pos - 1]` - `node` e `node` - `path[best_pos + 1]`) possano essere un caso di inserimento con costo minore rispetto a quello presente nello heap (riga 32-47).

Inoltre è importante notare come l'accesso agli elementi del tour avviene `% len(tour)` in quanto il tour è una lista circolare. Successivamente (come per le altre euristiche), è necessario ripristinare la struttura dello heap con il metodo `heapq.heapify(h)` e alla fine memorizzare il path finale nell'attributo `self.tour` e calcolare il costo della soluzione trovata.
Questa seconda versione di cheapestInsertion ha complessità temporale $O(n^3)$ come per la prima versione, però con una differenza importante: il terzo ciclo interno (riga 23) viene eseguito solo se è vera la condizione precedente (riga 19), quindi il ricalcolo del punto di inserzione ottimo per il nodo avviene solo se necessario (quando si perde il punto di inserzione ottimo precedente) mentre nella prima versione di questa euristica si ricalcola il punto di inserzione ottimo per ogni nodo ad ogni iterazione. Vedremo nel prossimo capitolo come le prestazioni (tempi) di questa seconda implementazione è sempre migliore della prima.
Una successiva possibile ottimizzazione può derivare dalla seguente osservazione: una volta che il punto di inserzione ottimo si perde perché è stato già inserito un nodo proprio in quel punto, può essere ragionevole pensare che il nuovo punto di inserimento ottimo sia vicino a quello appena perso. Per verificare (o smentire) questa ipotesi si può molto semplicemente contare il numero di volte che il nuovo punto di inserzione ottimo è "vicino" al precedente punto di inserzione e con "vicino" si può intendere ad esempio che sia uno dei due nuovi archi creati dall'inserzione precedente. Quindi dal punto di vista dell'implementazione:

```
1  def cheapestInsertion(self):
2    ...
3    conteggio = 0
4    tot = 0
5    while len(path) < n:
6      ...
```

È possibile inizializzare due variabili conteggio e tot prima del ciclo principale dell'euristica.

```
1  def cheapestInsertion(self):
2    ...
3    while len(path) < n:
4      ...
5      for i, (cost, node, nodeLeft, nodeRight) in enumerate(h):
6        ...
7        if node not in in_path and \
8           nodeLeft == path[(best_pos - 1) % len(path)] and \
9           nodeRight == path[(best_pos + 1) % len(path)]:
10         ...
11         for i2 in range(len(path)):
12           next_i = (i2 + 1) % len(path)
13           insertion_cost = adj[path[i2]][node] + \
```

```
14                              adj[node][path[next_i]] - \
15                              adj[path[i2]][path[next_i]]
16            if best_cost > insertion_cost:
17              best_cost, posL, posR = insertion_cost, i2, next_i
18          h[i] = (best_cost, node, path[posL], path[posR])
19          (cost, node, nodeLeft, nodeRight) = h[i]
20          if abs(posR - best_pos) <= 2:
21            conteggio += 1
22          tot += 1
23    ...
24    print(f"{self.name} -> {conteggio / tot}")
```

Le aggiunte sono:

1. Riga 20: si verifica che la nuova posizione `posR` sia "vicina" alla posizione di inserzione precedente, ovvero riga `best_pos`, nel nostro caso che sia nell'intorno del punto `best_pos` con raggio 2.

2. Riga 24: si stampa a schermo la percentuale trovata.

I risultati trovati sono (scegliendo alcune istanze dalla libreria TSP-LIB):

```
berlin52 -> 0.9963898916967509
kroB200 -> 0.9996959562176954
rat575 -> 0.9990732159406858
vm1084 -> 0.999213293590091
u1817 -> 0.9980191593934437
```

Quindi è possibile affermare (con una probabilità molto alta) che l'arco con costo di inserzione minimo si trova nelle vicinanze del precedente arco con costo di inserzione minimo. Dopo questa osservazione è possibile applicare questa semplice modifica all'implementazione:

```
1  def cheapestInsertion(self, k):
2    ...
3    while len(path) < n:
4      ...
5      for i, (cost, node, nodeLeft, nodeRight) in enumerate(h):
6        ...
7        if node not in in_path and \
8           nodeLeft == path[(best_pos - 1) % len(path)] and \
9           nodeRight == path[(best_pos + 1) % len(path)]:
10         best_cost = np.inf
11         posL, posR = -1, -1
12         #for i2 in range(len(path)):
13         #  next_i = (i2 + 1) % len(path)
14         for f in range(best_pos - k, best_pos + k):
```

```
15            i2 = f % len(path)
16            next_i = (i2 + 1) % len(path)
17            insertion_cost = adj[path[i2]][node] + \
18                             adj[node][path[next_i]] - \
19                             adj[path[i2]][path[next_i]]
20            if best_cost > insertion_cost:
21              best_cost, posL, posR = insertion_cost, i2, next_i
22          h[i] = (best_cost, node, path[posL], path[posR])
23          (cost, node, nodeLeft, nodeRight) = h[i]
24          ...
```

La modifica (riga 14, alla riga 12 è presente la versione precedente) consiste nel ridurre lo spazio di ricerca da tutto il path al solo intorno (di raggio k) attorno al punto `best_pos`, k è un parametro (riga 1) che può assumere valore 2, 3 o anche 4 ad esempio. Questa modifica rappresenta una ottimizzazione rilevante soprattutto nel caso di istanze molto grandi (con migliaia o decine di migliaia di città): in quel caso al posto di iterare su tutto il path, si itera solo per le 3-4 città attorno al punto di inserzione ottimo precedente, tuttavia è importante sottolineare che la complessità di questa implementazione è $O(n^2)$ ma non garantisce la scelta del punto ottimo di inserzione: infatti è sempre presente una probabilità $\leq 1$ nella quale il punto di inserzione ottimo per un determinato nodo sia fuori dallo spazio di ricerca. Per questo motivo questa versione "approssimata" di Cheapest Insertion opera in modo molto simile alla versione 2 (in termini di qualità delle soluzioni) ma non allo stesso modo.

