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

FURTHEST INSERTION ALGORITHM

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

Ringraziamenti **iii**

Abstract **v**

1 Il Problema del Commesso Viaggiatore **1**
1.1 Il Problema del commesso viaggiatore 2
1.2 Formulazione del problema 2
1.3 Metodi esatti per il TSP 3

2 Euristiche per il TSP **7**
2.1 Nearest Neighbor 7
2.2 Euristiche basate su Inserzione 8
2.2.1 Nearest Insertion 9
2.2.2 Cheapest Insertion 9
2.2.3 Farthest Insertion 10
2.2.4 Furthest Insertion 10
2.2.5 Random Insertion 11

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-difficile [5], 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**[7]: 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. *Inizializzazione*: Sia ILP^0 il problema di programmazione lineare intera iniziale, sia $L = \{ILP^0\}$ la lista dei nodi attivi. Sia $\overline{z} = +\infty$ l'upperbound e $\underline{z}_l = -\infty$ il lower bound per l'unico problema $l \in L$.

2. *Terminazione*: Se $L = \emptyset$, allora la soluzione x^* che ha prodotto il valore obiettivo $\overline{z}$ è ottimale. Se non esiste nessuna x^* allora ILP è un problema inammissibile.

3. *Selezione del problema*: Seleziona ed elimina ILP^l da L.

4. *Rilassamento del problema*: Risolvi il rilassamento continuo del problema ILP^l: se è inammissibile allora assegna $\underline{z}_l = +\infty$ e vai al punto 6. Sia z_l il valore della funzione obiettivo del rilassamento se risulta finito e sia x^{lR} una soluzione ottima; altrimenti poniamo $\underline{z}_l = -\infty$.

5. *Aggiungi piani di taglio*: cerca i piani di taglio violati da x^{lR}; se sono presenti allora aggiungili al rilassamento e torna al passaggio 4.

6. *Fathoming and Pruning*:

   6.1. (a): Se $z_l \geq \overline{z}$ allora vai al punto 2.

   6.2. (b): Se $z_l < \overline{z}$ e x^{lR} è intero allora aggiorna $\overline{z} = z_l$, cancella tutti i problemi l dalla lista L tali che $z_l \geq \overline{z}$ e torna al punto 2.

7. *Partizionamento*: Sia $\{S^{lj}\}_{j=1}^{j=k}$ una partizione dell'insieme dei vincoli S^l del problema ILP^l. Aggiungi i problemi $\{ILP^{lj}\}_{j=1}^{j=k}$ nella lista L dove ILP^{lj} è ILP^l con la regione ammissibile ridotta a S^{lj} e $\underline{z}_{lj}$ per $j = 1, ..., k$ è l'insieme dei valori z_l per il problema l. Torna al punto 2.

Come già detto in precedenza, questo approccio è attualmente lo stato dell'arte per risolvere il TSP all'ottimo, implementazioni di algoritmi branch-and-cut per il TSP possono essere trovati 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 e le euristiche basate su inserzione, che partono da un ciclo iniziale al quale vengono aggiunte via via le città fino ad ottenere un ciclo Hamiltoniano e poi le euristiche basate su inserzione.

2.1 Nearest Neighbor

Nearest Neighbor è probabilmente l'euristica 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

Una soluzione per il problema del TSP può essere costruito in un modo differente: si può iniziare 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 vicine. 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 $r \in L$ che risulti il più vicino ad un nodo qualunque $j \in T$.
   2.2. Trova l'arco (i, j) nel tour T tale che $c_{ir} + c_{rj} - c_{ij}$ risulti minimo. Inserisci r tra i e j.
   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. Seleziona un nodo $r \in L$ e un arco (i, j) in T per cui $c_{ir} + c_{rj} - c_{ij}$ risulti minimo.
   2.2. Seleziona l'arco (i, j) in T tale che $c_{ir} + c_{rj} - c_{ij}$ risulti minimo. Inserisci r tra i e j.
   2.3. Rimuovi r da L, quindi $L = L \setminus \{r\}$.

Cheapest Insertion ha complessità computazionale pari a $O(n^2 log_2(n))$ in quanto è possibile implementare uno heap che tiene traccia dei nodi non ancora inseriti nel tour e dei relativi costi se inseriti nel modo migliore possibile.

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. Seleziona un nodo $r \in L$ che risulti il più lontano rispetto ad un nodo qualunque $j \in T$.
   - 2.2. Trova l'arco (i, j) nel tour T tale che $c_{ir} + c_{rj} - c_{ij}$ risulti minimo. Inserisci r tra i e j.
   - 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. Seleziona un nodo $r \in L$ e un arco (i, j) in T per cui $c_{ir} + c_{rj} - c_{ij}$ risulti massimo.
   2.2. Seleziona l'arco (i, j) in T tale che $c_{ir} + c_{rj} - c_{ij}$ risulti minimo. Inserisci r tra i e j.
   2.3. Rimuovi r da L, quindi $L = L \setminus \{r\}$.

Furthest Insertion ha complessità computazionale pari a $O(n^2 log_2(n))$ in modo del tutto analogo a Cheapest Insertion.

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.

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] R. Karp, Complexity of the Traveling Salesman Problem

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

[7] 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