LUISS

Dipartimento di Impresa e Management
Corso di laurea triennale in Economia e Management
Cattedra di Matematica Finanziaria

IL PROBLEMA DEL COMMESSO VIAGGIATORE: METODI ESATTI, EURISTICHE, PYTHON

RELATORE:
Prof. Gennaro Olivieri

CANDIDATO:
Tommaso Roncacci, Matr 222991

ANNO ACCADEMICO 2019 / 2020

A tutta la mia famiglia

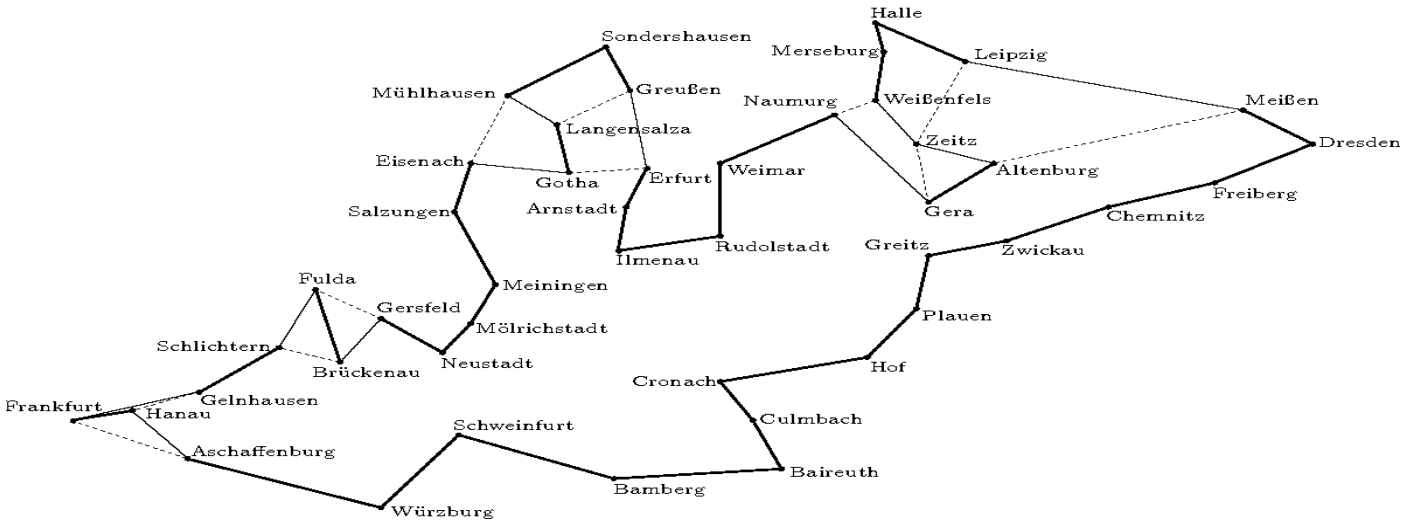
INDICE

INTRODUZIONE

Il problema del commesso viaggiatore (meglio conosciuto in inglese con il nome ***travelling salesman problem*)** è un noto problema della teoria dei grafi, strettamente collegato alla teoria della complessità computazionale ed alla teoria di ottimizzazione combinatoria.
Questo prende il nome dalla sua più tipica applicazione nell'ambito del routing e dello scheduling: dato un insieme di città, e definite le distanze tra ogni coppia di esse, si trovi il percorso di lunghezza minima che un commesso viaggiatore dovrebbe seguire per passare attraverso ogni città una ed una volta soltanto, tornando infine alla città di partenza.
Le origini storiche del problema non sono del tutto chiare. Una prima comparsa pratica dell'enigma è presente all'interno di un manuale per commercianti tedesco della prima metà del 1800 (manuale che tuttavia non presenta alcuna formulazione matematica).

Figura – 1

la figura mostra il tragitto proposto ai commercianti (linee continue) ed il percorso ottimale in termini di distanza (linee continue e tratteggiate).



Con la definizione di ciclo Hamiltoniano, all'interno della teoria dei grafi (dal matematico irlandese William Hamilton che creò anche un gioco basato sulla ricerca di un ciclo hamiltoniano tra diversi punti), vennero gettate le basi per lo studio matematico del problema

del commesso viaggiatore. Nel corso della storia, il dilemma venne studiato non solo in ambito matematico ma anche fisico, chimico ed informatico, raggiungendo una popolarità ed una complessità tale che tra il 1950 ed il 1960 la RAND corporation (uno tra i più importanti istituti di ricerca non profit negli Stati Uniti) offrì premi a coloro che fossero stati in grado di fornire passaggi per la risoluzione del problema.

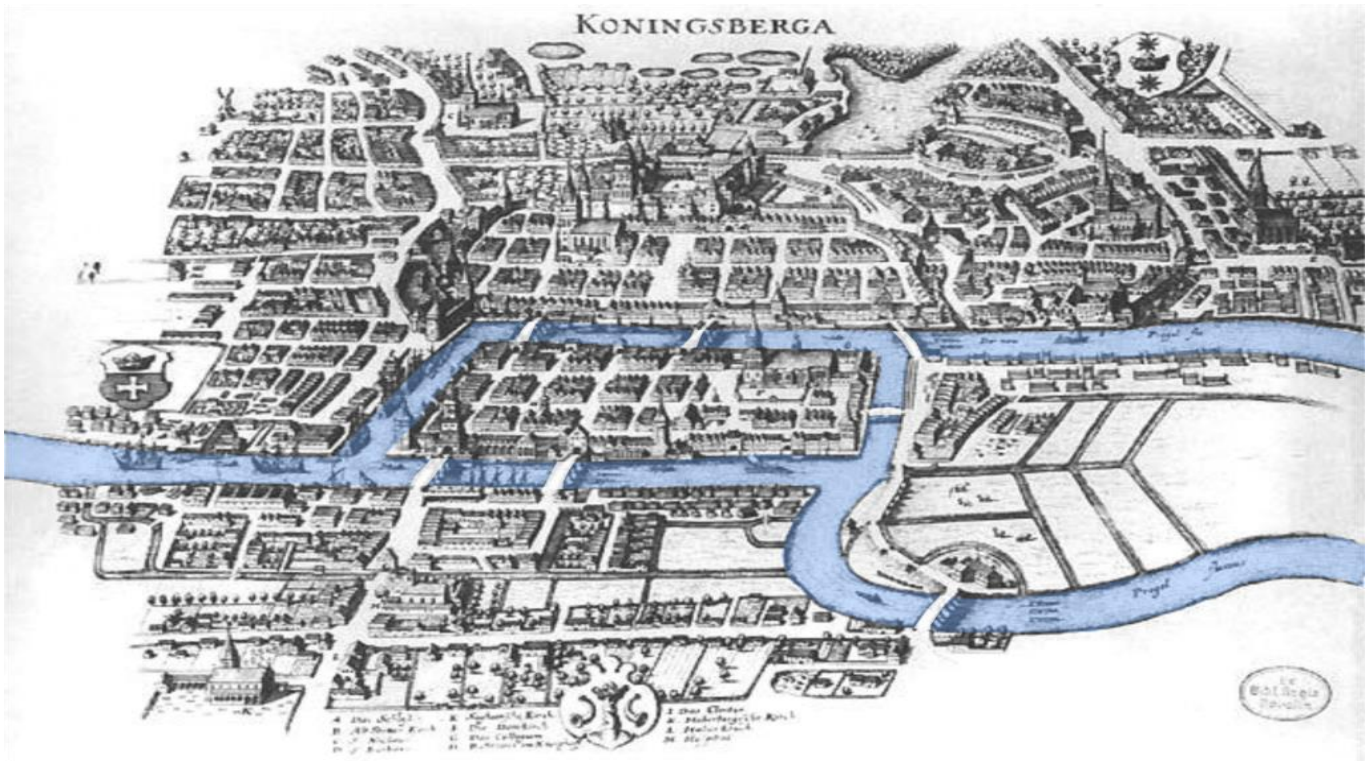
All'interno di questo scritto, saranno esposte nel primo capitolo le nozioni fondamentali della teoria dei grafi, nel secondo capitolo saranno definiti matematicamente e spiegati il TSP problem ed i principali algoritmi per la sua risoluzione, mentre nel terzo ed ultimo capitolo sarà illustrato un algoritmo in python da me proposto, per la risoluzione approssimativa del problema.

CAPITOLO 1 - TEORIA DEI GRAFI

1.1 STORIA DEI GRAFI

Nel 1736 lo svizzero Leonhard Euler (italianizzato Eulero), considerato uno tra i più importanti matematici e fisici del Settecento e della storia, risolse il dilemma conosciuto come ***il problema dei sette ponti di Konigsberg***, noto per essere uno tra i primi problemi di carattere toponomastico della storia.
La città di Konigsberg (oggi Kaliningrad), al tempo capoluogo della Prussia orientale, era attraversata dal fiume Pregel e dai suoi affluenti che dividevano la città in 3 penisole ed un'isola centrale collegate tra loro da 7 ponti.

Figura 2 – Città di Konigsberg



Il quesito che gli abitanti della città si ponevano al tempo era il seguente:
è possibile fare una passeggiata per la città percorrendo un tragitto che permetta di attraversare ogni singolo ponte una ed una volta soltanto?

Eulero dopo aver studiato il problema, decise di riformularlo prescindendo dal caso specifico di Konigsberg. Egli decise di sostituire ogni area urbana con un punto che definì ***vertice*** ed ogni ponte con un segmento che definì ***arco,*** sviluppando inoltre il concetto di ***grado di un vertice***, definendolo come il numero di archi che si collegano ad un vertice.
Basandosi su queste osservazioni egli enunciò il seguente teorema:
Un qualsiasi grafo è percorribile se e solo se ha tutti i nodi di grado pari, o se due di essi sono di grado dispari.
Da ciò Eulero dimostrò l'impossibilità di percorrere il tragitto proposto all'interno del problema.
I vocaboli quali vertice, arco e grado di un vertice, fondarono le basi della teoria dei grafi.
Da quel momento molti altri matematici, fisici e scienziati lavorarono su questa teoria implementandola con numerosi concetti.
Nel 1859 Willian Hamilton, matematico fisico ed astronomo irlandese propose il gioco dell'icosaedro, il quale scopo era simile a quello dei sette ponti di Konigsberg.
Lo scopo era quello di trovare un percorso che permettesse di attraversare ogni città (vertice) del gioco una ed una volta soltanto, tornando infine alla città di partenza;
se un tragitto permette di essere attraversato rispettando tali caratteristiche viene definito ***Hamiltoniano.***

Figura 3 – Gioco Icosiano, Hamilton 1859

1.2 DEFINIZIONI

I ***grafi*** sono strutture razionali matematiche che trovano il loro senso applicativo in diversi ambiti di studio tra cui i principali sono la scienza matematica, quella fisica e quella informatica fino ad arrivare alla topologia, alla chimica ed all'ingegneria.

Definizione 1 – grafo, multigrafo, grafo semplice, grafo nullo

Viene definito ***grafo*** una struttura matematica caratterizzata da un insieme di elementi chiamati ***nodi*** o ***vertici*** collegati tra loro da linee definite ***spigoli, archi o lati.*** Matematicamente parlando un grafo è una coppia ordinata di insiemi ***G= (V, E)*** con ***V(G)*** rappresentante l'insieme dei ***vertici*** ed ***E(G)*** l'insieme degli ***archi***, tale che $E \subseteq VxV$ (con VxV prodotto cartesiano di V per sé stesso). Solitamente ***V*** ed ***E*** vengono assunti come finiti e la maggior parte dei risultati più noti non sono veri per i ***grafi infiniti*** (cioè quei grafi in cui ***V*** ed ***E*** vengono presi infiniti). Nel caso in cui ***E(G)*** sia un multi insieme si parla non di grafo ma di ***multigrafo.*** Un multigrafo allora è una struttura caratterizzata da un insieme finito di vertici e archi che collegano due o più vertici, oppure un vertice con sé stesso (si parla in questa evenienza di ***cappio o self loop***). Nel caso in cui all'interno del grafo non siano presenti cappi allora il grafo viene definito ***semplice.***

Un grafo $G= (V, \varnothing)$, ossia un grafo privo di archi, viene definito ***grafo nullo.*** Il grafo $G= (\varnothing, \varnothing)$, ossia un grafo in cui sia l'insieme dei nodi che l'insieme dei vertici è vuoto, è un caso estremo di grafo nullo.