### 3.3.5 Cheapest Insertion: versione 3

In questa ultima versione di Cheapest Insertion si riesce (finalmente) a ridurre la complessità computazionale rispetto alle due versioni precedenti. Nella versione 2 abbiamo visto una implementazione con un heap che tiene traccia per ogni nodo del punto di inserzione ottimo, inoltre abbiamo visto come questa informazione non sia sufficiente nel caso in cui il punto di inserzione ottimo si perda dopo l'inserimento di un altro nodo. L'idea per questa ultima implementazione quindi sarebbe quella di tenere traccia per ogni nodo anche di altre posizioni di inserzione sub-ottime che possono subentrare nel caso appunto la posizione ottima si perda. Per implementare questa idea quindi è necessario avere più heap (uno associato ad ogni nodo fuori dallo heap) ed è necessario avere una struttura dati che consenta di associare, ad ogni arco presente nel tour, tutti i record che fanno riferimento a quell'arco in modo da poterli eliminare nel caso in cui, appunto, l'arco scompare dal path. Per fare questo ho deciso di utilizzare un dizionario che associa coppie di interi (archi) a liste di record. Vediamo l'implementazione:

```
1 def cheapestInsertionOttimizzato(self, m):
2   n = self.numCity
```

```
3   adj = np.array(self.adj)
4   path = [0, 0]
5
6   minDist = np.inf
7   for i in range(n):
8     for j in range(0, i):
9       if adj[i][j] < minDist:
10        path[0], path[1] = i, j
11        minDist = adj[i][j]
12  in_path = {path[0], path[1]}
```

Come per le euristiche precedenti, inizialmente si seleziona la coppia delle città più vicine per inizializzare il tour. Il parametro `m` aggiunto (riga 1) verrà descritto nelle prossime pagine.

```
1  def cheapestInsertionOttimizzato(self, m):
2    ...
3    # heap principale
4    h = []
5    # dizionario che associa ogni arco (i, j) nel tour
6    # alla lista dei record che puntano all'arco (i, j)
7    d = dict()
8    d[(path[0], path[1])] = []
9    d[(path[1], path[0])] = []
10   for i in set(range(n)) - in_path:
11     cost = adj[path[0]][i] + adj[i][path[1]] - adj[path[0]][path[1]]
12     h_i = [cost, i, path[0], path[1]]
13     h_i2 = [cost, i, path[1], path[0]]
14     h.append([h_i, h_i2])
15     d[(path[0], path[1])].append(h_i)
16     d[(path[1], path[0])].append(h_i2)
17   heapq.heapify(h)
```

Successivamente si inizializzano le due strutture dati principali dell'euristica: lo heap principale (che andrà a contenere i vari heap dei nodi fuori dal path) e il dizionario descritto precedentemente che viene inizializzato con la coppia di città città più vicine. Dopodiché per ogni nodo fuori dal path, si inizializza lo heap specifico di ogni nodo (riga 12-13) che contiene inizialmente gli unici due inserimenti possibili: tra l'arco `path[0] - path[1]` oppure tra l'arco `path[1] - path[0]`, inoltre si aggiungono i nuovi record alla lista degli archi `path[0] - path[1]` e `path[1] - path[0]` del dizionario. Alla fine, si procede con la costruzione dello heap tramite istruzione `heapq.heapify(h)`. Importante specificare che in questa fase non è necessaria l'istruzione `heapq.heapify(h_i)` per ogni piccolo heap dei nodi fuori dal tour, in quanto per ora sono solo presenti due record con lo stesso costo.

```
1  def cheapestInsertionOttimizzato(self, m):
2    ...
```

```
3   while len(path) < n:
4     h_i = heapq.heappop(h)
5     (_, to_ins, l, r) = heapq.heappop(h_i)
6     best_pos = path.index(r)
7     path.insert(best_pos, to_ins)
8     in_path.add(to_ins)
9
10    for p in d[(l, r)]:
11      p[0] = np.inf
```