Definizione 2 – estremi, ordine, dimensione, grado

All'interno di un grafo, due ***vertici (u, v)*** legati da un arco ***e,*** prendono il nome di ***estremi dell'arco*** e sono definiti ***adiacenti;*** tale arco ***e,*** inoltre, può essere identificato attraverso la coppia composta dai suoi estremi, nonché vertici, (***u, v);*** non necessariamente però una coppia di vertici rappresenta gli estremi di un arco ($(u, v) \notin E(G)$**).** L'arco che collega due vertici è definito ***arco incidente;*** 2 archi sono definiti ***adiacenti*** se hanno un vertice in comune.

Il numero di vertici in un grafo determina ***l'ordine n*** del grafo, per cui $n = |V(G)|$.

Il numero degli archi ***m*** in un grafo determina ***la dimensione*** del grafo stesso; se vengono considerati grafi semplici, si può dire che un generico grafo semplice G ha dimensione ***m***

appartenente all'intervallo dato dal ***coefficiente binomiale n su 2*** (con n il numero di vertici presenti all'interno del grafo); osservando quindi un grafo semplice con n nodi ed m archi, può essere osservato a priori che il numero di archi può variare da un minimo di 0 ad un massimo di n su 2.

Il concetto di ***dimensione*** può essere traslato dal grafo al singolo vertice $v \in V(G)$; dato un vertice v, viene definito ***grado del vertice k(v),*** il numero di archi incidenti al vertice v (ossia il numero di archi che si connettono ad esso); laddove un arco sia connesso allo stesso vertice da ambedue le estremità (dunque nel caso in cui ci sia un cappio), esso viene conteggiato due volte.

Il ***grado massimo di G*** viene definito come il grado del vertice avente il maggiore numero di archi incidenti. Il ***grado minimo di G*** viene definito invece come il grado del vertice con il minor numero di archi incidenti. Nell'eventualità il grado massimo sia coincidente con il grado minimo, pari ad un numero ***k***, ci si trova in presenza di un ***grafo k-regolare o grafo regolare.***

Definizione 3 – grafo orientato e non orientato

Nel caso in cui E(G) sia una ***relazione simmetrica (o di equivalenza,*** il grafo viene definito ***non orientato (indiretto)***. in caso contrario si parla di grafo ***orientato (diretto).***

Figura 4 - grafo non orientato.

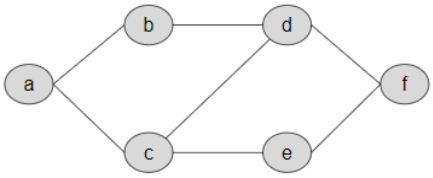
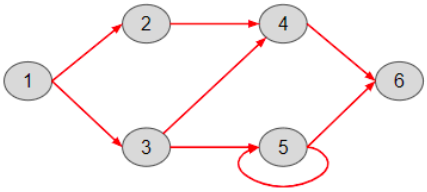


Figura 5 - grafo orientato



Definizione 4 – Grafo completo (clique) e grafo complemento

Viene definito ***completo*** quel grafo semplice in cui è impossibile aggiungere un arco senza replicarne uno già presente. Più formalmente è ***completo*** quel grafo per cui, presi 2 qualsiasi vertici, essi si dimostrano adiacenti**.** Viene inoltre definito ***grafo complemento*** di G, quel grafo composto dagli stessi vertici di G e da tutti gli archi non presenti in G.

Figura 6 - grafo completo (clique).

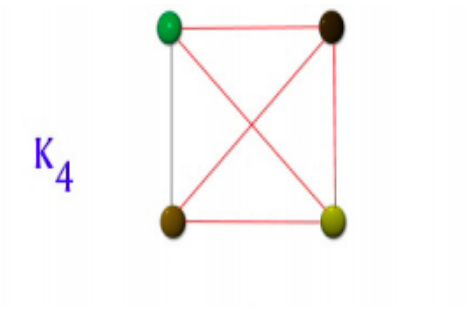
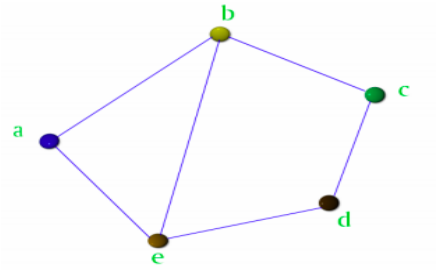
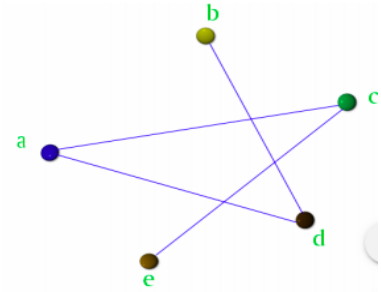


Figura 7 – grafo e grafo complemento



Grafo G



Grafo complemento di G

Definizione 5 – percorso, cammino, circuito, ciclo

All'interno di un grafo G, è un ***'percorso',*** avente ***lunghezza n*** in G, quell'elemento formato da una sequenza di vertici v0,v1,….,vn (non obbligatoriamente tutti diversi), e da una sequenza di archi che collegano i vn vertici (v0, v1),(v1, v2),(vn-1, Vn). Il vertice dalla quale parte il percorso e quello alla quale esso termina, prendono il nome di ***estremi*** del percorso.

Dati 2 vertici v, w $\in$G(V) in un grafo G, un ***cammino*** da v a w è una sequenza di vertici (v0, v1,…, vn$\in$ V non necessariamente diversi) ed archi (e1,e2,…, en$\in$ E tutti distinti) tale che (ei,ei+1)$\in$E(G) con i=0,1,…,n-1 (quindi ogni 2 archi consecutivi sono adiacenti).

Il numero intero ***n*** dato dal numero di archi rappresenta la ***lunghezza*** del cammino.

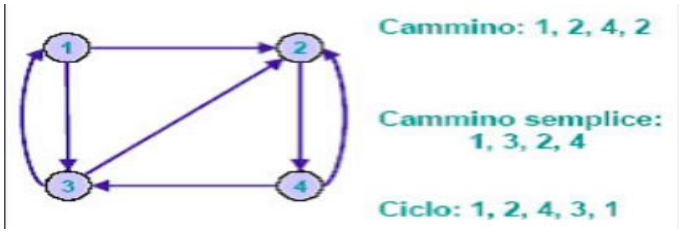
Se un cammino non passa per più di una volta attraverso uno stesso vertice allora il cammino è di tipo ***semplice.***

Una sequenza di vertici ed archi (v0e1- v1e2- … -vn-1en -vn) in un grafo G, viene definita ***passeggiata*** se viene rispettata la sola ed unica condizione per la quale ogni arco ei (con i=0, 1, …, n) abbia come estremi i vertici vi-1, vi (risulta allora possibile percorrere più di una sola volta ogni arco, anche avanti e indietro). Ne risulta quindi che ogni cammino è una passeggiata ma non il contrario.

Nel caso in cui l'estremo di partenza di un cammino non coincida con l'estremo di arrivo si parla di ***cammino aperto;*** se l'estremo di partenza (v0) coincide con l'estremo di arrivo (vn), dunque se v0=vn, allora il cammino viene definito ***circuito.*** Se nel circuito tutti i vertici sono distinti tra loro (tranne quello di partenza, che per la definizione di circuito, coincide con il vertice di arrivo) allora si è in presenza di un ***circuito semplice (***quindi se per ogni $1 \leq i, j \leq n$, allora $i \neq j$, e $v_i \neq v_j$ (il cammino non passa più di una volta per lo stesso vertice); si parla di ***circuito non semplice*** in caso contrario***.***

Un ***ciclo*** è un circuito semplice costituito da almeno 3 archi.

Figura 7 – esempio di cammino, cammino semplice e ciclo



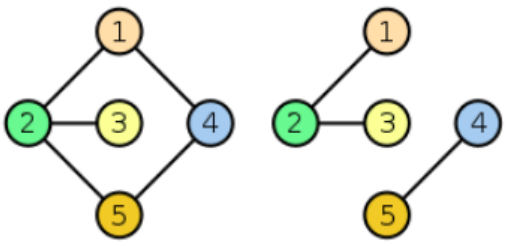
Definizione 6 – connettività

Un grafo G viene considerato ***connesso*** se per ogni coppia di suoi vertici v,w esiste in G un cammino tra v e w. Quando questa condizione non persiste il grafo è ***non connesso.***

Un arco ed un vertice che se eliminati rendono il grafo sconnesso vengono definiti rispettivamente ***'ponte' e 'snodo'***

Viene definito ***nodo isolato*** un vertice che non è connesso a nessun altro vertice.

Figura 8 – grafo connesso (sulla sinistra) e grafo sconnesso (sulla destra)



Definizione 7 – cammini Eureliani Hamiltoniani

Dato un grafo G=(V, E), viene definito ***euleriano*** in G, quel particolare cammino che rispetta la condizione secondo la quale l'insieme degli archi appartenenti al cammino corrisponda con l'intero insieme degli archi appartenenti al grafo G (dunque se l'insieme composto dagli archi e1, e2, e3, ….., en =E(G)); analogamente un circuito è euleriano nel caso in cui l'insieme dei suoi lati sia uguale a tutto E e dunque se se il circuito attraversa una ed una volta soltanto ogni arco di G. Nell'evenienza in cui all'interno di un grafo G sia presente un circuito euleriano allora il grafo prende il nome di ***grafo euleriano.***

Dallo studio del problema ***dei 'sette ponti di Konigsberg'*** (che domandava circa l'esistenza di un circuito euleriano per passare attraverso tutti i ponti una ed una volta soltanto, arrivando al punto di partenza), Eulero formulò ciò che viene definito il ***primo teorema fondamentale*** della teoria dei grafi nella storia della matematica occidentale.

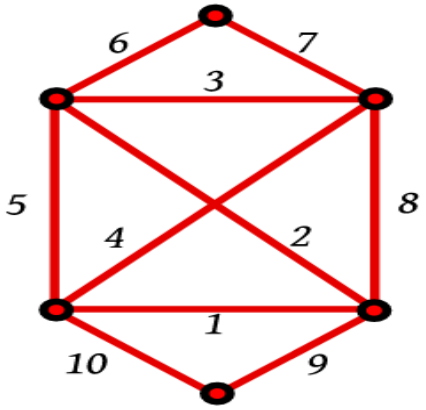
primo teorema fondamentale (Eulero):

Un grafo senza vertici isolati è euleriano se e soltanto se è connesso ed ogni suo vertice ha grado pari.

Da un adattamento al primo teorema fondamentale è stato possibile completare il risultato di Eulero al caso dei cammini Euleriani, derivando che:

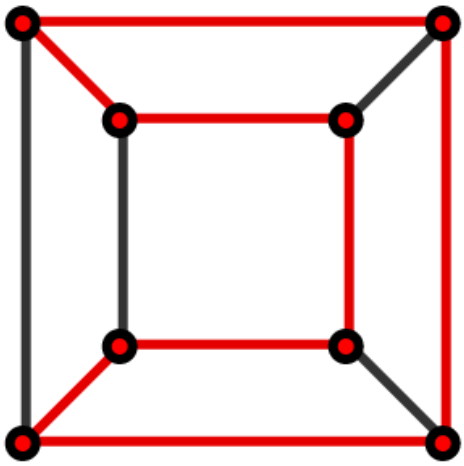
Un grafo G senza vertici isolati ha un cammino euleriano non chiuso, se e soltanto se, G è connesso ed ha esattamente 2 vertici dispari. In questa evenienza, se i vertici u e v sono i 2 vertici dispari, allora tutti i cammini euleriani di G iniziano e terminano u e v.

Figura 9 – grafo euleriano



Nel 1859 sir William Rowan Hamilton propose (anche commercializzandolo) il ***'problema del viaggiatore sul dodecaedro'.*** Conferiti i nomi di città famose ad ogni vertice di un dodecaedro, il problema richiedeva di verificare la possibilità per un viaggiatore di partire da una città-vertice, passare una ed una volta soltanto per ogni città-vertice tornando infine alla città-vertice di partenza. Come suggerì lo stesso Hamilton il problema deve essere inquadrato sotto il punto di vista grafico cercando dunque un circuito semplice nel grafo del dodecaedro che passa una ed una volta soltanto per tutti i vertici del grafo. Dato un grafo G=(V, E), un cammino (v0 e1 v1 e2 v2 e3 ….. en vn) in G viene definito ***Hamiltoniano*** se è semplice e (v0, v1, …, vn)=V(G). Analogamente un circuito è ***hamiltoniano*** se è semplice (quindi è un ciclo) e se l'insieme dei vertici che lo compongono è tutto V(G). Un grafo all'interno del quale è presente un ciclo hamiltoniano viene definito ***grafo hamiltoniano.*** La realizzazione di un programma che risulti efficiente nella verifica dell'esistenza di un circuito hamiltoniano all'interno di un grafo, risulta essere un problema molto complesso (questo perché, non si conosce alcuna caratterizzazione dei grafi hamiltoniani simile a quanto accade contrariamente per i circuiti hamiltoniani.)

Figura 10 – grafo hamiltoniano



La ricerca di un ciclo hamiltoniano è dunque alla base del ***problema del commesso viaggiatore.***

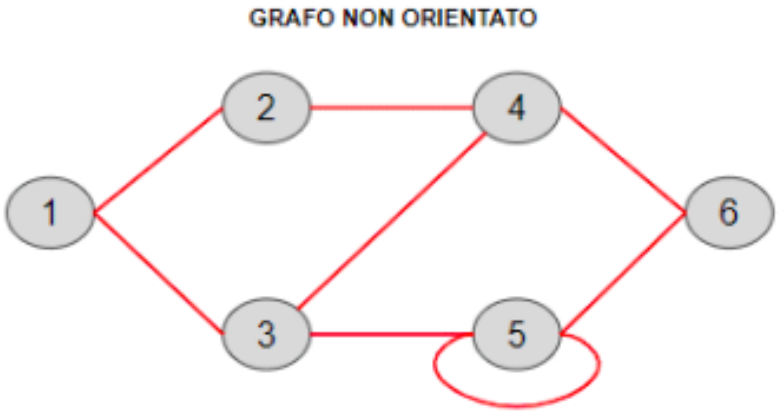
1.3 RAPPRESENTAZIONE DI GRAFI

Per far si che i grafi possano essere rappresentati in maniera adeguata da un calcolatore sono necessarie strutture dati che possano essere implementate in memoria e utilizzabili da da un programma. Queste strutture dati sono la ***matrice di adiacenza*** e la ***lista di adiacenza.***

Matrice di adiacenza

Dato un grafo G e fissato un ordinamento totale di V(G), viene definita ***matrice di adiacenza*** del grafo G, la matrice binaria quadrata A(G) che ha come indici di righe (i) e colonne (j) i vertici del grafo (dunque la matrice è indicizzata ai vertici). All'interno della matrice nella cella (in, jn) si trova un 1, se e soltanto se, all'interno del grafo esiste un arco che collega il vertice in al vertice jn; nel caso in cui tra i 2 vertici i, j non vi è alcun arco allora è presente uno 0. Nel caso in cui, all'interno della matrice, non siano presenti degli 1 ma numeri positivi diversi, questi ultimi devono essere interpretati come il peso attribuito ad ogni contatto. Per esempio nel caso in cui i vertici di un grafo rappresentino una serie di città, allora il peso degli archi può essere interpretato come la distanza tra le città che l'arco connette. Se la matrice offre la rappresentazione di un grafo non orientato allora essa è simmetrica rispetto alla diagonale principale (risulterebbe quindi uno spreco di memoria rappresentare l'intera matrice in quanto la diagonale principale separa 2 aree totalmente identiche; in tal caso, è possibile e più conveniente memorizzare soltanto una delle 2 aree insieme alla diagonale principale risparmiando così spazio in memoria.).

Figura 11- matrice di adiacenza di un grafo non orientato



nodi destinazione

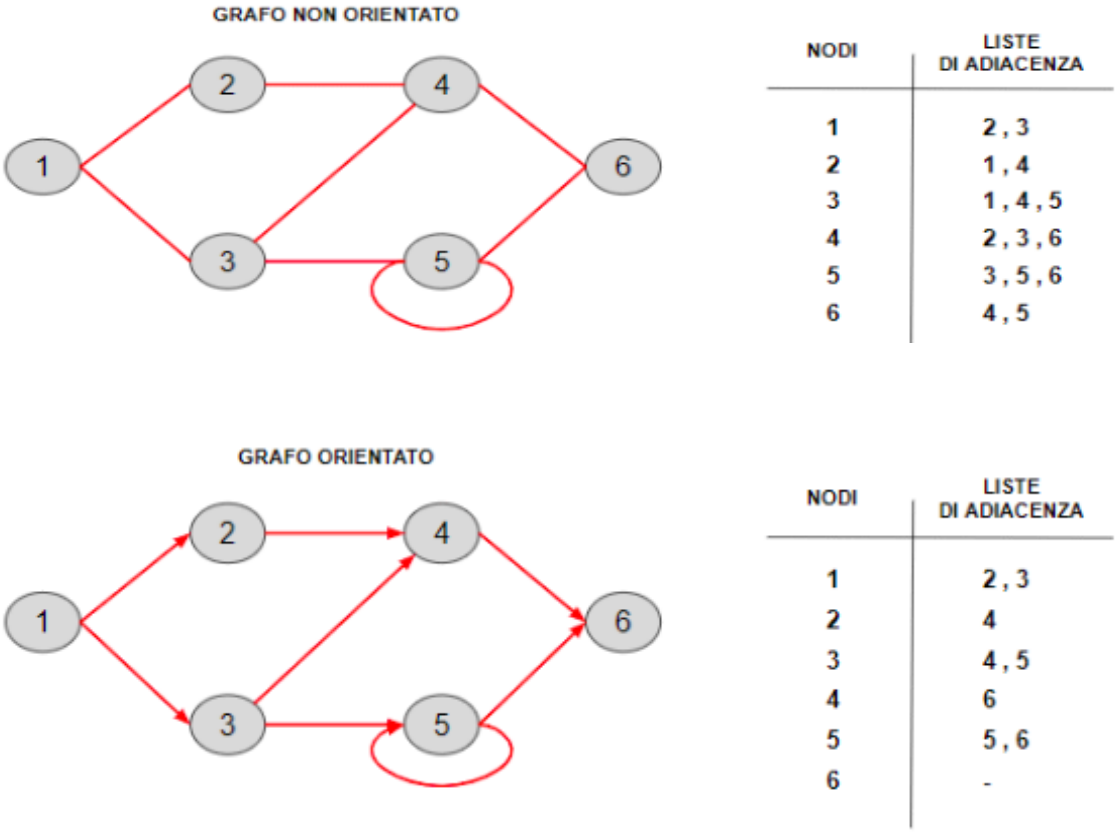
nodi di partenza	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	1	0
4	0	1	1	0	0	1
5	0	0	1	0	1	1
6	0	0	0	1	1	0

E' una matrice trasposta

Lista di adiacenza

La lista di adiacenza è la modalità di rappresentazione di un grafo più immediata e più semplice da implementare, sebbene in generale non risulta essere la più efficiente in termini di memoria occupata. La rappresentazione tramite liste di adiacenza avviene attraverso il collegamento ad ogni vertice V di un grafo G una lista che contiene tutti i vertici W ai quali il vertice V è collegato attraverso un arco (V, W). Ogni vertice risulta essere quindi un elemento di una variabile array V[n] alla quale viene assegnata la lista delle coppie ordinate che collega. Così come nella rappresentazione attraverso matrici di adiacenza, anche usando liste di adiacenza si colgono alcune differenze tra la rappresentazione di grafi orientati e grafi non orientati. In presenza di grafi orientati, ogni arco ha una certa direzione, quindi è una relazione univoca. In caso di grafi non orientati invece la relazione è biunivoca.

Figura 12 – liste di adiacenza



1.3.1 EFFICIENZA DI RAPPRESENTAZIONE

Le liste di adiacenza e le matrici di adiacenza hanno un certo costo computazionale e occupano una certa quantità di memoria. Sorge dunque spontaneo il quesito riguardo la convenienza di un metodo di rappresentazione rispetto ad un altro, avendo entrambi i propri vantaggi e svantaggi.

Nel caso in cui si voglia rappresentare un grafo ***'sparso',*** risulta essere più conveniente la rappresentazione attraverso ***liste di adiacenza***. Un grafo viene definito ***sparso*** quando il numero degli archi $|E|$ è sostanzialmente piccolo rispetto al numero dei vertici $|V|$.

Se si vuole rappresentare un grafo ***'denso'***, risulta invece più conveniente la rappresentazione attraverso ***matrici di adiacenza***. Un grafo è detto ***denso*** quando il numero degli archi $|E| \simeq |V|^2$. Inoltre le matrici consentono di comprendere immediatamente se 2 vertici di un grafo sono collegati tra loro, mentre per le liste la ricerca è più lunga. Ne deriva, dunque, che le matrici hanno una complessità temporale di accesso ai dati inferiore rispetto alle liste. Tuttavia, le matrici di adiacenza rispetto alle liste occupano uno spazio di memoria più grande (che il grafico sia orientato o non orientato). Generalmente si può dire che lo spazio di memoria di una lista è:

Il numero $|E|$ per i grafici orientati, $2|E|$ per i grafici non orienati.

Per quanto riguarda le matrici di adiacenza esse occupano uno spazio di memoria pari a: $|V|^2$ per i grafici orientati, e $|V|$ per quelli non orientati.

Le liste di adiacenza inoltre hanno in qualsiasi caso (sia per grafi orientati che per grafi non orientati) una complessità pari a $O(V+E)$, mentre le matrici di adiacenza hanno un costo pari a $O(V^2)$.

Per queste ragioni, risulta più conveniente lavorare con liste di adiacenza nel caso in cui il tempo di ricerca non costituisce un problema.

CAPITOLO 2 – *IL PROBLEMA DEL COMMESSO VIAGGIATORE*

Si supponga che un commerciante, per adempiere al suo ruolo, debba passare per un numero n di città una ed una sola volta, tornando a fine giornata al punto di partenza, al minor costo possibile (costo che può essere tradotto per esempio in distanza, tempo, prezzo ecc.).
Affinché il commerciante svolga correttamente il suo lavoro, egli prova ad identificare i vari percorsi praticabili, confrontandoli e scegliendo infine di muoversi verso quello più conveniente. Questo breve esempio spiega il ***problema del commesso viaggiatore.***

All'interno di tale enigma dunque, l'obiettivo risulta essere quello di individuare in modo efficiente un metodo che permetta di scegliere tra i vari percorsi (che soddisfino le varie condizioni) quello di costo minimo. Tale situazione può essere rappresentata tramite un grafo completo G con n vertici (città da dover visitare) e m archi pesati, in cui si cerca di identificare, prima di tutto la presenza di eventuali percorsi hamiltoniani (quindi di stabilire se il grafo è hamiltoniano) e successivamente di stabilire il circuito hamiltoniano di peso minimo.

2.1 - TSP SIMMETRICO ED ASIMMETRICO

Un'importante distinzione che occorre fare è quella tra il problema del commesso viaggiatore ***simmetrico ed asimmetrico.***

Il problema viene definito ***simmetrico*** nel caso in cui tra due città (vertici all'interno di un grafo G), qualsiasi sia la direzione, la distanza risulta essere la stessa. In questa situazione dunque, ci si trova in presenza di un grafo non orientato.

Al contrario, il tsp ***asimmetrico*** è quel tipo di problema in cui la direzione considerata per spostarsi tra 2 città (vertici) potrebbe influire sulla distanza tra le città stesse. È questo il caso in cui ci si trova davanti ad un grafo orientato. Questa differenza è estremamente importante sotto il punto di vista delle soluzioni poiché nel caso del tsp di tipo simmetrico, la simmetria stessa consente di dimezzare il numero delle soluzioni possibili.

2.2 – DEFINIZIONE MATEMATICA

Il problema del commesso viaggiatore è rappresentabile attraverso un grafo G= (V, E) tale che V(G)=(v1,v2,v3,….,vn) sia l'insieme dei vertici all'interno del grafo G (che rappresentano le città) ed E(G)=(e1,e2,e3,…., en) gli archi del grafo G (che rappresentano le strade tra le città). Viene indicato inoltre con c_{ij} il costo dell'arco che unisce il vertice i al vertice j (distanza tra le città i e j). Nel caso in cui $\mathbf{c_{ij} = c_{ji} \forall (i, j)}$ il problema è simmetrico altrimenti è asimmetrico; a seconda che il problema sia di un tipo o di un altro, esso può essere rappresentato attraverso modelli matematici differenti.

Definita $\mathbf{x_{ij}}$ la variabile binaria tale che $\mathbf{x_{ij} = 1}$ se l'arco ij fa parte del circuito Hamitoniano e $\mathbf{x_{ij} = 0}$ in caso opposto, è possibile enunciare matematicamente il problema nell'ipotesi più generale di simmetria come segue:

$$z = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \quad (1)$$

sotto i vincoli:

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n \qquad (2)$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n \qquad (3)$$

$$\sum_{i \in Q} \sum_{j \in V \setminus Q} x_{ij} \geq 1 \qquad \forall Q \subset V, |Q| \geq 1 \qquad (4)$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n$$

in questa definizione:

la (1) è la ***funzione obiettivo*** (funzione che mappa un evento ad un numero reale e che solitamente rappresenta il costo associato all'evento) che consiste nella minimizzazione del costo del cammino hamiltoniano.