Dalla riga 3 inizia il ciclo principale dell'euristica: si seleziona dallo heap principale `h` lo heap specifico di un determinato nodo fuori dal tour `h_i`, dallo heap piccolo si estrae il record `(costo, to_ins, l, r)` nel quale sono presenti le informazioni utili per l'inserimento: il nodo da inserire `to_ins` e i due nodi `l` ed `r` tra i quali va inserito il nodo, mentre il primo elemento (il costo) può essere scartato e quindi poi si procede all'inserimento del nodo tra `l` ed `r`. Importante notare come il modulo `heapq` effettua dei confronti tra record andando a usare come chiave l'elemento in prima posizione nel record, nel nostro caso il costo. Inoltre nel caso dello heap principale `h`, l'elemento chiave diventa il costo presente nel primo record di ogni lista, quindi le liste di record vengono confrontate considerando solo il costo del primo record di ogni lista. A questo punto è necessario cancellare tutti i record che puntano all'arco `(l - r)`: per ottenere questo risultato ho deciso di assegnare costo infinito ai record associati a quell'arco: in questo modo si evita che possano essere estratti (alla riga 5) record con archi che non sono più presenti nel path.

```
1  def cheapestInsertionOttimizzato(self, m):
2    ...
3    while len(path) < n:
4      ...
5      for hp in h:
6        node = hp[0][1]
7
8        # sx: path[(best_pos - 1) % len(path)]
9        # node
10       # dx: path[best_pos]
11       sx = path[(best_pos - 1) % len(path)]
12       dx = path[best_pos]
13       newCost = adj[sx][node] + \
14                 adj[node][dx] - \
15                 adj[sx][dx]
16       l = [newCost, node, sx, dx]
17       if (sx, dx) not in d:
18           d[(sx, dx)] = [l]
19       else:
20         d[(sx, dx)].append(l)
21       hp.append(l)
```

```
22
23        # sx: path[best_pos]
24        # node
25        # dx: path[(best_pos + 1) % len(path)]
26        sx = path[best_pos]
27        dx = path[(best_pos + 1) % len(path)]
28        newCost = adj[sx][node] + \
29                  adj[node][dx] - \
30                  adj[sx][dx]
31        l = [newCost, node, sx, dx]
32        if (sx, dx) not in d:
33          d[(sx, dx)] = [l]
34        else:
35          d[(sx, dx)].append(l)
36        hp.append(l)
```

Una volta inserito il nodo nel tour, si procede con l'aggiornamento del dizionario e degli heap associati ai nodi fuori dal tour, ovvero: per ogni heap "piccolo" presente nello heap principale, si procede aggiungendo (allo heap piccolo) i due nuovi record relativi ai due nuovi archi creati: nelle righe 8-21 si procede con l'arco a sinistra e con le righe 23-36 con l'arco a destra. In più si aggiornano i dizionari con i nuovi archi creati aggiungendo i riferimenti ai nuovi record (righe 20 e 35).

```
1  def cheapestInsertionOttimizzato(self, m):
2    ...
3    while len(path) < n:
4      ...
5      for hp in h:
6        ...
7        i = 0
8        while i < len(hp):
9          if hp[i][0] == np.inf:
10           del hp[i]
11         else:
12           i += 1
13       heapq.heapify(hp)
14     for i in range(len(h)):
15       h[i] = h[i][:m]
16     heapq.heapify(h)
17   self.tour = path
18   self.calculateCost()
```

In conclusione si eliminano dagli heap piccoli i record con costo infinito tramite istruzione `del hp[i]` e si procede con il ripristino della struttura dello heap piccolo.
Importante notare l'istruzione alla riga 15: concluso l'aggiornamento di ogni heap, si procede limitando la dimensione degli heap piccoli a `m`: questo parametro è necessario perché con `m = n` gli heap piccoli `hp` avrebbero nel caso peggiore una dimensione pari

a `n` e quindi si otterrebbe una complessità temporale pari a $O(n^3)$ (cicli alla riga 3, 5 e 8 o 14 tutti con un numero massimo di iterazioni pari a `n`). Per questo motivo è necessario tenere il parametro m il più basso possibile in modo da limitare la complessità temporale, ma la complessità spaziale: infatti sarebbe necessario tenere in memoria $O(n^2)$ record, mentre limitando il parametro si avrebbe complessità spaziale pari a $O(nm)$.
Questa implementazione riduce la complessità temporale delle versioni precedenti da $O(n^3)$ a $O(n^2 * m)$ pur garantendo che, ad ogni iterazione, si selezioni il nodo che minimizza il costo di inserimento e si selezioni il punto di inserimento ottimo (nella versione approssimata vista in precedenza questa garanzia non c'era).
Nel prossimo capitolo analizzeremo le prestazioni di queste implementazioni, in particolare vedremo quest'ultima versione come si comporta al variare del parametro m.

### 3.3.6 Farthest Insertion

In questa sezione discutiamo l'implementazione di Farthest Insertion. L'implementazione di questa euristica condivide molto del codice già discusso nella implementazione di Nearest Insertion, sono però presenti delle piccole modifiche che consentono di selezionare il nodo più lontano al posto del nodo più vicino (al tour), di seguito l'implementazione:

```
1   def farthestInsertion(self):
2     n = self.numCity
3     distances = np.array(self.adj)
4     path = [0, 0]
5
6     maxDist = 0
7       for i in range(n):
8         for j in range(0, i):
9           if distances[i][j] > maxDist:
10            path[0], path[1] = i, j
11            maxDist = distances[i][j]
12      in_path = {path[0], path[1]}
```

L'euristica inizializza il tour con la coppia di città più lontane (a differenza della coppia di città più vicine nel caso di Nearest Insertion) presenti nell'istanza TSP, viene quindi cercato l'arco $(i, j)$ con costo più alto e inizializzato il path (posizione 0 e 1) con questi due nodi.

```
1   def farthestInsertion(self):
2     ...
3     h = []
4     for i in set(range(n)) - in_path:
```

```
5       h.append((-min(distances[path[0], i], distances[path[1], i]),
                                                  i))
6     heapq.heapify(h)
```