La funzione obiettivo viene poi sottoposta ai vincoli (2), (3) definiti ***vincoli di assegnazione*** che sono necessari per puntualizzare che in ogni nodo **i** entra ed esce un solo arco; tuttavia, poiché questi vincoli non assicurano una soluzione formata da un unico circuito, è necessario definire la (1) anche sotto il vincolo (4) che assicura l'assenza di sotto circuiti, stabilendo che,

scegliendo un qualsiasi sottoinsieme di nodi Q in V, deve esistere almeno un lato che colleghi un nodo di Q con un nodo non appartenente a V.

2.3 – EFFICIENZA E COMPLESSITÀ COMPUTAZIONALE

Nonostante a primo impatto il problema del commesso viaggiatore possa risultare 'semplice' ad occhi inesperti, in realtà è tutt'altro che facile trovare una soluzione; questo perché il punto fondamentale dell'enigma non è tanto trovare un algoritmo che sia in grado di individuare un ciclo hamiltoniano di peso minimo, quanto trovare un algoritmo che risulti ***efficiente.*** Secondo la ***teoria della complessità computazionale*** (branca della ***teoria della ricorsività*** che studia le risorse minime necessarie affinché un problema venga risolto), per ***efficienza*** di un algoritmo si fa riferimento alle risorse di calcolo richieste dallo stesso (principalmente ***tempo di calcolo*** e ***memoria***). Diverse sono le ***classi di complessità*** in cui i problemi vengono raggruppati in base all'efficienza del miglior algoritmo conosciuto in grado di risolvere quel particolare problema.
Una distinzione importante è quella tra problemi considerati ***semplici*** (per i quali algoritmi risolutivi sono ***efficienti***) e quelli considerati ***complessi o difficili*** (per i quali algoritmi ovvi sono ***non efficienti***). Per il problema del commesso viaggiatore dunque, l'algoritmo 'ovvio' (***algoritmo brute-force*** o ***algoritmo di Enumerazione Totale)*** non risulta essere efficiente; tale algoritmo consiste prima di tutto nel prendere in considerazione i cicli hamiltoniani nel grafo, calcolare per ognuno di essi il peso totale e scegliere infine quello di peso minimo. L'inefficienza è facilmente dimostrabile attraverso l'analisi dell'algoritmo ovvio; se infatti viene preso un grafo **G** con **V** vertici ed archi **E**, ogni successione di v-vertici (all'interno di cui ogni vertice si presenta una ed una volta soltanto) rappresenta la sequenza di vertici di un ciclo hamiltoniano; allora se viene fissato il vertice di partenza v1 dei diversi cicli è chiaro che ci sia una correlazione biunivoca tra i cicli nel grafo G ed i modi per ordinare i rimanenti v-1 vertici. ***Da ciò, ne deriva che in G il numero di cicli hamiltoniani è uguale al numero di permutazioni dei v-1 vertici diversi da v1, uguale quindi a (v-1)!.***

È evidente allora, che l'algoritmo ovvio richieda un numero di passaggi crescente al crescere del numero di vertici v, come v! nel caso peggiore, in cui ogni nodo è connesso con tutti gli altri, e che diventi presto un numero intrattabile per un qualsiasi elaboratore in quanto ciò implica una **complessità di tipo esponenziale**. Proprio per questo motivo potrebbe risultare più efficiente adottare per la risoluzione del problema in questione un approccio ***non deterministico***, in cui l'algoritmo può avere comportamenti diversi su diversi percorsi e che non fornisce una soluzione ottima, ma una soluzione che se pure approssimata risulta corretta (permettendo cosi di risparmiare sul costo dell'algoritmo stesso rendendolo polinomiale).
Per queste ragioni quello del commesso viaggiatore è uno dei dilemmi appartenenti alla classe dei problemi considerati ***NP-completi (non deterministic polinomial***). Appartengono a questa classe di complessità computazionale i problemi più complessi della classe ***NP (***ovvero i problemi considerati ***non deterministici in tempo polinomiale),*** in relazione al fatto che se venisse trovato un algoritmo capace di risolvere in maniera 'rapida' (in termini di ***tempo polinomiale***) un qual si voglia problema del tipo NP-completo, allora lo stesso algoritmo potrebbe essere applicato per risolvere sempre in maniera 'rapida' ogni problema del tipo NP.

2.4 – SOLUZIONI DEL PROBLEMA: EURISTICHE E METODI ESATTI

Diverse sono le alternative all'algoritmo brute force che hanno come obbiettivo la risoluzione del problema del commesso viaggiatore. La distinzione più importante che occorre fare tra le soluzioni stesse è quella che divide i **metodi esatti** dai **metodi euristici.**
Sono considerati **esatti,** quei metodi che determinano la soluzione ottima del problema ma che hanno bisogno di tempo esponenziale al crescere del numero dei vertici.
Vengono definiti **euristici,** quei metodi che consistono in algoritmi circa veloci (generalmente sono in grado di offrire una soluzione approssimata in tempo polinomiale) che tuttavia incorporano un errore variabile rispetto alla soluzione ottima.

2.4.1 – METODI ESATTI

In questa sezione del mio elaborato saranno presentati alcuni tra i più importanti metodi esatti per la risoluzione del problema in analisi tra cui **l'algoritmo Held-Karp ed il metodo branch-and-bound.** Questi metodi nonostante richiedano un tempo esponenziale per trovare la soluzione esatta al problema, sono comunque più rapidi rispetto all'algoritmo brute-force che per arrivare alla soluzione necessita di tempo fattoriale all'aumentare dei vertici.

2.4.2 – L'ALGORIMO HELD-KARP

Quello di Held-Karp, anche conosciuto con il nome **Bellman-Held-Karp**, è un algoritmo che è stato proposto indipendentemente dai 3 matematici nel 1962 per risolvere il problema del commesso viaggiatore. L'algoritmo è costruito sulla base dei principi della **programmazione dinamica,** una tecnica risolutiva di problemi complessi fondata sulla divisione di un problema in sotto problemi e sull'utilizzo di sottostrutture ottimali; questo vuol dire che dalle soluzioni ottimali dei sotto problemi, che vengono trovate attraverso l'uso della ricorsione, è possibile giungere alla soluzione ottimale dell'intero problema.
Dato un grafo G dunque, l'idea sulla quale si fonda l'algoritmo Held-Karp per risolvere il problema in esame è quella di, selezionare un vertice di partenza e calcolare la soluzione ottima per tutti i sotto circuiti di lunghezza N usando tutte le informazioni raccolte dalle soluzioni ottimali dei sotto circuiti di lunghezza N-1.
al vertice 3 che è pari a 3).
w[1,2]= 21 (poiché w(0-2-1) è uguale a ***w[2,]= 15 +*** il peso dell'arco che collega il vertice 2 al vertice 1 e pari a 6).
w[3,2]= 27 (poiché che w(0-2-3) è uguale a ***w[2,]= 15 +*** il peso dell'arco che collega il vertice 2 al vertice 3 e pari a 12).
w[1,3]=10 (poiché w(0-3-1) è uguale a ***w[3,] =6*** + il peso dell'arco che collega il vertice 3 al vertice 1 e pari a 4).
w[2,3]= 14 (poiché w(0-3-2) è uguale a ***w[3,]=6*** + il peso dell'arco che collega il vertice 3 al vertice 2 e pari a 8).

Una volta terminato il procedimento attraverso i sottoinsiemi composti da 1 elemento, si passa ad utilizzare i sottoinsiemi composti da 2 elementi. In questa maniera si passerà dal cercare il costo di un percorso a cercare il percorso di costo minimo. Infatti se si prende w[3, (1,2)] si osserva che il percorso potrebbe essere 0-1-2-3 oppure 0-2-1-3. Proprio per questo motivo è necessario calcolare il costo di entrambi i percorsi (sfruttando i risultati ottenuti in precedenza) e prendere in considerazione esclusivamente il percorso avente costo minimo (in quanto il fine ultimo dell'algoritmo è calcolare proprio il circuito hamiltoniano di costo minimo).

Proseguendo quindi con i sottoinsiemi composti da 2 elementi si ottengono questi risultati:
min w[3,(1,2)]=
a) ***w[3,(1,2)]=24*** (poiché il costo del percorso 0-2-1-3 è pari al costo di ***[1,2]= 21 +*** il costo dell'arco che collega il vertice 1 al vertice 3 uguale a 3).
b***) w[3,(2,1)]=20*** (poiché il costo del percorso 0-1-2-3 è pari al costo di ***[2, 1]= 8 +*** il costo dell'arco che collega il vertice 2 al vertice 3 uguale a 12).
Quindi il percorso meno costoso è 0-1-2-3 dato che w[3,(2,1)]=20.

min w[1,(2,3)]=
a) ***w[1,(2,3)]= 20*** (poiché il costo del percorso 0-3-2-1 è pari al costo di ***[2,3]= 14 +*** il costo dell'arco che collega il vertice 2 al vertice 1 uguale a 6).
b) ***w[1,(3,2)]= 31*** (poiché il costo del percorso 0-2-3-1 è pari al costo di ***[3,2]= 27*** + il costo dell'arco che collega il vertice 3 al vertice 1 uguale a 4).
Quindi il percorso meno costoso è 0-3-2-1 dato che w[1,(2,3)]= 20.

min w[2,(1,3)]=
a) ***w[2,(1,3)]= 17*** (poiché il costo del percorso 0-3-1-2 è pari al costo di ***[1,3]=10*** + il costo dell'arco che collega il vertice 1 al vertice 2 uguale a 7).
b) ***w[2,(3,1)]=12*** (poiché il costo del percorso 0-1-3-2 è pari al costo di ***[3, 1]= 4*** + il costo dell'arco che collega il vertice 3 al vertice 2 uguale a 4).
Quindi il percorso meno costoso è 0-1-3-2 dato che w[2,(3,1)]=12.

L'ultima fase dell'algoritmo consiste nel prendere in considerazione l'ultimo sottoinsieme composto da 3 elementi (1, 2, 3). In questa fase, tramite i risultati ottenuti grazie ai sottoinsiemi più piccoli, è possibile dunque ottenere la risposta al problema del commesso viaggiatore. Utilizzando il solito procedimento [0, (1,2,3)] si avranno 3 possibilità diverse, nonché 3 percorsi diversi; infatti il vertice d'arrivo 0 (che è anche quello di partenza) potrebbe essere preceduto dal vertice 1 dal vertice 2 o dal vertice 3. Anche in questa situazione bisognerà allora calcolare prima di tutto il costo dei 3 ipotetici percorsi e successivamente mettere in evidenza il percorso di costo minimo che rappresenta proprio la soluzione al problema del commesso viaggiatore.

I risultati per l'esempio preso in considerazione sono i seguenti:
min w[0(1,2,3)]=

a) ***w{0, [1,(2,3)]}= 22*** (in questo caso il costo del percorso 0-3-2-1-0 è uguale al costo dell'arco che collega il vertice 1 al vertice 0, uguale a 2 , + il costo del sottopercorso di costo minimo ***[1,(2,3)] pari a 20***)

b) ***w{0, [2,(3,1)]}= 21*** (in questo caso il costo del percorso 0-1-3-2-0 è uguale al costo dell'arco che collega il vertice 2 al vertice 0, uguale a 9, + il costo del sottopercorso di costo minimo ***[2,(3,1)] pari a 12***. Si noti che viene selezionato [2(3,1)] e non [2(1,3)], poiché [2(1,3)] risulta avere un costo più alto)

c) ***w{0, [3,(2,1)]}=30*** (in questo caso il costo del percorso 0-2-1-3-0 è uguale al costo dell'arco che collega il vertice 3 al vertice 0 uguale a 10 + il costo del sottopercorso ***[3,(2,1)] pari a 20***. Si noti che viene selezionato [3,(2,1)] e non [3,(1,2)] poiché [3,(1,2)] risulta essere più costoso.)
Quindi il percorso meno costoso risulta essere b 0-1-3-2-0 =21. Osservando la matrice dei costi si può verificare che il costo totale degli archi è effettivamente 21.
Questa è la soluzione al problema del commesso viaggiatore per l'esempio proposto.
L'algoritmo Held-Karp-Bellman è un metodo esatto più efficiente dell'algoritmo di forza bruta, tuttavia risulta essere meno efficiente al crescere del numero dei vertici all'interno del

grafo. Infatti esso ha ***complessità temporale di tipo esponenziale*** poiché è presente un numero ipotetico esponenziale di sottoinsiemi e per ogni sottoinsieme si dovrà prendere in considerazione ogni singolo vertice. Proprio per questo motivo si dice che l'algoritmo ha complessità pari a $O(2^n \times n^2)$.