In modo del tutto analogo a nearest insertion, viene inizializzato uno heap il quale conterrà il costo di inserimento se il nodo viene inserito nel modo migliore possibile. Dato che il metodo heappop all'interno del modulo `heapq` restituisce l'elemento più piccolo (quindi il nostro heap h opera come un min-heap) e dato che per l'euristica farthest insertion è necessario prelevare il nodo con il costo maggiore (e non minore), ogni distanza inserita nello heap (primo elemento di ogni tupla) verrà inserita con valore negativo: con questa semplice modifica il min-heap si comporterà come un max-heap e restituirà (tramite il metodo `heapq.heappop(h)`) il nodo con distanza maggiore dal tour. Per questo motivo il codice è identico a nearest insertion, con la differenza che alla riga 5 è presente un segno meno all'interno della tupla $(distance, node)$.

```
1   def farthestInsertion(self):
2     ...
3     while len(path) < n:
4       _, to_insert = heapq.heappop(h)
5
6       best_increase = np.inf
7       best_position = None
8       for i in range(len(path)):
9         next_i = (i + 1) % len(path)
10        increase = distances[path[i], to_insert] +\
11                   distances[to_insert, path[next_i]] - \
12                   distances[path[i], path[next_i]]
13        if increase < best_increase:
14          best_increase = increase
15          best_position = i + 1
16
17      path.insert(best_position, to_insert)
18      in_path.add(to_insert)
19
20      for i, (cost, node) in enumerate(h):
21        cost *= -1
22        if node not in in_path and \
23           distances[to_insert, node] < cost:
24          h[i] = (-distances[to_insert, node], node)
25      heapq.heapify(h)
26
27    self.tour = path
28    self.calculateCost()
```

Questo è il ciclo principale dell'euristica: il codice è totalmente identico a quanto visto con nearest insertion, con le seguenti piccole (ma importanti) differenze:

1. Alla riga 21 il costo viene moltiplicato per -1 in modo da riottenere una distanza positiva: in questo modo la ricerca di una nuova distanza minima può avvenire tramite la condizione `distances[to_insert, node] < cost`

2. Alla riga 24 la nuova distanza trovata (migliore, ovvero minore di cost) deve essere riconvertita come una distanza negativa in modo da mantenere lo heap funzionante come max-heap.

In conclusione è presente l'istruzione `heapq.heapify(h)` che consente di ripristinare la struttura dello heap e le ultime istruzioni come per le altre euristiche (memorizzazione del tour trovato e calcolo del costo).
La complessità temporale di questa implementazione è la stessa di Nearest Insertion, quindi $O(n^2)$.

### 3.3.7 Furthest Insertion: versione 1

Nelle seguenti sezioni discutiamo l'implementazione di Furthest Insertion ovvero la nuova euristica proposta in questa tesi. Nella precedente sezione abbiamo visto come farthest insertion condivide molto del codice di nearest insertion con delle piccole modifiche che consentono di selezionare il nodo più lontano al posto del nodo più vicino. Allo stesso modo furthest insertion condivide molto del codice di cheapest insertion, verranno di seguito presentate le piccole (ma importanti) modifiche rispetto alle tre versioni di Cheapest Insertion che consentono di selezionare il nodo che, se inserito in modo da minimizzare il costo di inserzione, ha costo di inserimento massimo. Come nel caso di Cheapest Insertion, vediamo le tre versioni in ordine dalla peggiore alla migliore dal punto di vista della complessità computazionale:

```
1  def furthestInsertionOn3(self):
2    n = self.numCity
3    adj = np.array(self.adj)
4    path = [0, 0]
5
6    maxDist = 0
7    for i in range(n):
8      for j in range(0, i):
9        if adj[i][j] > maxDist:
10           path[0], path[1] = i, j
11           maxDist = adj[i][j]
12   in_path = {path[0], path[1]}
```

Come per visto nell'implementazione di Farthest Insertion, si seleziona la coppia di nodi più lontana per inizializzare il path.

```
1  def furthestInsertionOn3(self):
2    ...
```

```
3   while len(path) < n:
4     ln = []
5     for v in set(range(n)) - in_path:
6       record = [np.inf]
7       for i in range(len(path)):
8         l = path[i]
9         r = path[(i + 1) % len(path)]
10        cost = adj[l][v] + adj[v][r] - adj[l][r]
11        if cost < record[0]:
12          record = [cost, v, l, r]
13      ln.append(record)
14    [_, to_ins, l, r] = max(ln)
15    path.insert(path.index(r), to_ins)
16    in_path.add(to_ins)
17  self.tour = path
18  self.calculateCost()
```

Successivamente (ciclo principale del'euristica) in modo analogo alla versione di Cheapest Insertion versione 1, si genera una lista di record che contiene, per ogni nodo `v` fuori dal tour, il punto di inserzione ottimo `(l, r)` in modo da minimizzare l'inserimento, però è presente una differenza importante: alla riga 14 si procede selezionando il massimo (`max(path.index(r), to_ins)`) ovvero la tupla che massimizza il miglior costo di inserimento. Successivamente si procede all'inserimento come per Cheapest Insertion.
Come descritto nei capitoli precedenti, la complessità temporale di questo algoritmo è $O(n^3)$ e non è presente nessun tipo di ottimizzazione: ad ogni iterazione è necessario ricalcolare completamente il punto di inserzione ottimo.