2.4.3 - L'ALGORITMO BRANCH AND BOUND

L'algoritmo Branch and Bound è un algoritmo che sfrutta la tecnica generica per la risoluzione di problemi di ottimizzazione combinatoria (che risulta nella scelta della soluzione migliore rispetto alle altre) e che si fonda sulla scomposizione del problema generale in sotto problemi di risoluzione più semplice (similmente a quanto accade nell'algoritmo di Held-Karp). Gli algoritmi che si basano su questa tecnica di risoluzione sono detti di ***enumerazione implicita***; questo perché hanno un comportamento simile a quello degli algoritmi di enumerazione, in quanto tentano di trovare la soluzione ottima passando per tutte quelle possibili, con la notevole differenza che i branch and bound riescono a scartare a priori alcune soluzioni (senza dunque 'esplorarle'), dimostrando la loro non ottimalità.

Il nome dell'algoritmo proviene innanzitutto dalla tecnica di suddivisione di un problema in problemi più semplici (***branching***).

È possibile quindi definire l'algoritmo come segue.

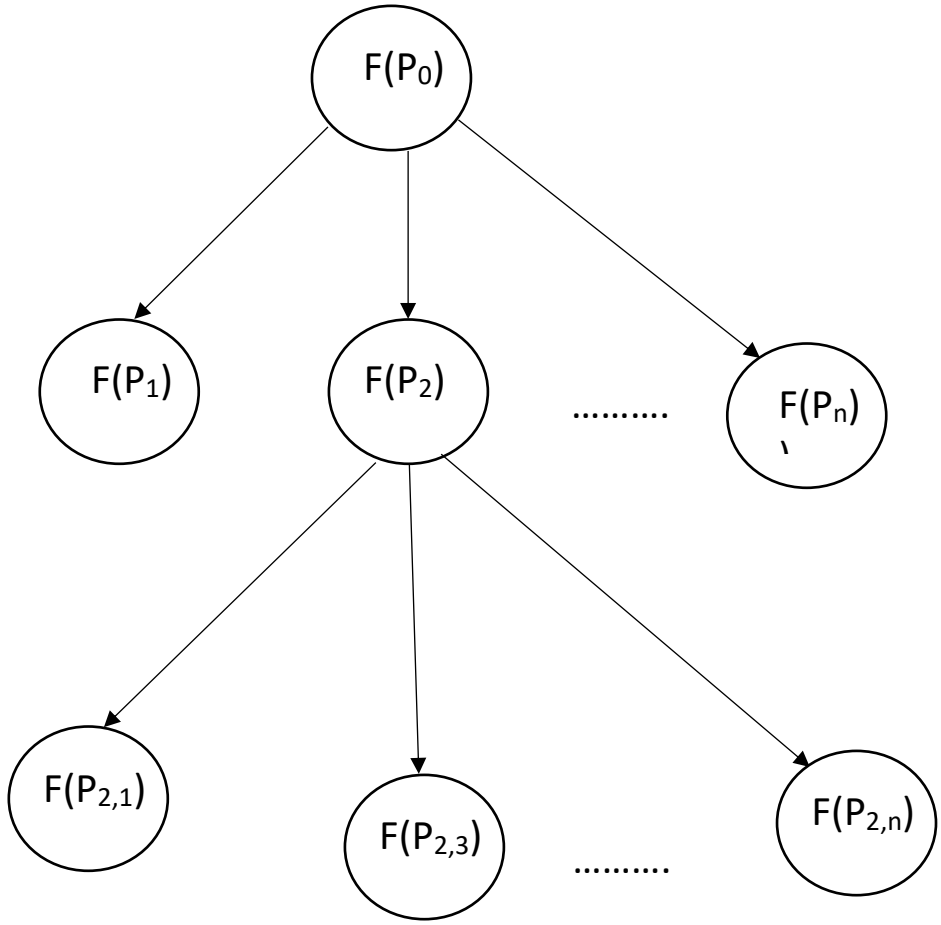
Si ipotizzi di avere un problema $P_0=(z,F(P_0))$ in cui z è la ***funzione obbiettivo*** del problema, mentre $F(P_0)$ rappresenta la ***regione delle soluzioni ammissibili***. La ***migliore soluzione ammissibile*** è $z^* = z(P_0)= \{z(x) : x \in F(P_0)\}$ mentre z^{best} è la ***migliore soluzione possibile nota.*** Il problema principale P_0 è scomponibile in n problemi più piccoli: $P_1, P_2, P_3, \ldots, P_n$ tali che $P_1+P_2+P_3+\ldots+P_n=P_0$ e lo stesso può essere fatto con la regione delle soluzioni ammissibili $F(P_0)$ in modo da ottenere n sotto regioni $F(P_1), F(P_2), F(P_3),\ldots., F(P_n)$ tali che

$F(P_1)+F(P_2)+F(P_3)+\ldots..+ F(P_n)=F(P_0)$ e possibilmente tali che,

$F(P_i)\cap F(P_j)=\varnothing \ \forall P_i,P_j:i\neq j$

Questa situazione può essere rappresentata attraverso un ***albero decisionale*** (o state space tree o ancora brench decision tree).

Figura 14 – albero decisionale



Quindi risolvere il problema P_0 equivale a risolvere i suoi sotto problemi:

$$z^*=z(P_0)=min\{z(P_1), z(P_2), \ldots, z(P_n)\}.$$

In questo modo suddividendo i sotto problemi in altrettanti sotto problemi (esplorando quindi i possibili collegamenti tra nodi) è possibile riuscire ad arrivare alla soluzione del problema del commesso viaggiatore. Come detto in precedenza, la peculiarità dell'algoritmo branch and bound sta nella capacità di eliminare soluzioni che a priori risultano essere 'peggiori' in termini di costo rispetto alla soluzione migliore conosciuta z^{best}; questo si verifica quindi nel caso in cui $z(P_i)>z^{best}$ (il sotto problema P_i può essere tralasciato in quanto la sua soluzione risulta essere peggiore rispetto a quella migliore conosciuta z^{best}). Una delle tecniche per eliminare a priori le soluzioni che risultano essere meno efficienti rispetto a quella ottima conosciuta è quella di utilizzare un ***lower bound*** (viene presa la sotto soluzione di un sotto problema come parametro di confronto delle sotto soluzioni degli altri problemi; se la sotto soluzione di un altro sotto problema è meno efficiente rispetto al parametro allora il sotto problema viene scartato a priori). Per trovare il lower bound di un sotto problema è

necessario trovare il suo ***rilassamento***. Viene definito ***rilassamento*** di un problema P, un problema ***P'*** tale che l'insieme delle soluzioni ammissibili di P sono contenute in quello di P'. Tramite proprio la nozione di rilassamento è possibile procedere con l'eliminazione dei sotto problemi più costosi; infatti un problema P (o sotto problema) si dice eliminato se un suo rilassamento p':

1) Non ammette soluzioni
2) Il valore ottimo della funzione obbiettivo P' è maggiore rispetto a quella corrente.

Le caratteristiche principali dell'algoritmo branch and bound quindi risultano essere le seguenti:

1) La soluzione ottima è raggiunta in step successivi partendo da un nodo iniziale
2) In ogni nodo viene considerata una soluzione parziale
3) Per ogni soluzione parziale viene definita una funzione di lower bound
4) Le soluzioni parziali vengono elaborata tramite un albero
5) L'algoritmo si interrompe nel momento in cui viene trovata la soluzione ottima che rispetta tutti i vincoli

Per comprendere al meglio il funzionamento dell'algoritmo si consideri questo esempio.
Dato un grafo pesato composto da 5 vertici, si vuole calcolare il ciclo hamiltoniano minimo all'interno del grafo attraverso il metodo branch and bound.
Il primo passo per risolvere il problema consiste nello stabilire il vertice di partenza (vertice 1 nell'esempio) e rappresentare (nel caso in cui non è già data) la matrice dei costi. Una volta fatto questo inizia la fase di branching (cioè di scomposizione del problema in sotto problemi più semplici) che può essere rappresentata graficamente tramite un albero decisionale.
Tramite esso è possibile vedere chiaramente quali sono tutti i percorsi possibili (prendendo per implicito il fatto che ognuno di essi ritorni al vertice di partenza ossia il vertice 1).
Dopo aver deciso il vertice di partenza e dopo aver costruito l'albero decisionale è necessario calcolare il 'costo' del vertice di partenza. Per fare questo occorre ridurre la matrice dei costi. Una matrice viene definita ridotta se per ogni riga ed ogni colonna è presente un elemento pari a 0.

Matrice dei costi del grafo

∅	20	30	10	11
15	∅	16	4	2
3	5	∅	2	4
19	6	18	∅	3
16	4	7	16	∅

A questo punto per trovare la matrice ridotta, bisogna selezionare il minor elemento per ogni riga e sottrarlo ad ogni elemento della riga presa in considerazione e successivamente prendere il minor elemento di ogni colonna e sottrarlo ad ogni elemento della colonna selezionata.

La prima riga della matrice ha come elemento minimo il valore 10 (costo dell'arco che collega il vertice 1 al vertice 4) quindi si sottrae ad ogni elemento della prima riga il valore 10. Una volta fatto ciò per ogni riga si ottiene la seguente matrice.

Matrice ridotta per riga

∅	10	20	0	1
13	∅	14	2	0
1	3	∅	0	2
16	3	15	∅	0
12	0	3	12	∅

Il **costo della riduzione per riga** è la somma dei valori mini sottratti ad ogni riga.

In questo caso il costo della riduzione per riga equivale quindi a 21.

Per ultimare la riduzione occorre ora ridurre le colonne partendo dalla matrice ridotta per righe.

La prima colonna della matrice ha come elemento minimo il valore 1, quindi viene sottratto 1 da ogni elemento della prima colonna. Una volta eseguito lo stesso procedimento per ogni colonna, si ottiene la matrice ridotta.

Matrice ridotta

Ø	10	17	0	1
12	Ø	11	2	0
0	3	Ø	0	2
15	3	12	Ø	0
11	0	0	12	Ø

Il **costo della riduzione per colonne** equivale alla somma dei valori mini sottratti ad ogni colonna. In questo caso dunque il costo della riduzione per colonne equivale a 4.
Il **costo totale della riduzione** equivale alla somma tra il costo della riduzione per righe ed il costo per la riduzione per colonne; in questo caso quindi il costo della riduzione equivale a 21+4=25.
Una volta calcolato il costo della riduzione è possibile calcolare **Il costo del vertice di partenza**. Per il primo vertice esso equivale al costo totale della riduzione, ossia 25.

Terminata questa procedura, ci si trova in una situazione in cui, dal vertice di partenza bisogna scegliere verso quale vertice andare, ovvero se proseguire il percorso verso il vertice 2, 3, 4 o 5. Il parametro per cui viene scelto il vertice successivo al vertice 1, è il costo del vertice successivo; dato che il fine ultimo dell'algoritmo è di trovare il ciclo hamiltoniano di costo minimo, la scelta deve quindi ricadere sul vertice dotato di minor costo. In questa fase dunque avviene il primo ***'branching'***, ossia la prima suddivisione del problema in problemi più piccoli. A questo punto dunque è necessario calcolare il costo di ogni vertice verso il quale è possibile andare dal vertice 1. Per calcolare il costo, partendo dalla matrice di partenza ridotta (tramite la quale si è calcolato il costo del vertice 1), si deve cercare la matrice ridotta relativa ad ogni vertice. Questo significa prendere la matrice ridotta di partenza, eliminare la riga relativa al vertice di partenza, la colonna relativa al vertice d'arrivo, e l'arco che mette in relazione vertice d'arrivo con quello di partenza (questo per togliere la possibilità di tornare indietro), ed infine se la matrice ottenuta non risulta essere ridotta, calcolare il costo dell'eventuale riduzione. Il costo del vertice d'arrivo è la somma tra il peso dell'arco che collega il vertice di partenza a quello di arrivo $c(v_i, v_j)$ nella matrice ridotta, il costo della matrice ridotta iniziale (**γ**)ed il costo di un'eventuale seconda riduzione

(**γ'**). Si voglia quindi calcolare il costo del vertice 2. La matrice modificata (ovvero scorporata di prima riga, seconda colonna e costo dell'arco (2,1))
è la seguente.

Matrice modificata per vertice 2 da 1

Ø	Ø	Ø	Ø	Ø
Ø	Ø	11	2	0
0	Ø	Ø	0	2
15	Ø	12	Ø	0
11	Ø	0	12	Ø

Dato che la matrice risulta essere ridotta, non è necessario fare ulteriori modifiche. **Il costo del vertice 2 pertanto risulta essere = c(1, 2)+ γ= 10 + 25=35.**

A questo punto si deve procedere calcolando il costo degli altri vertici.