### 3.3.8 Furthest Insertion: versione 2

Come nel caso della versione 2 di Cheapest Insertion, si può evitare di ricalcolare totalmente il punto di inserzione ottimo con uno heap, con una differenza importante: lo heap deve operare come un max-heap e non come un min-heap, ovvero deve selezionare ad ogni iterazione il nodo con costo di inserzione massimo (se inserito nel modo migliore possibile). Vediamo l'implementazione:

```
1  def furthestInsertion(self):
2    n = self.numCity
3    adj = np.array(self.adj)
4    path = [0, 0]
5
6    maxDist = 0
7    for i in range(n):
8        for j in range(0, i):
9            if adj[i][j] > maxDist:
10               path[0], path[1] = i, j
```

```
11                maxDist = adj[i][j]
12    in_path = {path[0], path[1]}
```

Inizialmente l'euristica determina la coppia di nodi più lontani (come visto prima).

```
1  def furthestInsertion(self):
2    ...
3    h = []
4    for i in set(range(n)) - in_path:
5      cost = adj[path[0]][i] + \
6             adj[i][path[1]] - \
7             adj[path[0]][path[1]]
8      h.append((-cost, i, path[0], path[1]))
9    heapq.heapify(h)
```

Esattamente come nel caso di farthest insertion, viene inizializzato lo heap con costi negativi in modo da poter successivamente selezionare le città con costo di inserimento più alto, per questo motivo il costo nella tupla aggiunta allo heap presenta un segno negativo (riga 8).

```
1  def furthestInsertion(self):
2    ...
3    while len(path) < n:
4      # Ottieni dallo heap la citta'
5      # che minimizza il minor costo di inserimento
6      (_, to_ins, _, r) = heapq.heappop(h)
7      best_pos = path.index(r)
8      path.insert(best_pos, to_ins)
9      in_path.add(to_ins)
10     ...
```

Questo è il ciclo principale dell'euristica: viene selezionato dallo heap il nodo `to_ins` ovvero il nodo che massimizza il costo di inserimento se inserito nel modo migliore possibile; successivamente (dalla riga 6 alla 17) viene effettuata una ricerca del punto di inserimento nel path, in modo da minimizzare il costo di inserimento (in modo identico a cheapest insertion).

```
1  def furthestInsertion(self):
2    ...
3    while len(path) < n:
4      ...
5      for i, (cost, node, nodeLeft, nodeRight) in enumerate(h):
6        cost *= -1
7        # costo di inserimento di node tra
8        # (path[(best_pos - 1) % len(path)], to_ins)
9        left_cost = adj[path[(best_pos - 1) % len(path)]][node] + \
10                   adj[node][to_ins] - \
11                   adj[path[(best_pos - 1) % len(path)]][to_ins]
12       # costo di inserimento di node tra
```

```
13        # (to_ins, path[(best_pos + 1) % len(path)])
14        right_cost = adj[to_ins][node] + \
15                     adj[node][path[(best_pos + 1) % (len(path))]] - \
16                     adj[to_ins][path[(best_pos + 1) % (len(path))]]
17
18        # se ho inserito to_ins proprio tra nodeLeft e nodeRight:
19        # ricalcolo completamente il costo migliore
20        if node not in in_path and \
21           nodeLeft == path[(best_pos - 1) % len(path)] and \
22           nodeRight == path[(best_pos + 1) % len(path)]:
23          best_cost = np.inf
24          posL, posR = -1, -1
25          for i2 in range(len(path)):
26            next_i = (i2 + 1) % len(path)
27            insertion_cost = adj[path[i2]][node] + \
28                             adj[node][path[next_i]] - \
29                             adj[path[i2]][path[next_i]]
30            if best_cost > insertion_cost:
31              best_cost, posL, posR = insertion_cost, i2, next_i
32          h[i] = (-best_cost, node, path[posL], path[posR])
33          (cost, node, nodeLeft, nodeRight) = h[i]
34          cost *= -1
35        # path[best_pos - 1] -- node -- to_ins
36        # se il nuovo arco a sinistra
37        # permette un inserimento migliore di cost, quindi:
38
39        if node not in in_path and left_cost < cost:
40          new_cost = left_cost
41          h[i] = (-new_cost, node, \
42                  path[(best_pos - 1) % len(path)], to_ins)
43          (cost, node, nodeLeft, nodeRight) = h[i]
44          cost *= -1
45
46        # to_ins -- node -- path[best_pos + 1]
47        # se il nuovo arco a destra
48        # permette un inserimento migliore di cost, quindi:
49        if node not in in_path and right_cost < cost:
50          new_cost = right_cost
51          h[i] = (-new_cost, node, to_ins, \
52                  path[(best_pos + 1) % (len(path))])
53      heapq.heapify(h)
54    self.tour = path
55    self.calculateCost()
56
57  self.tour = path
58  self.calculateCost()
```

Dalla riga 5 in poi avviene l'aggiornamento dello heap in modo del tutto analogo a

cheapest insertion. Le modifiche per tenere aggiornato lo heap con costi negativi sono le seguenti:

1. Riga 6: è presente l'istruzione `cost *= -1` in modo da riportare il costo negativo presente nello heap in un costo positivo, questo sarà utile successivamente (righe 23 e 30) dove il nuovo costo deve essere confrontato con il vecchio costo.