Matrice ridotta del vertice 3 da 1

Ø	Ø	Ø	Ø	Ø
1	Ø	Ø	2	0
Ø	3	Ø	0	2
4	3	Ø	Ø	0
0	0	Ø	12	Ø

Costo della seconda riduzione = 11

Costo del vertice 3 = c(1,3)+ γ+ γ'= 17+25+11= 53

Matrice ridotta del vertice 4 da 1

Ø	Ø	Ø	Ø	Ø
12	Ø	11	Ø	0
0	3	Ø	Ø	2
Ø	3	12	Ø	Ø
11	0	0	Ø	Ø

Costo della seconda riduzione= 0

Costo del vertice 4= c(1,4)+ γ= 0+25=25

Matrice ridotta del vertice 5 da 1

Ø	Ø	Ø	Ø	Ø
10	Ø	Ø	0	Ø
0	3	Ø	0	Ø
12	0	9	Ø	Ø
Ø	0	0	12	Ø

Costo della seconda riduzione = 5

Costo del vertice 5 = c(1,5)+ γ+ γ'= 1+ 25+5= 31

Avendo quindi calcolato il costo di ogni vertice di arrivo è chiaro che il vertice 4, con un costo di 25, rappresenta l'alternativa più efficiente di spostamento. La scelta del vertice più efficiente rappresenta il concretizzarsi dell'operazione di ***'bound'***, ossia la valutazione ottimistica della funzione obbiettivo per le soluzioni rappresentate da ciascun nodo (bound), in modo da non dover esplorare completamente tutte le vie possibili dell'albero decisionale. Scelto quindi il secondo vertice, cioè il vertice 4, si procede con la seconda fase di branching ossia la 'creazione' di 3 nuovi sotto problemi rappresentati dai vertici tra
cui scegliere il terzo all'interno del percorso. A questo punto, come nella fase precedente, bisogna calcolare il costo di ogni vertice per capire quale sia quello di costo minimo. Per fare questo occorre nuovamente calcolare, prima le matrici modificate relative ad ogni vertice da esplorare (2, 3, 5), e poi calcolare la matrice ridotta. Bisogna ora tener conto del fatto che, la matrice di riferimento tramite cui calcolare le suddette matrici, non è più la matrice ridotta iniziale relativa al vertice 1, ma quella relativa al vertice di partenza della seconda fase di branching, ossia quella del vertice numero 4.
Partendo quindi dalla matrice ridotta del vertice 4 si ottengono i seguenti risultati.

Matrice ridotta del vertice 2 da 4

Ø	Ø	Ø	Ø	Ø
Ø	Ø	**11**	Ø	**0**
0	Ø	Ø	Ø	**2**
Ø	Ø	Ø	Ø	Ø
11	Ø	**0**	Ø	Ø

Costo della seconda riduzione = 0

Costo del vertice 2 = c(4, 2)+ γ= 3 + 25=28

Matrice ridotta del vertice 3 da 4

Ø	Ø	Ø	Ø	Ø
1	Ø	Ø	Ø	**0**
Ø	**1**	Ø	Ø	**0**
Ø	Ø	Ø	Ø	Ø
0	**0**	Ø	Ø	Ø

Costo della seconda riduzione =13

Costo del vertice 3= c(4, 3)+ γ+ γ'=12+25+13=50

Matrice ridotta del vertice 5 da 4

Ø	Ø	Ø	Ø	Ø
1	Ø	**0**	Ø	Ø
0	**3**	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
Ø	**0**	**0**	Ø	Ø

Costo della seconda riduzione= 11

Costo del vertice 5= c(4, 5)+ γ+ γ'=0+25+11=36

Terminata l'esplorazione dei vertici 2, 3 e 5 si può osservare che quello avente costo minore è il numero 2 (28). A questo punto quindi si seleziona proprio il numero 2 come terzo vertice all'interno del percorso (seconda fase di bound).
Inizia dunque la terza fase di branching in cui vengono esplorati, partendo dal 2, i vertici 3 e 5. Si ripete quindi il calcolo dei costi di tali vertici attraverso le relative matrici modificate e ridotte, partendo dalla matrice ridotta del vertice numero 2.

Matrice ridotta del vertice 3 da 2

Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	0
Ø	Ø	Ø	Ø	Ø
0	Ø	Ø	Ø	Ø

Costo della seconda riduzione = 13

Costo del vertice 3= c(2, 3) + γ+ γ'= 11+28+13=52

Matrice ridotta del vertice 5 da 2

Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
0	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
Ø	Ø	0	Ø	Ø

Costo della seconda riduzione= 0

Costo del vertice 5= c(2, 5) + γ= 0+28=28

Poiché il vertice 5 risulta avere il minor costo allora sarà questo il quarto vertice all'interno del percorso (terza fase di bound).
Dopo la terza fase di branching è chiaro che l'unico vertice restante da esplorare è il numero 3. Per esplorarlo si ricorre nuovamente alla sua matrice ridotta prendendo come matrice di partenza la ridotta del vertice numero 5.

Matrice ridotta del vertice 3 da 5

Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
0	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø

Costo della seconda riduzione= 0

Costo del vertice 3=c(5, 3)+ γ= 0+28=28

Terminata questa procedura si può dire con certezza che nella soluzione trovata ***l'upperbound*** (ossia il limite superiore rappresentato dall'ultimo vertice) è 28. Poiché nessun vertice non esplorato ha un costo inferiore a 28, allora non vi è alcuna necessità di esplorare gli atri vertici perché porterebbero a soluzioni meno efficienti rispetto a quella trovata muovendosi passo dopo passo verso il vertice avente costo inferiore. Esclusi quindi i vertici non esplorati, per trovare la soluzione non resta che tornare al vertice di partenza 1 chiudendo in tal modo il circuito hamiltoniano di costo minimo.
Soluzione: 1-4-2-5-3-1 con costo di 28.

Nel caso in cui ci fossero stati nodi aventi un costo inferiore rispetto a quello dell'ultimo vertice esplorato, allora sarebbe stato necessario esplorare tali vertici e riproporre le varie fasi di branching.

2.4.4 – ALGORITMI EURISTICI

A causa della 'difficoltà' della maggior parte dei problemi di ottimizzazione combinatoria, spesso risulta essere conveniente e necessario lo sviluppo di ***algoritmi euristici.***
Questi tipi di algoritmi non forniscono la garanzia di trovare la soluzione ottima, però sono in grado di fornire una '***buona***' soluzione per il problema. Per buona si intende che, nonostante la soluzione trovata potrebbe non essere quella ottima, è pur sempre una soluzione ***ammissibile*** per il problema. Il grande vantaggio che si ottiene nello sviluppo di euristiche allora, è la capacità di calcolare una soluzione ammissibile in un tempo più rapido (***polinomiale)*** rispetto a quanto accade con i metodi esatti per la risoluzione di problemi complessi (***esponenziale***). Generalmente gli algoritmi euristici non hanno elevata complessità computazionale, sebbene per problemi di grandi dimensioni e dotati di una struttura complessa, potrebbe essere necessario sviluppare algoritmi più sofisticati. Inoltre essi sono soggetti a possibile ***fallimento,*** nel senso che potrebbero non trovare alcuna soluzione senza però dimostrare che non ce ne sia alcuna.
Due tra le euristiche più usate per la risoluzione del problema del commesso viaggiatore sono il metodo ***Nearest Neighbor,*** ed il metodo chiamato ***Twice around the three.***

2.4.5 –NEAREST NEIGHBOR

Il metodo nearest neighbor è uno tra i metodi euristici più utilizzati data la sua semplicità. Applicato al problema del commesso viaggiatore, esso si basa sulla costruzione del ciclo hamiltoniano muovendosi da una città a quella più vicina fino al momento in cui tutte sono state visitate. La costruzione del ciclo hamiltoniano (trovato in questa maniera) deve essere ripetuta ogni volta con una città di partenza diversa fino all'esaurirsi delle n città.
Una volta ottenuti tutti i cicli, quello avente costo inferiore risulta essere la scelta migliore.
Si osservi il seguente esempio.

Data la matrice dei costi di un grafo pesato non orientato e composto da 5 nodi, si vuole trovare una soluzione al problema del commesso viaggiatore per il relativo grafo.

Matrice dei costi

–	**10**	**8**	**9**	**7**
10	**–**	**10**	**5**	**6**
8	**10**	**–**	**8**	**9**
9	**5**	**8**	**–**	**6**
7	**6**	**9**	**6**	**–**

Partendo dalla città 1 si costruisce il ciclo hamiltoniano muovendosi da una città a quella più vicina.

Ciclo hamiltoniano partendo dalla città 1.

Osservando la prima riga della matrice dei costi, si nota che la città più vicina è la numero 5 poiché la distanza pari a 7 è minore rispetto alla distanza dalle altre città.

Dalla città 5 osservando la quinta riga si nota che ci si potrebbe muovere verso la città numero 2 o quella numero 4 poiché hanno la stessa distanza. Arbitrariamente ci si muove la città numero 2. Da quest'ultima si nota che la città più vicina è il numero 4 e successivamente dalla 4 si passa alla 3. A questo punto tornando alla città di partenza si ottiene il **ciclo 1: 1-5-2-4-3-1 di costo= 34.**

Ripetendo il procedimento si ottengono i seguenti cicli con città di partenza diverse.

Ciclo 2: 2-4-5-1-3-2 di costo=36

Ciclo 3: 3-1-5-2-4-3 di costo=34

Ciclo 4: 4-2-5-1-3-4 di costo=34

Ciclo5: 5-2-4-3-1-5 di costo=34

L'ultima fase dell'algoritmo prevede dunque la scelta della soluzione migliore in base al minor costo. In questo esempio è evidente che 4 soluzioni su 5 hanno lo stesso costo e se si fornisce attenzione ai circuiti si può notare che il ciclo 1 con il ciclo 3 ed il ciclo 5 presentano lo stesso schema dispositivo delle città. Si può concludere quindi dicendo che i cicli 1, 3, 4 e 5 sono soluzioni ammissibili.

Tali soluzioni possono eventualmente coincidere con la soluzione ottima, ma questo può essere verificato esclusivamente attraverso il calcolo della stessa tramite un metodo esatto.

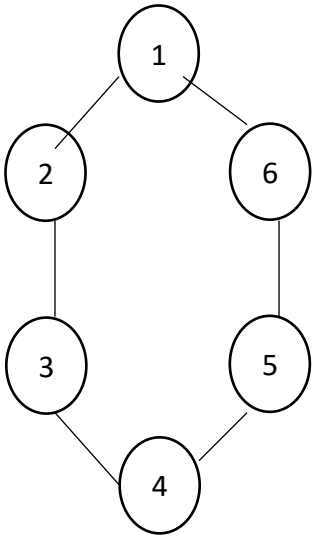
Se non si è a conoscenza della soluzione ottima, si può dire che, sia L_0 la lunghezza del tour ottimo e L_H la lunghezza del tour trovato tramite il metodo euristico, sussiste la seguente relazione tra le 2 lunghezze: $L_H \geq L_0$ e quindi L_h rappresenta un limite superiore.
Nel momento in cui viene utilizzato un metodo euristico per la ricerca di soluzioni ammissibili per problemi complessi, è bene analizzare l'efficacia della soluzione euristica rispetto alla soluzione ottima, ossia quanto quest'ultima 'diverge' dall'euristica.
A tale proposito è possibile derivare alcune espressioni in grado di esprimere la misura entro la quale cui la soluzione ammissibile sovrasta la soluzione ottima.
Relativamente all'algoritmo nearest neighbor, si può dire che $(L_H/L_0) \leq (1+\log_2 N)/2$ e cioè che il quoziente tra le due soluzioni (euristica ed ottima) è sempre minore o uguale al quoziente tra la somma di 1 ed il logaritmo in base 2 di n (con n numero delle città, ossia grandezza del problema) e 2. Risulta evidente quindi come l'efficacia di tale algoritmo sia inversamente correlata alla grandezza del problema, infatti al crescere di n l'efficacia viene meno. Per esempio nel caso in cui si avesse n= 16 si ottiene che $\log_2 16 = 4$ e dunque $(L_H/L_0) \leq (1+4)/2$, da cui $(L_H/L_0) \leq 2.5$; da quest'ultima se ne deriva che L_0 potrebbe essere in 2,5 volte la misura di L_H. Se poi n venisse posto uguale a 64 se ne deriverebbe che L_0 potrebbe risultare in 3,5 volte la misura di L_H.