2. Righe 31, 40 e 50: i nuovi costi devono essere inserito con costo negativo nello heap, per questo motivo le variabili `new_cost` vengono moltiplicate per -1.

3. Inoltre alla riga 43 è presente l'istruzione `cost *= -1` per riportare la variabile `cost` ad un valore positivo in quanto alla riga 48, la condizione `right_cost < cost` necessita che la variabile cost sia positiva (altrimenti la condizione risulterebbe sempre falsa).

La restante parte dell'implementazione è del tutto identica a cheapest insertion.
Le considerazioni riguardo la complessità temporale di questa implementazioni sono le stesse della versione 2 di Cheapest Insertion, ovvero mantiene un tempo pari a $O(n^3)$ nel peggiore dei casi, ma nella pratica ha delle prestazioni (tempi) sempre migliori rispetto alla prima versione in quanto il terzo ciclo interno viene eseguito solo in certi casi e quindi non sempre.

### 3.3.9 Furthest Insertion: versione 3

Esattamente come nel caso della versione 3 di Cheapest Insertion, è possibile implementare una versione di Furthest Insertion con complessità minore di $O(n^3)$. Questa implementazione quindi condivide molto del codice visto nella versione 3 di Cheapest Insertion, con la differenza che lo heap principale deve operare come un max-heap (in modo da selezionare il nodo con costo maggiore) mentre gli heap interni devono continuare ad operare come dei min-heap in modo da selezionare, per un determinato nodo fuori dal tour, il punto di inserimento che minimizza il costo di inserzione. Vediamo l'implementazione:

```
1  def furthestInsertionOttimizzato(self, m):
2    n = self.numCity
3    adj = np.array(self.adj)
4    path = [0, 0]
5
6    maxDist = -1
7    for i in range(n):
8      for j in range(0, i):
9        if adj[i][j] > maxDist:
10         path[0], path[1] = i, j
11         maxDist = adj[i][j]
12   in_path = {path[0], path[1]}
```

Si inizializza il path con la coppia di città più lontane.

```
def furthestInsertionOttimizzato(self, m):
  ...
  h = []
  d = dict()
  d[(path[0], path[1])] = []
  d[(path[1], path[0])] = []
  for i in set(range(n)) - in_path:
    cost = adj[path[0]][i] + adj[i][path[1]] - adj[path[0]][path[1]]
    h_i = [-cost, i, path[0], path[1]]
    h_i2 = [-cost, i, path[1], path[0]]
    h.append([h_i, h_i2])
    d[(path[0], path[1])].append(h_i)
    d[(path[1], path[0])].append(h_i2)
  heapq.heapify(h)
```

Successivamente si inizializzano le due strutture dati fondamentali dell'implementazione, ovvero lo heap principale e il dizionario (come per la versione 3 di Cheapest Insertion) con la differenza che i costi vengono inizializzati come negativi in modo da consentire allo heap principale `h` di selezionare il record con costo di inserimento maggiore.

```
def furthestInsertionOttimizzato(self, m):
  ...
  while len(path) < n:
    h_i = heapq.heappop(h)
    (_, to_ins, l, r) = heapq.heappop(h_i)

    best_pos = path.index(r)
    path.insert(best_pos, to_ins)
    in_path.add(to_ins)
```

Dalla terza riga inizia il ciclo principale dell'euristica: dallo heap `h` si preleva il record con costo di inserimento massimo, dopodiché si procede con l'inserimento del nodo nel path.

```
def furthestInsertionOttimizzato(self, m):
  ...
  while len(path) < n:
    ...
    for p in d[(l, r)]:
      p[0] = np.inf
    for hp in h:
      node = hp[0][1]

      # sx: path[(best_pos - 1) % len(path)]
      # node
      # dx: path[best_pos]
```

```
13        sx = path[(best_pos - 1) % len(path)]
14        dx = path[best_pos]
15        newCost = adj[sx][node] + \
16                  adj[node][dx] - \
17                  adj[sx][dx]
18        l = [newCost, node, sx, dx]
19        if (sx, dx) not in d:
20          d[(sx, dx)] = [l]
21        else:
22          d[(sx, dx)].append(l)
23        hp.append(l)
24
25        # sx: path[best_pos]
26        # node
27        # dx: path[(best_pos + 1) % len(path)]
28        sx = path[best_pos]
29        dx = path[(best_pos + 1) % len(path)]
30        newCost = adj[sx][node] + \
31                  adj[node][dx] - \
32                  adj[sx][dx]
33        l = [newCost, node, sx, dx]
34        if (sx, dx) not in d:
35          d[(sx, dx)] = [l]
36        else:
37          d[(sx, dx)].append(l)
38        hp.append(l)
39
40        i = 0
41        for p in hp:
42          if p[0] < 0:
43            p[0] *= -1
44        while i < len(hp):
45          if hp[i][0] == np.inf:
46            del hp[i]
47          else:
48            i += 1
49        heapq.heapify(hp)
50        hp[0][0] *= -1
51      for i in range(len(h)):
52        h[i] = h[i][:m]
53      heapq.heapify(h)
54    self.tour = path
55    self.calculateCost()
```

La restante parte dell'implementazione è analoga alla versione 3 di Cheapest Insertion, con la differenza che è necessario mantenere il max-heap in modo che selezioni il record con costo di inserimento massimo (e non minimo) e gli heap interni come min-heap. Per ottenere questa proprietà, alla riga 41-43 si procede convertendo tutti i

valori negativi ad un valore positivo: in questo modo l'istruzione `heapq.heapify(hp)` procederà a porre in prima posizione il nodo con costo minimo (e non massimo). Successivamente per fare in modo che lo heap principale `h` operi come max-heap, è necessario che almeno tutti i costi nel primo record di ogni heap interno abbiano valore negativo, per questo motivo alla riga 50 il costo positivo viene convertito in un costo negativo. In questo modo, alla riga 53, l'istruzione `heapq.heapify(h)` potrà operare correttamente generando un max-heap.
Da un punto di vista della complessità temporale, l'algoritmo ha un tempo (nel caso peggiore) pari a $O(n^2 * m)$ come già visto per la descrizione di Cheapest Insertion versione 3.

### 3.3.10 Random Insertion

In questo capitolo verrà discussa l'implementazione di Random Insertion. Vedremo come di fatto è molto più semplice rispetto alle implementazioni viste prima in quanto bisogna:

1. Selezionare casualmente un nodo fuori dal tour
2. Inserirlo nel modo migliore possibile

Di seguito il metodo:

```
1 def randomInsertion(self):
2   n = self.numCity
3   distances = np.array(self.adj)
4   path = [0]
5   in_path = {0}
6   notInPath = [x for x in range(1, n)]
7   random.shuffle(notInPath)
```

L'euristica inizialmente sceglie come primo nodo arbitrario il nodo 0. Successivamente per quanto riguarda il primo punto discusso prima (scegliere casualmente il nodo da inserire), sono presenti due strade possibili:

1. Inizializzare una lista `[1, 2, ..., n]` tramite il metodo `range(1 ,n)` e poi nel ciclo principale dell'euristica selezionare casualmente un nodo dalla lista per poi rimuoverlo tramite il metodo `notInPath.remove(i)` (dove i è l'elemento da rimuovere) o tramite istruzione `del notInPath[pos]` dove pos è la posizione dell'elemento da rimuovere.
2. Inizializzare una lista `[1, 2, ..., n]` tramite il metodo `range(1 ,n)`, mescolare la lista tramite metodo `random.shuffle(notInPath)` e poi procedere con la selezione del nodo casuale tramite `notInPath.pop()`.

Da un punto di vista della complessità temporale, ho scelto la seconda strada in quanto rimuovere un elemento dalla lista (tramite `del` oppure `remove`) richiede un tempo $O(n)$ (dove n è la lunghezza della lista), mentre il metodo `l.pop()` ha complessità temporale $O(1)$[5]. Per questo motivo (contando il ciclo principale dell'euristica) il primo metodo richiederebbe in un tempo $O(n^2)$ (n iterazioni per il ciclo principale, O(n) per l'istruzione `remove` o `del`), mentre il secondo metodo richiederebbe un tempo $O(n)$ (n iterazioni per il ciclo principale e poi solo $O(1)$ per il metodo `notInPath.pop()`). Inoltre l'implementazione del metodo `random.shuffle()`[6] utilizza il l'algoritmo Fisher-Yates shuffle[13], ovvero un algoritmo che opera in tempo $O(n)$.

```
1  def randomInsertion(self):
2    ...
3    while len(path) < n:
4      to_insert = notInPath.pop()
5      best_increase = np.inf
6      best_position = None
7      for i in range(len(path)):
8        next_i = (i + 1) % len(path)
9        increase = distances[path[i], to_insert] + \
10                  distances[to_insert, path[next_i]] - \
11                  distances[path[i], path[next_i]]
12       if increase < best_increase:
13         best_increase = increase
14         best_position = next_i
15
16     path.insert(best_position, to_insert)
17     in_path.add(to_insert)
18
19   self.tour = path
20   self.calculateCost()
```

Come discusso precedentemente, il ciclo principale dell'algoritmo procede estraendo il nodo casuale (riga 4), successivamente viene effettuata una ricerca della posizione migliore dove inserire il nodo casuale. Voglio sottolineare come questo metodo non sia "completamente casuale", ma solo la parte della selezione del nodo fuori dal tuor è casuale, mentre la parte relativa all'inserimento all'interno del tour opera scegliendo il punto di inserzione che minimizzi il costo di inserimento.
In conclusione sono presenti le stesse istruzioni viste nelle altre euristiche (calcolo del costo).