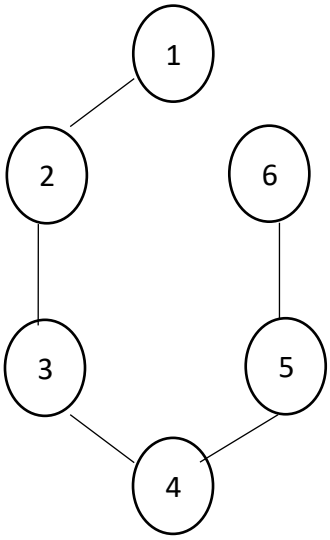
2.4.6 –TWICE AROUND THE TREE

Questo algoritmo prende il nome dal suo comportamento che prevede di andare 2 volte lungo il **'minimum spanning tree'** (ossia l'albero ricoprente minimo del grafo su cui si applica l'algoritmo), e calcolare una serie di soluzioni euristiche attraverso lo stesso. Per capire meglio il funzionamento dell'algoritmo occorre prima di tutto definire il concetto di albero ricoprente e di albero ricoprente minimo.

MINIMUM SPANNING TREE

Dato un grafo G definito da G=(V, E), lo **'spanning tree'** (o albero ricoprente) è un sotto grafo G' di G avente tutti i vertici di G e $|V|$-1 archi.
Per esempio:





GRAFO G SPANNING TREE G'

Togliendo l'arco (1, 6) si ottiene uno spanning tree, ma se si togliesse un altro arco per esempio l'arco (2, 3) si sarebbe ottenuto un altro spanning tree. In un grafo possono essere ottenuti $|E|\ C\ |V|$-1-n alberi ricoprenti con n= numero di cicli all'interno del grafo iniziale.

Viene definito **'minimum spanning tree'** lo spanning tree avente costo minimo.

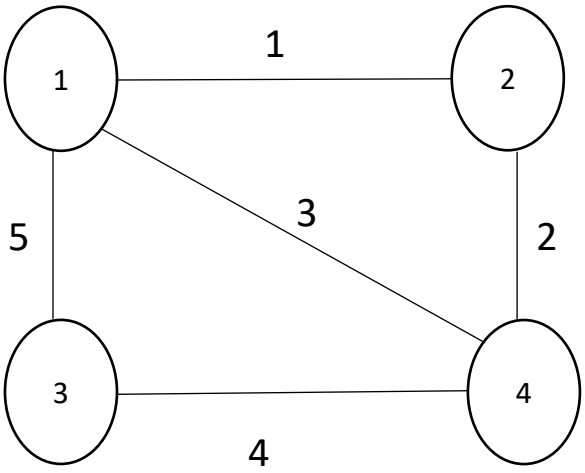
Per trovare il minimum spanning tree vengono utilizzati diversi metodi, tra i quali uno dei più utilizzati è **l'algoritmo di Prim.**

ALGORITMO DI PRIM

Secondo questo algoritmo è possibile calcolare il minimum spanning tree di un grafo pesato non orientato attraverso alcuni semplici passaggi a condizione che il grafo sia completamente connesso. Per prima cosa è necessario prendere in considerazione l'arco con peso minore con i relativi vertici collegati. Successivamente si osservano gli archi collegati ad i vertici selezionati in precedenza e si prende in considerazione l'arco con peso minore ed il successivo vertice collegato. Il procedimento va avanti fino al momento in cui tutti i vertici sono collegati e si ha un numero di archi pari a $|V|$-1.

Si osservi tale esempio.

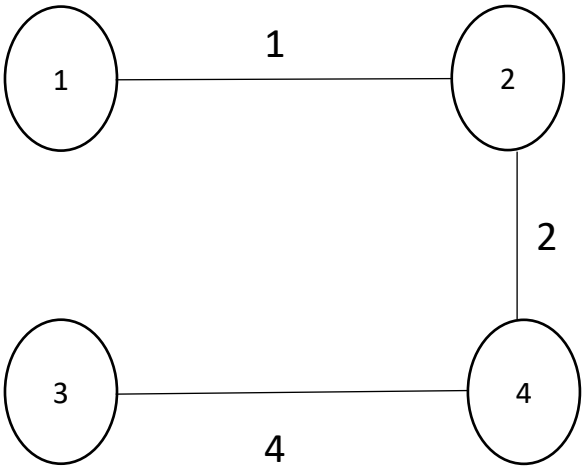
Grafo G pesato non orientato



Volendo calcolare l'mst del grafo G attraerso l'algoritmo di Prim, si parte selezionando l'arco di costo minimo con i relativi vertici collegati. Nel grafo G l'arco di costo minimo è quello che collega il vertice 1 al vertice 2, quindi viene selezionato. Successivamente si osservi che dal vertice 1 un arco uscente ed entrante in 3 ha costo 5,'arco uscente da 1 ed entrante in 4 ha costo 3, mentre il uscente dal vertice 2 ed entrante in 4 ha costo 2, quindi viene selezionato quest'ultimo con il vertice 4. A questo punto l'unico vertice che manca da collegare è il vertice 3 e lo si collega attraverso il l'arco uscente da 4 ed entrante in 3 che ha costo pari a 4.

Il minimum spanning tree del grafo G risulta quindi essere il seguente **1-2-4-3** di **costo=6.**

MINIMUM SPANNING TREE DI G



Costo= 1+2+4=6

TWICE AROUND THE TREE

Avendo ora definito il concetto di minimum spanning tree, è possibile comprendere il funzionamento dell'algoritmo **'twice around the tree'.** L'euristica in questione ruota completamente intorno alla relazione che intercorre tra la lunghezza della soluzione ottima e la lunghezza del minimum spanning tree. Per capire quale sia tale relazione si prenda in considerazione un grafo pesato non orientato composto da 5 vertici. Una volta calcolato il minimum spanning tree, viene calcolata (euristicamente) un'ipotetica soluzione al problema del commesso viaggiatore, ossia un ciclo hamiltoniano che tuttavia potrebbe non rappresentare il ciclo hamiltoniano di costo minimo. In questo momento è logico osservare che $\mathbf{L_H \geq L_0}$ poiché, essendo L_H una soluzione euristica, essa risulta essere una soluzione approssimata. Se ora venisse tolto dalla soluzione ottima un arco, verrebbe generato un albero ricoprente della soluzione ottima definito L_{st}. Logicamente, sotto la condizione per cui $\mathbf{C(i, j) \geq 0 \ \forall i,j \in G}$ (cioè il costo di ogni arco è non negativo), $\mathbf{L_{st} \leq L_0}$ ossia la lunghezza dell'albero ricoprente della soluzione ottima risulta essere inferiore rispetto alla lunghezza della soluzione ottima stessa; questo perché ovviamente l'albero ricoprente possiede un arco in meno. A questo punto è possibile dire che se $\mathbf{L_{mst} \leq L_{st}}$ (poiché logicamente la lunghezza dell'albero ricoprente minimo è inferiore rispetto alla lunghezza di un albero ricoprente generico) e $\mathbf{L_{st} \leq L_0}$, allora è ovvio che $\mathbf{L_{mst} \leq L_{st} \leq L_0 \leq L_H}$. Da questa relazione si deduce allora che ***il minimum spanning tree di un grafo è un limite minimo di L_0, ossia la lunghezza della soluzione ottima non può essere inferiore alla lunghezza del minimum spanning tree.*** Tenendo bene a mente questa relazione, è possibile calcolare diverse soluzioni euristiche tramite il minimum spanning tree. Per fare ciò, occorre prendere l'mst (minimum spanning tree) e duplicare i suoi archi per creare una sorta di ciclo euleriano. La lunghezza di tale ciclo L_e è quindi 2mst ossia 2 volte la lunghezza del minimum spanning tree ($L_e = 2L_{mst}$). Dal circuito Euleriano è possibile ora calcolare diverse soluzioni euristiche. Poiché però la soluzione al problema del commesso viaggiatore non ammette sotto circuiti, e quindi non è possibile percorrere 2 volte un singolo vertice, allora la soluzione euristica sarà composta dagli stessi vertici del circuito euleriano troncato nel punto in cui i vertici iniziano a ripetersi e chiuso nel vertice di partenza. Il vertice di partenza potrà poi essere cambiato con il vertice successivo a quello precedente. Ripetendo tale procedimento si otterranno diverse soluzioni.

Unica condizione da rispettare è quella per cui durante la creazione di una soluzione, selezionato un vertice, non è possibile scegliere quello successivo tornando indietro nella sequenza di vertici del ciclo euleriano.

L'efficacia di tale euristica anche in questo caso è data dal quoziente L_h/L_o. Osservando le caratteristiche di tale euristica, tale formula può essere implementata.

Per questo tipo di euristica si ha che:

1) $L_{mst} \leq L_0$ sotto il 1° vincolo $\mathbf{C(i, j) \geq 0 \ \forall i,j \in G}$
2) $L_e = 2L_{mst}$
3) $L_H \leq L_e$ sotto il 2^0 vincolo **C(i, j)≤ C(j, k)+C(k, i) diseguaglianza triangolare**
4) $L_H \leq 2L_{mst}$ da (2) e (3)
5) $L_H \leq 2L_0$ da (1)

Si osservi il comportamento dell'algoritmo tramite questo esempio.

Dato un grafo pesato non orientato, e la sua matrice dei costi, si trovi la soluzione al problema del commesso viaggiatore tramite l'algoritmo twice around the tree.

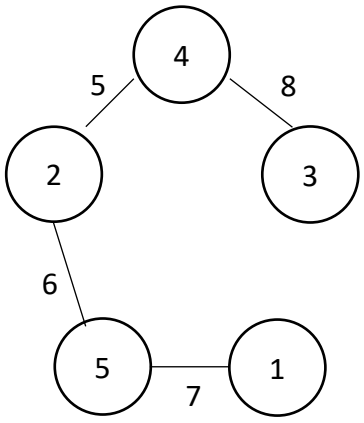
Matrice di G

–	**10**	**8**	**9**	**7**
10	**–**	**10**	**5**	**6**
8	**10**	**–**	**8**	**9**
9	**5**	**8**	**–**	**6**
7	**6**	**9**	**6**	**–**

La prima fase dell'algoritmo consiste nel calcolare il minimum spanning tree del grafo.

Questo è possibile farlo attraverso l'algoritmo di Prim che genera il seguente risultato:

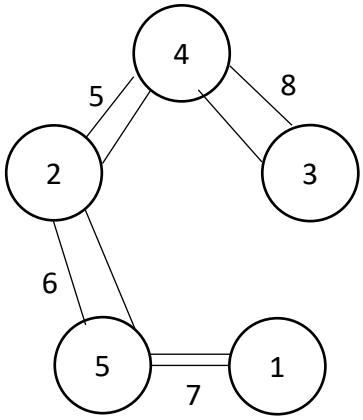
MST



COSTO= 26

Una volta ottenuto il minimum spanning tree ed il relativo costo, occorre duplicare i suoi archi per creare un ciclo Euleriano. Si ottiene dunque il seguente grafo G'.

Ciclo Euleriano di G'



COSTO=26x2=52

Grazie a questo grafo Euleriano è possibile a questo puto calcolare diverse soluzioni euristiche sfruttando uno dei circuiti Euleriani all'interno di esso. Per fare ciò si seleziona un nodo di partenza. In questo esempio il nodo selezionato è il numero 1. Ora si costruisce il circuito Euleriano intorno al vertice 1 (è un circuito Euleriano quello che passa una ed una volta soltanto per ogni arco tornando poi al vertice di partenza). Il risultato che si ottiene è il ciclo: **1-5-2-4-3-4-2-5-1**

La soluzione al problema del commesso viaggiatore tuttavia non ammette la ripetizione di uno o più vertici; è proprio per questo motivo che per creare soluzioni euristiche accettabili, si seleziona un vertice di partenza all'interno del circuito Euleriano trovato, e successivamente con i vertici consecutivi nella sequenza si crea un circuito Hamiltoniano avendo cura di non ripetere i vertici. In questo modo è possibile creare diverse soluzioni euristiche, l'unico vincolo da rispettare è quello secondo il quale, una volta selezionato un vertice, non è possibile scegliere il vertice successivo tra i vertici antecedenti a quello selezionato.

Nell'esempio è possibile calcolare le seguenti soluzioni:

A: 1-5-2-4-3-1 costo= 34

B: 1-2-4-3-5-1 costo= 39

C: 1-4-3-2-5-1 costo= 40

D: 1-3-4-2-5-1 costo = 34

….