[5]https://wiki.python.org/moin/TimeComplexity
[6]https://hg.python.org/cpython/file/2e8b28dbc395/Lib/random.py alla riga 276

### 3.3.11 Random Insertion: versione 2

Per completezza, ho implementato anche una versione di random insertion dove non solo la scelta del nodo da inserire nel tour avviene casualmente, ma anche il punto di inserzione viene scelto casualmente, di seguito l'implementazione:

```
1 def randomInsertion2(self):
2     tour = []
3     notInTour = [x for x in range(self.numCity)]
4     random.shuffle(notInTour)
5     for i in notInTour:
6       tour.insert(random.randint(0, len(tour) - 1), i)
7     self.tour = tour
8     self.calculateCost()
```

In questa versione dell'algoritmo, la prima parte è identica alla prima versione di random insertion (scelta casuale del nodo tramite una lista `notInTour` mescolata casualmente), mentre la seconda (ciclo principale) effettua un inserimento nel tour tramite istruzione `tour.insert(pos)` dove la posizione (pos) viene scelta casualmente tramite il metodo `random.randint()`.

### 3.3.12 Euristiche con inizializzazione casuale

Le euristiche viste nelle sezioni precedenti prevedono come inizializzazione del tour la coppia delle città più vicine per cheapest insertion e nearest insertion mentre prevedono la coppia delle città più lontane per farthest insertion e furthest insertion. Nel successivo capitolo verrà analizzata anche la robustezza delle euristiche implementate rispetto all'inizializzazione, per questo motivo nella classe TSP sono presenti 4 varianti delle euristiche descritte nelle sezioni precedenti dove le due città iniziali vengono scelte casualmente. Le 4 euristiche prendono il nome di NOMEInsertionRandomStart dove al posto di "NOME" è presente il nome dell'euristica (nearest, cheapest ...). Di seguito la descrizione della parte diversa:

```
1 def nearestInsertionRandomStart(self):
2   n = self.numCity
3   distances = np.array(self.adj)
4   path = [random.randint(0, n - 1), random.randint(0, n - 1)]
5   while path[0] == path[1]:
6     path = [random.randint(0, n - 1), random.randint(0, n - 1)]
7   in_path = {path[0], path[1]}
8   ...
```

In questo caso è stato selezionato nearestInsertionRandomStart a titolo di esempio (nelle altre 4 implementazioni il codice è lo stesso): vengono scelte due città casuali come prime due città nel tour. Successivamente (righe 5-6) è necessario ripetere il

processo nel caso in cui le due città scelte corrispondano alla stessa città. Una volta selezionati i primi due nodi casualmente, l'euristica può continuare normalmente come visto prima. Lo stesso criterio è stato applicato anche per nearestNeighbor, di seguito solo la prima istruzione dell'implementazione di nearestNeighborRandomStart:

```
def nearestNeighborRandomStart(self):
  self.tour = [random.randint(0, self.numCity - 1)]
  visited = set([self.tour[0]])
  ...
```

In questa variante (come per le euristiche basate su inserzione) il tour viene inizializzato con una città casuale (da 0 a $n-1$), inoltre viene aggiunto nodo casuale scelto nella riga precedente al set `visited` (mentre nell'implementazione di nearest nearestNeighbor viene aggiunto direttamente 0). La restante parte dell'implementazione è completamente uguale all'implementazione di nearestNeighbor.

# Capitolo 4

# Analisi dei risultati

In questo capitolo si analizzano i risultati trovati dalle euristiche precedentemente implementate.

# Bibliografia


[1] Traveling Salesman Problem, F. Greco, IntechOpen, 2008.

[2] M. Velednitsky, Short Combinatorial Proof that the DFJ Polytope is contained in the MTZ Polytope for the Asymmetric Traveling Salesman Problem, UC Berkeley, 2018.

[3] Analysis of Brute Force and Branch & Bound Algorithms to solve the Traveling Salesperson Problem (TSP), Informatics Department, Engineering Faculty, Widyatama University, 2021.

[4] T. G. Crainic, B. Le Cun, C. Roucairol, Parallel Branch-and-Bound Algorithms, Département de management et technologie École des Sciences de la Gestion Université du Québec à Montréal and CIRRELT, Canada

[5] Yuan Yuan, Diego Cattaruzza, Maxime Ogier, Frederic Semet. A branch-and-cut algorithm for the generalized traveling salesman problem with time windows. European Journal of Operational Research, 2020, 286 (3)

[6] R. Karp, Complexity of the Traveling Salesman Problem

[7] G. Reinelt, The Traveling Salesman: Computational Solution for TSP Applications, Heidelberg, Springer-Verlag

[8] J.E. Mitchell, Branch-and-cut algorithms for combinatorial optimization problems, in Handbook of applied optimization, P.M. Pardalos and M.G.C. Resende eds., Oxford University Press, 2000

[9] Olaf Mersmann, Bernd Bischl, Jakob Bossek, Heike Trautmann, Markus Wagner, Frank Neumann, Local Search and the Traveling Salesman Problem: A Feature-Based Characterization of Problem Hardness, Statistics Faculty, TU Dortmund University, Germany

[10] Matthias Englert, Heiko Röglin, Berthold Vöcking, Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP, Dept. of Computer Science, RWTH Aachen University

[11] S. Lin, B. W. Kernighan, An Effective Heuristic Algorithm for the Traveling- Salesman Problem, Bell Telephone Laboratories, Incorporated, Murray Hill, N.J., 1971

[12] K. Helsgaun, An effective implementation of the Lin-Kernighan traveling salesman heuristic, Department of Computer Science, Roskilde Universit, 13 April 1999

[13] Donald E. Knuth, The Art of Computer Programming: Volume 2 (Seminumerical Algorithms),Addison-Wesley Professional, 1997, Capitolo 3