È evidente dunque che tra queste soluzioni trovate, la D ossia quella di costo minimo, rappresenta quella migliore. Non avendo la soluzione ottima non è possibile affermare se essa combaci con la soluzione ottima, tuttavia data la relazione che intercorre tra quest'ultima ed il minimum spanning tree è possibile affermare con certezza che la D di costo 34 rappresenta un limite minimo della soluzione ottima $L_{0.}$

CAPITOLO 3 – *IMPLEMENTAZIONE DEL NEAREST NEIGHBOR ALGHORITM IN PYTHON*

Dato l'inarrestabile progesso tecnologico, che con il passare del tempo cambia giorno dopo giorno il corso della storia, l'invenzione di calcolatori automatici e computer ha avuto un forte impatto su qualsiasi campo scientifico e non dando un forte contributo al progresso. Proprio per queste ragioni l'ultimo capitolo di questo testo tratta di un programma da me realizzato in Python per la risoluzione del problema del commesso viaggiatore. L'algoritmo su cui si basa il programma, è l'euristica del **nearest neighbor** che consente dunque di ottenere in tempo polinomiale un risultato non necessariamente coincidente con l'ottimo (essendo un'euristica) ma sicuramente valido e ben approsimato. Viene mostrata in seguito la struttra del programma.

```
import sys
distanza_citta = [
 #  c0   c1  c2  c3 c4  c5
    [0,  3,  2,  7, 10, 3],  # c0
    [3,  0,  5,  1,  2, 4],  # c1
    [2,  5,  0,  8,  2, 9],  # c2
    [7,  1,  8,  0,  3, 11],  # c3
    [10, 2,  2,  3,  0, 7],  # c4
    [3,  4,  9,  11, 7, 0], #c5
]

percorso_citta = dict()

def crea_grafo_distanza_citta():
    """
    Crea il grafo necessario a trovare il percorso migliore
    """
    grafo_citta = {}
    j = 0
    for citta in distanza_citta:
        archi_citta = {i : citta[i] for i in range(0, len(citta))}
        archi_citta.pop(j, None)
        grafo_citta[j] = archi_citta
        j += 1
    return grafo_citta
```

```
def near_neighb(indice_citta, n=1):
        """
        Trova la prossima città più vicina, partendo dalla città "indice_citta"
        :param indice_citta: L'indice della città corrente
        :param n: Il numero di citta visitata (1 la prima, 2 la seconda, ...)
        """
        if len(grafo_citta) == 0:
            # Tutte le città sono state visitate
            return

        # Recupera gli archi che partono dalla città corrente
        archi_citta = grafo_citta[indice_citta]

        # Rimuove gli archi che partono dalla città corrente dal grafo
        grafo_citta.pop(indice_citta, None)

        # Trova l'indice e la distanza della città più vicina
        if n < len(distanza_citta):
            # È una città intermediaria, quindi trova quella più vicina
            distanza_prossima_citta = max(archi_citta.values())
            for k,v in archi_citta.items():
                if k not in grafo_citta.keys():
                    continue
                if (distanza_prossima_citta >= v):
                    indice_prossima_citta = k
                    distanza_prossima_citta = v
        else:
            # È l'ultima città visitata, quindi utilizza la città iniziale
            # per tornare lì
            indice_prossima_citta = citta_iniziale
            distanza_prossima_citta = archi_citta[indice_prossima_citta]

    # Aggiorna il percorso inserendo la coppia delle città e la distanza dello
    # spostamento tra queste
        percorso_citta[(indice_citta, indice_prossima_citta)] = distanza_prossima_citta
```

```
    # Visita la prossima città
        near_neighb(indice_prossima_citta, n+1)


# Crea il grafo delle città partendo dalla matrice
grafo_citta = crea_grafo_distanza_citta()

if len(sys.argv) == 1:
    # Chiede all'utente di scegliere la città di partenza
    citta_iniziale=int(input('Inserire la città dalla quale si vuole partire -> ',))
else:
    citta_iniziale = int(sys.argv[1])

# Esegue l'algoritmo "near_neighb" partendo dalla città "citta_iniziale"
near_neighb(citta_iniziale)

# Stampa il risultato
print('Il costo di ogni spostamento è:', list(percorso_citta.values()))
print(
    'Il costo del percorso che parte da',
    citta_iniziale,
    'è di:',
    sum(percorso_citta.values())
)
print('Percorso:', percorso_citta)

# Esce dal programma senza errori (exit status 0)
exit(0)
```

Il programma viene fatto partire per un totale di volte pari al numero delle città all’interno della mtarice di adiacenza. Una volta fatto ciò, si confrontano i risulati per vedere quale tra i percorsi sia quello di costo minore.

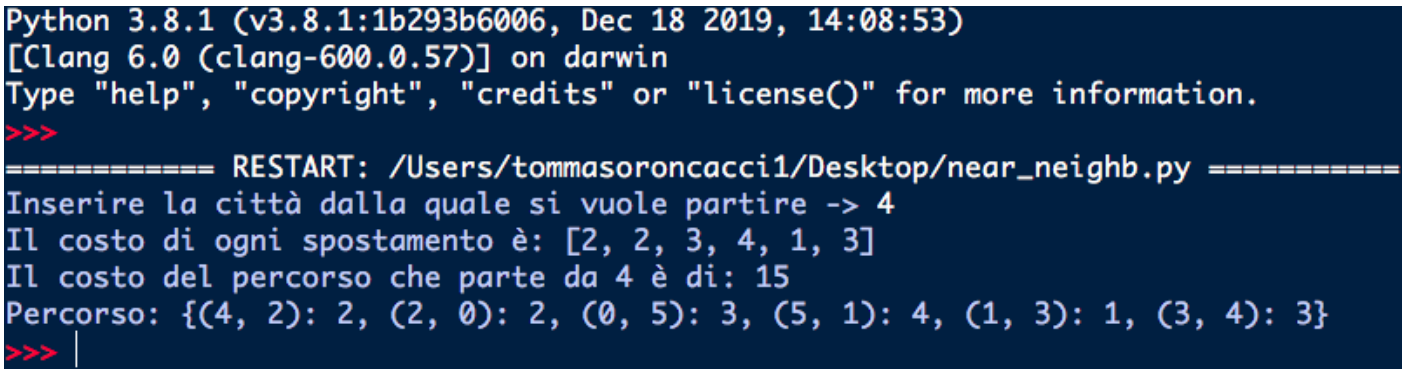
I risultati che si ottengono sono i seguenti.

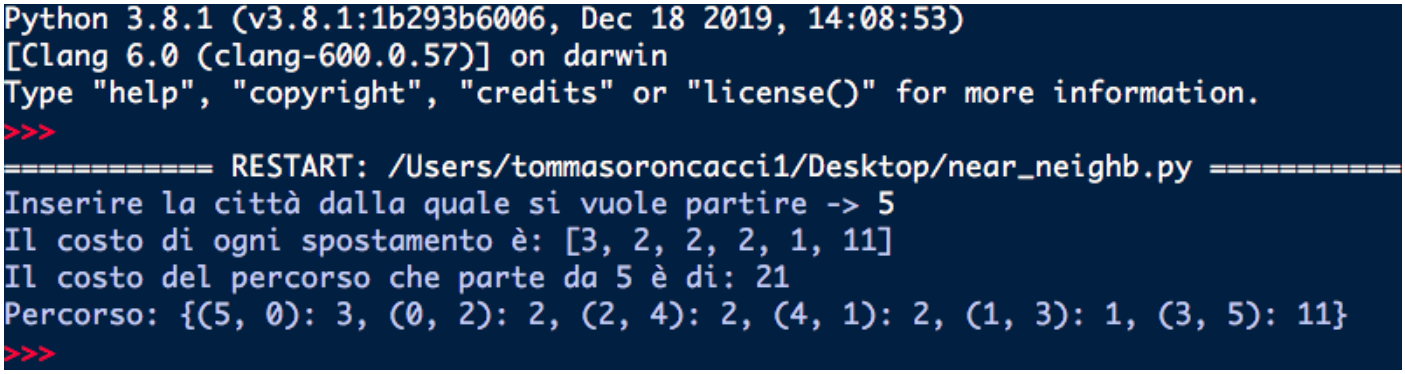
```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/tommasoroncacci1/Desktop/near_neighb.py ==========
Inserire la città dalla quale si vuole partire -> 0
Il costo di ogni spostamento è: [2, 2, 2, 1, 11, 3]
Il costo del percorso che parte da 0 è di: 21
Percorso: {(0, 2): 2, (2, 4): 2, (4, 1): 2, (1, 3): 1, (3, 5): 11, (5, 0): 3}
>>>
```

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/tommasoroncacci1/Desktop/near_neighb.py ==========
Inserire la città dalla quale si vuole partire -> 1
Il costo di ogni spostamento è: [1, 3, 2, 2, 3, 4]
Il costo del percorso che parte da 1 è di: 15
Percorso: {(1, 3): 1, (3, 4): 3, (4, 2): 2, (2, 0): 2, (0, 5): 3, (5, 1): 4}
>>>
```

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/tommasoroncacci1/Desktop/near_neighb.py ==========
Inserire la città dalla quale si vuole partire -> 2
Il costo di ogni spostamento è: [2, 2, 1, 7, 3, 9]
Il costo del percorso che parte da 2 è di: 24
Percorso: {(2, 4): 2, (4, 1): 2, (1, 3): 1, (3, 0): 7, (0, 5): 3, (5, 2): 9}
>>>
```

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/tommasoroncacci1/Desktop/near_neighb.py ==========
Inserire la città dalla quale si vuole partire -> 3
Il costo di ogni spostamento è: [1, 2, 2, 2, 3, 11]
Il costo del percorso che parte da 3 è di: 21
Percorso: {(3, 1): 1, (1, 4): 2, (4, 2): 2, (2, 0): 2, (0, 5): 3, (5, 3): 11}
>>>
```





Osservando i risultati che si ottengono è facilmente osservabile che 2 sono i percorsi che hanno costo minimo, quello che parte dalla città numero 1 e quello che parte dalla città numero 4. In questa situazione è sostanzialmente indifferente scegliere l'uno o l'altro, in quanto entrambi rappresentano soluzioni accettabili.

CONCLUSIONE

Nonostante il problema del commesso viaggiatore possa sembrare a primo impatto un enigma prettamente teorico, esso in realtà trova numerosissimi risvolti pratici in diversi settori. Il dilemma entra in gioco per esempio nel processo di foratura dei circuiti stampati; infatti tutti i componenti elettronici che fanno parte di un qualche tipo di dispositivo elettrico sono legati tra di loro attraverso ciò che viene definito un circuito stampato. Il TSP entra in gioco quando il circuito stampato deve essere forato per permettere la creazione delle così dette piste che collegano i diversi componenti di un dispositivo. Un altro tipo di applicazione riguarda l'uso di robot automatizzati. Un numero sempre più grande di magazzini oggi utilizza tali robot per disporre la merce, e tali macchine vengono programmate in modo che si spostino lungo il percorso più breve ed efficiente possibile. Anche in alcuni ristoranti straordinariamente è possibile trovare queste macchine. Tuttavia le applicazioni pratiche più comuni del TSP oggi rimangono l'utilizzo di un algoritmo esatto per spostarsi da un punto ad un altro come avviene quindi all'interno dei dispositivi di navigazione, e l'utilizzo del medesimo algoritmo per il cablaggio della fibra ottica o altri tipi di connessioni che possono essere identificate attraverso una rete. Infine credo che sebbene il TSP sia stato affrontato nel corso della storia sotto una veste altamente teorica (teoria dei grafi e ottimizzazione), in realtà tali studi siano sempre stati motivati da una componente pragmatica, derivante dalla necessità dell'uomo di rispondere a bisogni quotidiani relativi a spostamenti di persone, di oggetti e anche idee nella maniera più efficiente possibile.

BIBLIOGRAFIA

Alexander Schrijver '*Combinatorial optimization: Polyhedra and Efficiency'*. Springer, 2003.

Jhon Meier *'Groups, Graphs and Trees: An Introduction to the Geometry of Infinite Groups'.* London Mhatematical Society, 2008.

Richard J. Trudeau '*Introduction to Graph Theory'*. Parker Pub. Co, 2017

Kreher, Donald L.; Stinson, Douglas R. *'Combinatorial Algorithms: Generation, Enumeration, and Search (Discrete Mathematics and Its Application Book 7)'.* CRC Press, 2020.

Miguel Angel Avila Torres *'Data Structure & Algorithms – A Comprehensive Approach: for student and developers'*. Amazon Media, 2020.

David L. Applegate, Robert E. Bixby, Vasek Chvàtal, William J. Cook '*The Travelling Salesman Problem: A computational Study'.* Princeton University Press, 2011

SITOGRAFIA

https://www.youtube.com/watch?v=nN4K8xA8ShM&list=TLPQMjAwOTIwMjCqlNPLvsH9og&index=1

https://www.youtube.com/watch?v=LjvdXKsvUpU&t=2806s

https://www.youtube.com/watch?v=zM5MW5NKZJg

https://www.youtube.com/watch?v=4ZlRH0eK-qQ&t=698s